Deep Reinforcement Learning in Finance

Model-Free Methods, Q-Learning, and Beyond

Motivation

Why Deep Learning for RL?

Key Idea: Neural networks serve as function approximators to handle high-dimensional inputs (e.g., price series, large sets of indicators). They allow RL agents to map raw states to actions or value estimates more effectively than tabular methods.

Financial Rationale

- Markets produce complex, noisy data.
- Deep networks can uncover latent structures and patterns beyond handcrafted features.

RL Rationale

- Traditional tabular methods fail in high-dimensional or continuous state spaces.
- Deep networks facilitate scaling to large action/state domains, improving generalization.

Observation

Deep RL merges neural nets with reward-driven optimization.

Neural Network Basics (I)

Feedforward Networks

- Typically composed of multiple layers: input, hidden, and output.
- Common activation functions include Linear, ReLU, Sigmoid, and Tanh.

Forward Pass

$$\mathbf{h}^{(1)} = \sigma\big(W^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\big),$$

$$\mathbf{h}^{(2)} = \sigma\big(W^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}\big), \ldots$$

$$\mathbf{y} = W^{(L)}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)}.$$

Parameter Space

- Weights $\{W^{(l)}\}$ and biases $\{b^{(l)}\}$ define each layer.
- Typically optimized via gradient-based methods (e.g., SGD, Adam).

Relevance to RL

Value functions $Q(s, a)$ or policies $\pi_\theta(a \mid s)$ can be approximated by such layered structures.

Neural Network Basics (II)

Backpropagation

Definition: Algorithm applying chain rule to compute partial derivatives $\frac{\partial \mathcal{L}}{\partial W^{(l)}}$ and $\frac{\partial \mathcal{L}}{\partial b^{(l)}}$, where $\mathcal{L}$ is a loss function (e.g., MSE or cross-entropy).

Gradient-Based Updates

$$W \leftarrow W - \eta \frac{\partial \mathcal{L}}{\partial W}, \quad b \leftarrow b - \eta \frac{\partial \mathcal{L}}{\partial b},$$

where η is the learning rate.

Common Optimizers

- SGD, Momentum-based methods
- Adam, RMSProp (adaptive learning rates)

Practical Note

Large networks can be prone to vanishing or exploding gradients. Careful initialization (e.g. Xavier or Kaiming (He)) and normalization (e.g. BatchNorm) are widely used to address these issues.

Regularization in Deep Nets

Why Regularize?

Financial data is limited and noisy. Overfitting can lead to poor out-of-sample performance and spurious patterns.

Weight Decay

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda \sum_{l} \|W^{(l)}\|^2.$$

- Encourages smaller weight values.
- Reduces model variance.

Dropout

- Randomly zero out hidden units during training.
- Prevents co-adaptation of features (multiple neurons in a neural network develop dependencies on each other).
- Common in MLPs, CNNs, and LSTM layers.

Early Stopping

- Monitor validation metrics.
- Halt training once performance plateaus or reverts.
- In RL, reduces overfitting to a specific episode distribution.

Implication for RL

Excessive regularization might hamper the agent's ability to learn subtle signals. A balanced approach is essential to avoid both overfitting and underfitting.

Why is Overfitting a Problem?

An RL agent trained on a limited set of episodes may struggle to generalize to unseen scenarios, leading to poor real-world performance.

Specific Episode Distribution

- The agent may learn policies that work well only in a limited set of experiences.
- Limits adaptability in dynamic environments.

Early Stopping

- Monitor validation rewards or loss.
- Halt training once performance plateaus.
- Prevents excessive memorization of training trajectories.

Mitigation Strategies

- Encourage exploration using entropy regularization.
- Train on diverse environments using domain randomization.
- Use experience replay to expose the agent to varied episodes.

Implication for RL

Preventing overfitting to a specific episode distribution is needed for building RL agents that generalize effectively across different environments.

Deep Learning in Python (Skeleton)

Minimal MLP for Finance Features

Demonstration of a PyTorch pipeline for a feedforward network.

```
import torch
import torch.nn as nn
import torch.optim as optim

class MLP(nn.Module):
    def __init__(self, in_dim, out_dim):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(in_dim, 64),
            nn.ReLU(),
            nn.Linear(64, out_dim)
        )
    def forward(self, x):
        return self.net(x)
```

```
model = MLP(in_dim=10, out_dim=1)
optimizer = optim.Adam(model.parameters(), lr=1e-3)
criterion = nn.MSELoss()

for epoch in range(100):
    X, y = get_fin_data_batch()  # user function
    preds = model(X)
    loss = criterion(preds, y)

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

Usage in RL

This MLP can be extended as a Q-network or policy network in RL, with replay buffers and TD losses (for Q-learning) or policy gradients.

Multilayer Perceptron (MLP)

Basic MLP Architecture

A simple feedforward neural network for classification or regression tasks.

```
import torch
import torch.nn as nn

class MLP(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(input_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, output_size)
        )
    def forward(self, x):
        return self.net(x)
```

```
# Define model, loss, and optimizer
model = MLP(input_size=20, hidden_size=64, output_size=1)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
criterion = nn.MSELoss()

for epoch in range(50):
    X, y = get_data_batch()  # User function
    preds = model(X)
    loss = criterion(preds, y)

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

MLP Applications

MLPs are widely used in tabular data processing, reinforcement learning, and function approximation.

Hyperparameter Tuning

Key Hyperparameters

- Learning rate (η)
- Batch size
- Network depth/width

Search Methods

- Grid or random search
- Bayesian optimization
- Population-based training

Financial Twist

- Often limited historical data.
- Walk-forward or time-based splits recommended over random splits.

Practical Note

Over-tuning to one market regime yields fragile performance. Periodic retuning or "online" adaptation is often necessary in RL for finance.

Validation in Financial Context

Why Special Validation?

Financial time-series exhibit autocorrelation and changing regimes, invalidating typical i.i.d. assumptions used in standard cross-validation.

Rolling Window

- Train on a historical window (e.g., 2010–2015).
- Validate on the next segment (2016).
- Slide forward to gather multiple out-of-sample checks.

Walk-Forward Analysis

- Retrain or update the model after each validation period.
- Reflects real-world scenario where the agent adapts to new data.

Consequence for RL

An RL policy must handle non-stationary data. Thus, purely random train/val splits are misleading. Chronological splits and out-of-sample tests are more realistic.

Combining NN and RL Loops

Where Neural Networks Fit In

In model-free RL, the agent does not learn a transition model. Instead, the network typically approximates:

- **Q-function** (for DQN/Double DQN), or
- **Policy** (for policy gradient methods).

Generic RL Flow:

$$s \xrightarrow{\text{NN}} \pi_\theta(\cdot \mid s) \text{ or } Q_\theta(s, \cdot) \xrightarrow{a} \text{env.}$$

Reward r

The network's weights are updated via backprop, based on transitions (s, a, r, s').

Challenges

- Non-stationary financial data (shifting regimes).
- Catastrophic forgetting if older experiences are not revisited.
- Overfitting to specific training episodes or intervals.

Overall Flow

A trained NN-based agent can adapt to complex financial states **without an explicit model**, provided it is fed diverse experiences and robust reward signals.

Potential Pitfalls

Overtraining & Unstable Convergence

- Deep networks can memorize noise if reward signals are sparse or episodes are short.
- Financial data shifts (market regime changes) can render older parameters suboptimal.

Mitigations

- **Reward Shaping**: more frequent, smaller rewards to guide learning.
- **Periodic Retraining**: incorporate new market data.
- **Ensembles**: combine multiple networks for stability.

Data Issues

- High correlation among time steps.
- Rare extreme events (black swans) not well represented in historical data.
- Must carefully account for costs, slippage, or leverage constraints.

Financial Realism

No matter how advanced the architecture, ignoring real-world constraints (transaction costs, liquidity, risk controls) yields incomplete or misleading results.

Q&A on Deep Learning in RL

Common Questions

- **Q:** How large should a network be for DRL in finance?
- **A:** It depends on data availability, environment complexity, and compute. Oversized nets risk overfitting limited data.

Q: Can convolutions help?

- For time-series or image-like order-book data, 1D/2D CNN layers can capture local patterns.

Q: LSTM or Transformers?

- Recurrent or Transformer models may capture long-term temporal dependencies better than MLPs.
- Particularly valuable if multi-step patterns or seasonality matters.

Transition to Next Session

We now have an overview of deep learning fundamentals. Next, we shift our focus to **model-free RL**, using deep approximators without explicitly modeling environment dynamics.

Model-Free Overview

What is Model-Free RL?

Definition: An RL approach that learns policies or value functions directly from experience, without constructing a predictive model of the environment's transitions.

Why Model-Free?

- **Complexity:** In domains like finance, transition dynamics are extremely challenging to model accurately.
- **Data-Driven:** The agent adapts based on observed rewards from real or simulated interactions, bypassing explicit transition functions.

Relevance to Finance

Financial markets are partially observed and highly stochastic, making explicit environment models difficult. Model-free RL directly uses real or historical data logs to learn viable trading strategies.

Core Concepts in Model-Free RL (I)

Value Functions vs. Policy

- **Value-Based Methods:** Learn an action-value function $Q(s, a)$. The policy is then $\arg\max_a Q(s, a)$.
- **Policy-Based Methods:** Directly learn a policy $\pi_\theta(a \mid s)$ without storing full Q values.

Off-Policy

- E.g., Q-Learning uses an exploratory behavior policy (like ϵ-greedy) but converges to the greedy policy wrt Q.
- Historical data logs (collected by some other policy) can still be used.

On-Policy

- The behavior policy is identical to the one being improved (e.g., SARSA, REINFORCE).
- In finance, on-policy sampling can be expensive or risky, as each exploratory trade incurs real cost.

Mathematical Distinction

Model-free RL updates the policy or value parameters directly using (s, a, r, s') tuples, without constructing a transition function $\hat{P}(s'|s, a)$.

Temporal-Difference (TD) Learning

Idea: Update current estimates using immediate rewards plus a *bootstrap* from existing value function estimates.

$$\delta_t = r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') \; - \; Q(s_t, a_t).$$

$$Q(s_t, a_t) \; \leftarrow \; Q(s_t, a_t) \; + \; \alpha \, \delta_t.$$

Commonly used in Q-Learning updates.

Benefits

- No need to wait until the end of an episode (unlike Monte Carlo).
- Potentially faster convergence if α and exploration are well tuned.

Practical Note

Model-free RL often relies on TD learning or policy gradient. For discrete trading tasks, Q-Learning (a TD method) is a natural entry point in finance.

What is Bootstrapping?

Idea: Use existing value estimates to update other estimates, reducing variance and improving sample efficiency.

$$V(s_t) \leftarrow r_{t+1} + \gamma V(s_{t+1}).$$

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \big(r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)\big).$$

Bootstrap methods, such as Temporal-Difference (TD) Learning, rely on these updates.

Key Features

- Uses current estimates rather than waiting for full returns.
- Reduces variance compared to Monte Carlo methods.
- Common in Q-Learning and Actor-Critic algorithms.

Exploration vs. Exploitation

Defining Terms

- **Exploration:** Attempting actions that might not currently appear optimal to gather more information.
- **Exploitation:** Selecting the action that seems best based on current knowledge.

ϵ-Greedy

- With probability ϵ, pick a random action.
- With probability $1-\epsilon$, pick $\arg\max_a Q(s,a)$.

Alternatives

- Softmax/Boltzmann exploration.
- Parameter noise injected into network layers.
- Upper Confidence Bounds (UCB) from multi-armed bandit research.

Financial Angle

Random actions (ϵ-greedy) in real trading could be costly. "Safe" exploration or simulated pre-training might mitigate risk while still discovering better strategies.

Softmax Exploration

Key Idea

Instead of always selecting the best-known action, softmax exploration assigns probabilities to actions based on their estimated values, allowing exploration in a controlled way.

Softmax Action Selection

$$P(a) = \frac{\exp(Q(s,a)/\tau)}{\sum_{a'} \exp(Q(s,a')/\tau)}$$

- Higher $Q(s,a) \rightarrow$ Higher selection probability.
- Temperature τ controls randomness:
  - High $\tau \rightarrow$ More random actions.
  - Low $\tau \rightarrow$ More greedy behavior.

Advantages and Trade-offs

- Smoothly balances exploration and exploitation.
- Avoids abrupt exploration shifts seen in ϵ-greedy.
- Can struggle with sharp decision boundaries when τ is too high.
- Common in RL applications with continuous action spaces.

Practical Considerations

Softmax exploration is useful when selecting among multiple uncertain options, such as portfolio allocation in trading, but needs careful tuning of τ for stability.

Parameter Noise in Exploration

Key Idea

Instead of adding noise to actions, parameter noise perturbs network weights, leading to consistent exploration that adapts over time.

Noisy Network Formulation

$$W = W_{\text{base}} + \sigma \cdot \xi$$

- $W_{\text{base}} \rightarrow$ Learned network weights.
- $\sigma \rightarrow$ Trainable noise scale.
- $\xi \rightarrow$ Sampled noise (e.g., Gaussian).
- Noise is injected per episode, not per step.

Advantages and Trade-offs

- Enables structured exploration by modifying behavior rather than randomizing actions.
- Helps escape local optima more effectively than ϵ-greedy.
- Works well in high-dimensional action spaces.
- May require careful tuning of noise parameters for stability.

Practical Considerations

Parameter noise is particularly useful in deep RL algorithms like DDPG and PPO, where adaptive exploration is needed for continuous control tasks.

On-Policy vs. Off-Policy in Finance

Key Terminology

- **On-Policy Methods:** Improve the very policy that is used to generate experience (e.g., SARSA, REINFORCE).
- **Off-Policy Methods:** Learn about an optimal policy while following a different, exploratory policy (e.g., Q-Learning).

Off-Policy in Finance

- Suited to using historical or logged datasets that were generated by some other strategy.
- Q-Learning is off-policy, permitting the agent to learn from suboptimal or random trade data.

On-Policy in Finance

- Potentially more stable if the environment is not shifting too fast.
- Costly if real capital is at stake during exploration (the agent must “live” with its policy).

Practical Takeaway

Off-policy methods can efficiently reuse arbitrary data logs, making them attractive for many financial applications where real-time exploration is risky.

Convergence and Sample Complexity

What is Convergence?

Convergence implies the learning stabilizes to a Q-function or policy that changes negligibly with further updates.

Tabular Guarantees

- Q-Learning converges if each (s, a) is visited infinitely often and α decays suitably.
- In finance, infinite revisits to each state-action is unrealistic.

Function Approximation

- No guaranteed convergence without strong assumptions (e.g., linear function approximators).
- Neural networks may destabilize if hyperparameters or exploration are poorly tuned.

Finance Context

Due to non-stationary market behavior, we often rely on empirical validation and rolling retraining rather than strict convergence proofs.

Challenges in RL Convergence

Why Convergence is Not Guaranteed?

Convergence in RL depends on the learning rule, function approximator, and environment dynamics. Without strong assumptions, stability is not assured.

Theoretical Limits

- Q-learning with function approximation lacks formal convergence guarantees.
- Off-policy learning may lead to divergence due to **deadly triad**: function approximation, bootstrapping, and off-policy updates.
- Strong assumptions (e.g., linear models) enable proofs but limit real-world applications.

Practical Considerations

- Neural networks in deep RL require tuning to avoid instability.
- Divergence can occur due to **high variance gradients** or **poor exploration strategies**.
- Empirical success often relies on heuristics rather than strict convergence proofs.

Implication for RL

In complex environments like finance and robotics, RL models often prioritize **stability and performance metrics** over theoretical convergence.

Evaluating Model-Free Methods in Finance (I)

Performance Metrics

- **Net Profit/Loss** (cumulative or annualized)
- **Sharpe Ratio** (risk-adjusted returns)
- **Max Drawdown** (peak-to-trough decline)
- **Sortino Ratio** (focus on downside risk)

Time-based Splits

- Train on older data, validate on a subsequent segment.
- Final test on the most recent, unseen period.
- Mimics real chronological progression.

Walk-Forward

- Periodically retrain on an expanding window.
- Test on the next time segment.
- Evaluates adaptiveness over multiple regimes.

Importance of Metrics

In finance, volatility and drawdowns must be managed. RL agents should not merely maximize average return but also control risk.

Evaluating Model-Free Methods in Finance (II)

Exploration in Backtesting

Using ϵ-greedy exploration in a backtest can produce random trades that may not reflect real trading decisions.

Possible Solutions

- Decrease ϵ during later training or zero it out when testing out-of-sample.
- Maintain a separate "greedy" evaluation policy after training.

Live Trading Context

- Random trades can incur large losses in real markets.
- In practice, "safe exploration" or small position sizes during learning might be employed.

Implementation Detail

Always separate the exploratory training policy from the final evaluation policy. This ensures test metrics are not skewed by artificial exploration trades.

Sample Python Snippet for Model-Free Loop

Q-Learning with Historical Logs

Idea: Off-policy learning on historical data, where transitions (s, a, r, s') were logged by some earlier strategy or random exploration.

```
Q = np.zeros((num_states, num_actions))
alpha = 0.1
gamma = 0.99

for (s, a, r, s_next) in dataset:
    best_next = np.max(Q[s_next])
    td_error = r + gamma*best_next - Q[s, a]
    Q[s, a] += alpha * td_error

# Evaluate Q on a test set
```

- No environment stepping here; we rely on stored tuples.
- Q-Learning is off-policy, so it need not match the logging policy.
- Gaps in coverage remain an issue if the dataset lacks transitions for certain states or actions.

Limitations

Offline RL can be influcenced by limited or non-representative data. If critical state-action pairs are never logged, Q might fail in those scenarios.

Q&A on Model-Free RL

Common Questions

- **Q:** Does model-free RL ignore market microstructure or known dynamics?
- **A:** Yes, it bypasses explicit modeling. This is beneficial when dynamics are unknown, but might waste structure if it exists.

Q: Feasibility of model-based RL in finance?

- Possibly for well-studied dynamics (like certain interest rate models).
- For complex equity or derivative markets, model-free is often more flexible and practical.

Q: Which approach is simpler?

- Model-free RL is conceptually simpler; only (s, a, r, s') data is needed.
- Model-based RL requires constructing or learning $\hat{P}(s'|s, a)$, which is rarely straightforward in finance.

Transition

Next, we explore **Q-Learning (Tabular)** in depth, a fundamental model-free method and stepping stone toward deep Q-networks.

Tabular Q-Learning Fundamentals

Concept Recap

Q-Learning: A model-free RL algorithm that learns an *action-value function* $Q(s,a)$ to estimate the future cumulative reward (returns) of taking action a in state s.

$$Q(s,a) \approx \mathbb{E}\Big[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}\Big],$$

$$Q(s,a) \leftarrow Q(s,a) + \alpha\Big[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\Big].$$

- α: learning rate.
- γ: discount factor.

Exploration-Exploitation:

- ϵ-greedy ensures random actions with probability ϵ.
- Balances discovering new profitable actions and exploiting known ones.

Why Tabular First?

It's the simplest scenario to illustrate core ideas, despite being impractical for large-scale financial data.

Temporal-Difference Update

$$Q(s,a) \leftarrow Q(s,a) + \alpha\,\delta_t, \quad \text{where} \quad \delta_t = r + \gamma \max_{a'} Q(s',a') - Q(s,a).$$

TD Error δ_t

- Measures difference between current Q-estimate and a *bootstrapped* target.
- If $\delta_t > 0$, we increase $Q(s,a)$; if $\delta_t < 0$, we decrease it.

Significance

- Quick updates based on partial information, no need to wait for episode to finish.
- Convergence under certain conditions (e.g., decreasing α, sufficient exploration).

Interpretation in Finance

Reward r can be profit/loss at each time step. Q-values represent the expected return from a specific trading action sequence.

How TD Relates to Q-Learning

Q-Learning is an off-policy RL algorithm that uses the **Temporal-Difference (TD) learning** framework to update Q-values using bootstrapped estimates.

TD Update

$$V(s_t) \leftarrow V(s_t) + \alpha\big(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\big).$$

- TD learning updates **value estimates** incrementally.
- Uses **bootstrapping** (i.e., next-step estimates) instead of full rollouts.

Q-Learning as TD(0)

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big(r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)\big).$$

- Uses **TD error**:

$$\delta_t = r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t).$$

- **TD(0) approximation**: One-step lookahead for Q-values.

Why It Matters

TD learning enables **efficient updates** in Q-learning without waiting for the full episode, making it well-suited for dynamic environments like financial markets.

Exploration vs. Exploitation

Core Dilemma

To *exploit* the current best-known Q-values or to *explore* actions that might lead to higher rewards in the future?

ϵ-greedy

- With probability ϵ, pick a random action.
- With probability $1-\epsilon$, pick $\arg\max_a Q(s,a)$.

Scheduling

- Often ϵ decays over episodes.
- Start with high exploration to gather knowledge.

Risk in Finance

- Purely random actions can be costly.
- Real trading might limit exploration to "small trades" or simulation-based exploration.

Trade-Off

An agent that never explores may miss lucrative opportunities. An agent that explores too much wastes capital on suboptimal trades.

Short Python Demonstration (I)

Code Overview

Simple environment with states numbered 0 to 49, 3 possible actions. Rewards are 1 if action $==$ 1, else 0. Next state is $(s+1) \bmod 50$.

```
import numpy as np

num_states = 50
num_actions = 3
Q = np.zeros((num_states, num_actions))
alpha = 0.1
gamma = 0.99
epsilon = 0.1

def step_env(state, action):
    reward = 1 if action == 1 else 0
    next_state = (state + 1) % num_states
    done = (state == num_states-1)
    return next_state, reward, done
```

Explanation

- `Q` is a 2D array storing value estimates.
- α, γ, ϵ are hyperparameters.
- `step_env` transitions to the next state and returns reward.
- done is set to `True` at the last state for demonstration.

Short Python Demonstration (II)

Main Q-Learning Loop

Below is the training loop updating Q-values using the TD rule.

```
for episode in range(100):
    state = 0
    done = False
    while not done:
        # Epsilon-greedy
        if np.random.rand() < epsilon:
            action = np.random.randint(num_actions)
        else:
            action = np.argmax(Q[state])

        next_s, r, done = step_env(state, action)

        # TD Update
        Q[state, action] += alpha * (
            r + gamma * np.max(Q[next_s]) - Q[state, action]
        )
        state = next_s
```

Key Points

- **Choosing an action**: random with prob ϵ, otherwise exploit current Q.
- **Update Q**: use $\max_{a'} Q(\texttt{next\_s}, a')$ as the bootstrap target.
- **Iterate episodes**: gather experience in small loops, accumulate learning in `Q`.

Outcome

After sufficient episodes, the agent will learn to choose `action = 1` consistently, because that yields reward $= 1$.

Limitations in Finance (I)

High-Dimensional State Spaces

- **Tabular Explosion**: If states represent all combinations of technical indicators or asset prices, the table size becomes huge.
- **Practical Impossibility**: We can't visit every possible state sufficiently to fill the Q-table meaningfully.

Example:

$$\#\text{states} = 100 \times 100 \times 20 = 200{,}000$$

for some small discretized factors. Real markets can easily exceed millions of states.

Hence the Need for Deep Learning

- Use a neural net to approximate $Q(s, a)$.
- This yields *Deep Q-Learning* and extends to partial observability.

Conclusion

Tabular Q-Learning is only tractable for small-scale or toy finance environments. Real scenarios demand function approximation.

Limitations in Finance (II)

Non-Stationarity

- **Market Regimes:** Bull, bear, sideways. A single Q-table may become outdated if regime changes drastically.
- **Shifting Reward Distributions:** A strategy that worked last year might fail now due to new volatility patterns or sentiment.

Possible Remedies

- **Moving Window** training: discard old data.
- **Adaptive Exploration** or α-schedules.
- **Regime Detection**: maintain multiple Q-tables or specialized models for each regime.

Data Efficiency

- Q-Learning demands repeated visits to each (s,a).
- Market transitions might never exactly repeat.
- $\Longrightarrow$ Additional impetus for function approximation and robust generalization.

Reality Check

Financial time-series are rarely stationary. Tabular approaches assume fixed transition probabilities, rarely matching real markets.

Practical Note: Data Efficiency

Why a Problem?

- Each state-action pair must be visited multiple times to converge.
- Real or simulated trading data has limited coverage of rare states (extreme market crashes).

Tabular Q-Learning in Practice

- Often episodes $\times$ steps is insufficient to populate a huge Q-table reliably.
- Overfitting occurs if some state-action pairs are rarely visited.
- Large memory requirement to store the table if states are numerous.

Example

- Suppose 1 million possible states $\times$ 10 actions $\rightarrow$ 10 million Q-entries.
- Each entry updated many times is computationally heavy.

Contrast with Deep RL

Neural networks can *share* parameters across states, generalizing learned patterns to unseen states. This is more scalable for large finance problems.

Summary and Transition

Key Takeaways from Tabular Q-Learning

- Lays groundwork for the Q-learning principle.
- Illustrates the $\max$-based TD update.
- Demonstrates how exploration is integrated (ϵ-greedy).

Shortcomings

- Infeasible in large or continuous state spaces.
- Unsuitable for non-stationary finance data with complex features.

Next Step: Deep Q-Learning

- Replace the Q-table with a neural network.
- Use techniques like Experience Replay and Target Networks for stability.

Next Session

We move to **Deep Q-Learning (DQN)**—the modern extension that improves on tabular constraints and is highly relevant for financial applications.

Motivation for Deep Q-Learning

Why Move from Tabular to Deep Q-Learning?

Core Idea: In high-dimensional or continuous state spaces (as in finance), a Q-table is infeasible. Instead, a neural network approximates $Q_\theta(s, a)$, enabling generalization across unvisited states.

Function Approximation

- Replace discrete Q-table with Q_θ.
- Handle raw or partially processed data (price series, indicators).

High-Dimensional Inputs

- Market features can easily exceed 100+ dimensions.
- Deep networks learn hidden representations automatically.

Financial Rationale

- Allows the agent to discover patterns across correlated assets.
- Possibly identifies subtle signals from time-series or fundamental data.

Core DQN Mechanisms (I)

Experience Replay Buffer

- Stores transitions (s, a, r, s').
- Mini-batch sampling breaks correlation in sequential data.
- Re-uses past experiences, improving data efficiency.

Why Necessary?

- Financial time series are highly autocorrelated.
- Direct online updates cause instability in NN training.
- Replay buffer randomizes samples, resembling i.i.d. assumption in SGD.

Implementation Detail

- Typically a FIFO structure with fixed capacity (e.g., 100k transitions).
- Periodically sample mini-batches for training:

$$\{(s_i, a_i, r_i, s'_i)\}_{i=1}^{B}.$$

- Each transition is used multiple times, improving sample efficiency.

Core DQN Mechanisms (II)

Target Network

- Keep a second network with parameters θ^{-} as a slowly updated snapshot of the current Q-network.
- Reduces feedback loop where the network updates itself with constantly shifting targets.

Target Update

$$\theta^{-} \leftarrow \theta \quad \text{(every } C \text{ steps)},$$

where C is the target update frequency.

- θ: parameters being trained.
- θ^{-}: fixed copy for computing $\max_{a'} Q_{\theta^{-}}(s', a')$.

Benefit

- Stabilizes learning by reducing non-stationarity of the target.
- Without it, the network tries to chase a moving target, causing divergence.
- Commonly updated every few thousand steps in practice.

DQN Update Equations

Loss Function

$$L(\theta) = \mathbb{E}_{(s,a,r,s')\sim\text{Replay}}\Big[\big(r + \gamma \max_{a'} Q_{\theta^-}(s', a') - Q_\theta(s,a)\big)^2\Big].$$

Targets

$$y_i = r + \gamma \max_{a'} Q_{\theta^-}(s', a'),$$

used as the "label" for $Q_\theta(s,a)$.

Gradient Step

$$\theta \leftarrow \theta - \eta \nabla_\theta L(\theta),$$

where η is the learning rate.

Overestimation Bias

- $\max$ operator can inflate Q-values due to noise.
- Double DQN addresses this by decoupling action selection from evaluation.

Connection to Finance

The "label" for the Q-network includes discounted future reward. In trading, r might be instantaneous P&L minus costs, and $\max_{a'}$ finds the best subsequent action.

Python Example (I)

Basic DQN Architecture in PyTorch

Below is a DQN class approximating Q_θ.

```
import torch
import torch.nn as nn

class DQN(nn.Module):
    def __init__(self, state_dim, action_dim):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(state_dim, 128),
            nn.ReLU(),
            nn.Linear(128, action_dim)
        )
    def forward(self, x):
        return self.layers(x)
```

Features

- Input size: `state_dim`, e.g., number of features describing market condition.
- Output size: `action_dim`, e.g., discrete buy/sell/hold actions.
- Hidden layer with ReLU for nonlinearity.
- Network remains relatively small to avoid overfitting, but can be expanded for more complexity.

Alternative Layers

Convolutional or recurrent layers might be used if states are images (like order book depth maps) or time-series.

Python Example (II)

Replay Buffer

Store transitions (s, a, r, s'), then sample randomly for training to break correlation.

```
import random

replay_buffer = []
def push_to_replay(transition):
    # transition = (s, a, r, s_next)
    replay_buffer.append(transition)
    if len(replay_buffer)  10000:
        replay_buffer.pop(0)

def sample_replay(batch_size):
    return random.sample(replay_buffer, batch_size)
```

Practical Points

- **Buffer Size**: 10,000 here, but can be 1e6+ for complex tasks.
- **Sampling**: uniform random; or *prioritized* replay focusing on transitions with high TD error.
- **Memory Constraints**: large buffers require significant RAM, an issue for real-time or big data finance.

Finance Note

Experiences may come from simulated environment or a historical data “offline” scenario. Either way, randomizing them is needed for stable gradient-based updates.

Prioritized Experience Replay

Why Prioritize?

Instead of uniform sampling, prioritize transitions with **higher learning value** (e.g., larger TD error) to improve sample efficiency.

```
import numpy as np
import random

class PrioritizedReplayBuffer:
    def __init__(self, capacity, alpha=0.6):
        self.buffer = []
        self.priorities = []
        self.alpha = alpha
        self.capacity = capacity

    def push(self, transition, td_error):
        priority = (abs(td_error) + 1e-5) ** self.alpha
        self.buffer.append(transition)
        self.priorities.append(priority)
        if len(self.buffer) > self.capacity:
            self.buffer.pop(0)
            self.priorities.pop(0)

    def sample(self, batch_size):
        probs = np.array(self.priorities) / sum(self.priorities)
        indices = np.random.choice(len(self.buffer), batch_size, p=probs)
        return [self.buffer[i] for i in indices]
```

Key Features

- **TD Error-Based Sampling**: Larger errors $\rightarrow$ Higher priority.
- **Exponent α**: Controls balance between uniform and priority sampling.
- **Improves Efficiency**: Focuses updates on most informative experiences.
- **Stability Considerations**: Requires importance sampling corrections in deep RL.

Finance Context

Python Example (III)

Training Step Snippet

Demonstration of computing the DQN loss and performing a gradient update.

```
batch = sample_replay(32)
states, actions,
 rewards, next_states
 = process(batch)

with torch.no_grad():
    max_next_Q = \\ target_dqn(next_states).max(1).values

target = rewards + gamma * max_next_Q
```

```
pred = dqn(states).gather(1,
               actions.unsqueeze(1)
              ).squeeze(1)

loss = nn.MSELoss()(pred, target)

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

Key Points

- **target_dqn**: reference Q-network for stable targets.
- **gather(1, actions)**: picks Q-values of chosen actions.
- **Loss**: MSE between current Q-value and the TD target.
- **Update θ**: standard backprop through dqn.

Training Flow

Within an RL loop, we periodically sample from replay and run these steps. Then, every C steps, we copy $\theta \to \theta^-$.

Challenges in Finance

Non-Stationarity and Reward Sparsity

- **Regime Shifts**: Over time, market microstructure changes (volatility spikes, liquidity shifts).
- **Long Reward Horizons**: Significant profit might only appear after many steps, leading to sparse rewards.

Stationarity Assumption

- DQN presumes data distribution doesn't drastically shift.
- Real markets can break this assumption often.
- Periodic retraining or online updates needed.

Reward Frequency

- The agent might place trades rarely, so immediate rewards are often zero.
- Potential solution: design shaped rewards (e.g., partial credit for improved position).
- Or use certain heuristics to realize partial profits mid-episode.

Additional Solutions

Double DQN or distributional RL can help mitigate overestimation bias, while more advanced replay strategies can handle rare transitions (e.g., meltdown events).

Q&A and Summary

Key Points of DQN

- **Experience Replay** and **Target Network** are needed for stable NN-based Q-learning.
- **Loss Function**: Minimizes TD error across replayed samples.
- **Financial Challenge**: Non-stationary data, sparse rewards, risk constraints not natively handled by vanilla DQN.

Q: How to deal with large action spaces in DQN?

- If action space is too big or continuous, Q-learning might not be practical.
- Methods like DDPG or SAC handle continuous actions more directly.

Q: Is exploration still ϵ-greedy?

- Often yes, but can use parameter noise or Boltzmann exploration for finer control.

Next Steps

We will explore improvements like Double DQN, Dueling DQN, and ways to incorporate transaction costs, partial observability, and advanced risk metrics.

Double DQN: Background and Motivation

Overestimation Problem

Standard DQN: The TD target uses $\max_{a'} Q_{\theta^-}(s', a')$. Noise in Q can inflate this maximum, leading to overly optimistic estimates.

Mathematical Form

$$y = r + \gamma \max_{a'} Q_{\theta^-}(s', a'),$$

$$Q(s, a) \leftarrow Q(s, a) + \alpha(y - Q(s, a)).$$

- Noise or imprecise approximation can cause $\max$ to overshoot real value.

Consequence in Finance

- Agent might overvalue certain trades, causing excessive risk-taking.
- Volatility in P&L as Q-values jump around "optimistic" transitions.

Impact

Overestimation can make training difficult, especially in noisy financial markets with sparse reward signals.

Double DQN: Core Equation

Decoupling Selection and Evaluation

$$\max_{a'} Q_{\theta^-}(s', a') \text{ is replaced by } Q_{\theta^-}\Big(s', \arg\max_{a'} Q_\theta(s', a')\Big).$$

Explanation

- θ: parameters of the online network (used to select the best action).
- θ^-: parameters of the target network (used to evaluate that action's value).
- This separation reduces the positive bias introduced by a single $\max$.

TD Target

$$y_{\text{DDQN}} = r + \gamma\, Q_{\theta^-}\Big(s', \arg\max_{a'} Q_\theta(s', a')\Big).$$

Result

Better estimation of Q-values. Particularly relevant in finance, where frequent noise spikes might artificially inflate $\max_{a'} Q$.

Double DQN: Algorithm Sketch

Modified Steps

1. **Select action** $a_t = \arg\max_a Q_\theta(s_t, a)$ (with ϵ-greedy for exploration).
2. Observe $(s_t, a_t, r_{t+1}, s_{t+1})$ and store in replay buffer.
3. **Sample minibatch** of transitions: $\{(s_i, a_i, r_i, s_i')\}$.
4. $a^* = \arg\max_{a'} Q_\theta(s_i', a')$
5. $y_i = r_i + \gamma\, Q_{\theta^-}(s_i', a^*)$
6. **Update** θ by minimizing $\big(y_i - Q_\theta(s_i, a_i)\big)^2$.

Comparison to DQN

- θ picks the best action.
- θ^- evaluates that action, preventing over-optimism.

Update Frequency

- As in DQN, periodically copy $\theta \to \theta^-$.
- This ensures stable targets during training.

In Finance Terms

Double DQN is especially helpful if certain states (e.g., big market moves) yield large but uncertain rewards. Overestimation can be mitigated by decoupling action selection and evaluation.

Dueling DQN: Conceptual Overview

Value vs. Advantage

$$Q_\theta(s,a) = V_\theta(s) + A_\theta(s,a) - \frac{1}{|A|}\sum_{a'} A_\theta(s,a').$$

Interpretation: Decompose Q into a state-dependent baseline $V_\theta(s)$ and the advantage $A_\theta(s,a)$ of each action relative to that baseline.

Why?

- $\max_a Q(s,a)$ depends on how each action differs from the average or baseline value.
- Some states are good (high $V(s)$) regardless of action, so evaluating *which* action is best might be secondary.

Architecture Sketch

- The neural net splits into two "heads":

$$V_\theta(s) \quad \text{and} \quad A_\theta(s,a).$$

- Merges them to produce $Q_\theta(s,a)$.
- Reduces noise if many actions have similar effect in certain states.

Example in Trading

If a stock is stable and any small trades yield similar returns, $V(s)$ might be high, and $A(s,a)$ small for $\forall a$. Dueling helps separate "state quality" from "action difference."

Dueling DQN: Detailed Formula

Decomposition

$$Q_\theta(s,a) = V_\theta(s) + \Big(A_\theta(s,a) - \frac{1}{|A|}\sum_{a'} A_\theta(s,a')\Big).$$

State Value $V_\theta(s)$

- Captures overall desirability of the state.
- Independent of specific action choice.

Advantage $A_\theta(s,a)$

- Measures how much better action a is compared to the average action in state s.
- This can be negative if a is worse than the baseline.

Normalization Term

$$\frac{1}{|A|}\sum_{a'} A_\theta(s,a')$$

- Ensures identifiability: Q is unique if we subtract the mean advantage.

Learning Benefit

Backprop can distinctly update the state value part vs. the advantage part, allowing faster learning in states where actions differ minimally.

Dueling DQN: Network Architecture

Two-Stream Structure

- Common feature extractor: e.g., fully connected or CNN layers.
- Split into:
  - A **value stream** producing $V_\theta(s)$.
  - An **advantage stream** producing $A_\theta(s, a)$.
- Combine them into $Q_\theta(s, a)$ as per dueling formula.

```
class DuelingDQN(nn.Module):
    def __init__(self, in_dim, action_dim):
        super().__init__()
        self.feature = nn.Sequential(
            nn.Linear(in_dim, 128),
            nn.ReLU()
        )
        self.V = nn.Sequential(
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 1)   # single state value
        )
        self.A = nn.Sequential(
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, action_dim)
        )
    def forward(self, x):
        f = self.feature(x)
```

Key Observations

- The `feature` module extracts common representation.
- `V` head outputs a single scalar per sample.
- `A` head outputs an advantage vector of size action_dim.
- Final combination yields a q vector, same shape as standard Q-output.

Double DQN + Dueling Architecture

Often these enhancements are combined for improved stability:

- **Double DQN**: addresses overestimation.
- **Dueling**: speeds up and stabilizes value learning, especially in states where actions are similar.

Prioritized Replay

- Samples transitions with probability $\propto |\delta_t|$, where δ_t is TD error.
- Focuses updates on "surprising" or high-error experiences.

Finance Rationale

- Large changes in P&L or big price moves produce high TD error, so the agent learns from critical events more effectively.
- Regular transitions with small changes can be sampled less often.

Overall Goal

Build a more *robust* Q-approximator that handles volatile, noisy markets. Double DQN plus Dueling plus Prioritized Replay is often used as a strong baseline in DRL experiments.

Implementation Sketch: Double + Dueling DQN

Algorithm Outline

1. Initialize a Dueling DQN model Q_θ and a target Q_{θ^-}.
2. Use Double DQN update rule for action selection/evaluation.
3. Optionally use Prioritized Replay to sample transitions.
4. Periodically update target parameters: $\theta^- \leftarrow \theta$.

```
# Example pseudo-code
for each update step:
    batch = prio_replay.sample()
    (s, a, r, s_next) = batch
    with torch.no_grad():
        a_star = argmax(Q_theta(s_next))
        y = r + gamma * Q_theta_minus(s_next)[a_star]

    # Q-value for chosen action
    q_val = Q_theta(s)[a]
    loss = mse(q_val, y)

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

    if step % target_update_freq == 0:
        Q_theta_minus.load_state_dict(Q_theta.state_dict())
```

Collaboration of Techniques

- **Dueling**: network architecture for V and A.
- **Double**: $\arg\max$ from θ, but evaluate with θ^-.
- **Prioritized Replay**: high TD error transitions are re-sampled more often.
- **Target Network**: stabilize learning.

Finance-Oriented Advantages

Why These Extensions Matter

- **Financial Data** often has high variance, so *Double* approach keeps Q-values in check.
- **Dueling** quickly identifies if a state is profitable or not, ignoring action differences if they're minimal.
- **Prioritized Replay** ensures transitions with large gains/losses get more learning focus.

Volatility

- Large price jumps can produce big TD errors.
- Double DQN helps avoid "chasing" phantom large Q-values.

Stable Gains

- If in a stable uptrend, Dueling net can isolate the state value.
- Advantage signals how each action deviates from baseline.

Rare Events

- Prioritized Replay ensures rare but impactful market events are learned from repeatedly.
- Improves readiness for extreme market moves.

Practical Evidence

Many papers in DRL for finance adopt these methods to stabilize training and handle the complexity of real market data.

Frequently Asked Questions

- **Q:** How does Double DQN mitigate overestimation?
- **A:** It uses one network (θ) for $\arg\max$ action selection and the target network (θ^-) to evaluate that action, preventing optimism from a single $\max$.

Additional Pointers

- **Implementation Detail**: Combining dueling and double logic in the same model is straightforward; just apply the double update rule to a dueling architecture.
- **Hyperparameters**: Tuning learning rate, buffer size, and prioritized replay parameters (α, β) is key for stable training.

Next Steps

- Exploration of distributional RL approaches.
- Incorporation of risk metrics, transaction costs, partial observability.
- Real or paper-trading tests on historical data.

Conclusion

Double DQN and **Dueling DQN** offer important performance boosts in noisy, complex domains like finance, paving the way for more robust and efficient learning.

Why Adapt DQN for Finance?

- **Transaction Costs:** Unaccounted fees can spur excessive trading in a naive DQN agent.
- **Partial Observability:** Real markets are influenced by news, sentiment, macro data; pure price signals can be incomplete.

Cost of Trades

- Slippage: difference between expected execution price and actual fill.
- Brokerage fees, bid-ask spreads.
- In Q-learning, these should reduce reward to discourage overtrading.

Broader State Representation

- Technical indicators (moving averages, RSI, etc.).
- Fundamental data (earnings, macro variables).
- Sentiment analysis from news or social media.

Outcome

Well-designed environments and rewards reflect true P&L after costs, and incorporate hidden factors for robust policy learning.

Designing the Reward Function (I)

Components of Financial Reward

- **Profit & Loss (P&L)**: Baseline measure of trading success.
- **Cost Penalties**: Transaction fees, slippage, taxes, etc.
- **Risk Adjustments**: Include volatility or drawdown constraints.

Mathematical Form

$$R_t = \Delta \text{PortfolioValue}_t - \lambda_c \times \text{Costs}_t - \lambda_r \times \text{Risk}_t$$

- λ_c: cost coefficient.
- λ_r: risk penalty weight.

Interpretation

- $\Delta\text{PortfolioValue}_t$: net gains/losses at step t.
- Costs_t: e.g., fee per share $\times$ shares traded.
- Risk_t: could be measured by realized volatility or VaR.
- Adjusting λ_c, λ_r changes the agent's aggressiveness or risk appetite.

Balance

A purely P&L-based reward might cause reckless trading, while overly large λ_c or λ_r might paralyze the agent. Calibration is critical.

Designing the Reward Function (II)

Risk Measures

Idea: Incorporate typical finance metrics into RL to shape decisions.

Volatility Penalty

$$\text{Risk}_t = \sigma(\text{returns over window}),$$

or an exponential moving average of squared returns.

Drawdown

$$\text{Drawdown}(t) = \max_{0 \leq u \leq t} \{\text{Equity}(u)\} - \text{Equity}(t).$$

Reward might subtract a fraction of current drawdown.

VaR or CVaR

$$\text{VaR}_\alpha(X) = \inf\{x : P(X \leq x) \geq \alpha\},$$

$$\text{CVaR}_\alpha = \mathbb{E}[X \mid X \leq \text{VaR}_\alpha].$$

Harder to compute at each step, but feasible with approximation or distributional RL.

Practical Tip

Define

$$R_t = \text{P\&L}_t - \lambda_r \times \text{Risk}_t$$

to ensure the agent trades not just for returns but also for stable drawdowns.

Example: Discrete Trading Environment (I)

Pseudo-code Explanation

We show a simplified environment with discrete actions: buy, hold, sell. Reward includes transaction costs.

```
def step(state, action):
    current_price = prices[state]
    next_price = prices[state+1]

    if action == 1:  # buy
        reward = (next_price - current_price) - transaction_cost
    elif action == 2:  # sell
        reward = (current_price - next_price) - transaction_cost
    else:  # hold
        reward = 0

    done = (state+1 == len(prices)-1)
    return (state+1), reward, done
```

Notes

- `transaction_cost` might be a flat fee or proportional to trade size.
- `prices` is an array of length ≥ 2.
- In reality, "hold" might accrue opportunity cost, or carrying cost if leveraged.
- This environment is purely a stepping example; real data is more complex.

Reality Check

This toy setup omits partial fills, slippage, and other complexities. Realistic reward design is more nuanced.

Example: Discrete Trading Environment (II)

Incorporating Risk Factor

Extend the reward to penalize large drawdowns or volatility in holding positions.

```
def step(state, action, position):
    # position indicates how many shares owned
    reward = 0
    new_position = position

    if action == 1: # buy
        new_position += 1
        reward -= transaction_cost
    elif action == 2: # sell
        new_position -= 1
        reward -= transaction_cost

    # Mark-to-market PnL from old position
    current_pnl = (prices[state+1] - prices[state]) * position

    # Risk penalty: e.g., scaled by abs(new_position)
    risk_penalty = risk_lambda * abs(new_position)

    reward += current_pnl - risk_penalty

    done = (state+1 == len(prices)-1)
    return (state+1), new_position, reward, done
```

Additions

- `position` tracks inventory.
- `risk_lambda` controls how heavily the agent penalizes large positions.
- `current_pnl` is realized or mark-to-market gain from the prior step to current step.

Common Pitfalls

Overfitting to Historical Data

- Market data is *not* i.i.d.
- A policy that exploits anomalies in a single time period may fail in new regimes.

Train/Val/Test Splits

- Chronologically separate data.
- E.g. 2010-2015 (train), 2016-2017 (val), 2018-2019 (test).
- Prevents "peeking" into future data.

Walk-Forward Analysis

- Retrain periodically, then test on next segment.
- Simulates real deployment where the agent can be updated regularly.

Random Splits = Danger

- Standard random cross-validation is invalid for time-series.
- Leads to unrealistic performance estimates.

Lesson

Ensure your DRL approach is tested on truly *unseen* future data to avoid illusions of profitability.

Issue

High variance in financial returns can cause the Q-network to estimate extremely large (positive or negative) Q-values.

Possible Remedies

- **Reward Clipping**: e.g., clip reward $[-1, +1]$ or limit outliers.
- **Normalization**: scale or standardize rewards by recent volatility.
- **Double Q-Learning**: helps reduce noise-based overestimation.

Examples

- If the agent sees a "jackpot" trade, it might assign huge Q-values, overshadowing other states.
- Realistic approach: impose max daily P&L or risk-limits in environment to keep values bounded.

Trade-off

Clipping or bounding might lose some fine-grained reward detail. However, it stabilizes training in a domain with large extremes.

Key Points for Finance-Specific DQN

- **Reward Engineering**: Incorporate costs, partial P&L, risk penalties.
- **State Design**: Price + indicators + optional fundamental/sentiment data.
- **Handling Non-Stationarity**: periodic retraining, walk-forward splits.
- **Managing Volatility**: use normalization, double/dueling DQN variants.

Lesson

- The environment must reflect real trading frictions.
- The agent's objective must capture risk, not just raw returns.
- Evaluate carefully to avoid overfitting to historical quirks.

Next Topic

- Practical Implementation: hyperparameter tuning, large-scale training, parallelization.
- Real-time constraints for HFT vs. daily trading updates.

Takeaway

Adapting DQN to finance demands careful environment and reward design, ensuring the agent's learned strategy is viable under real conditions.

Hyperparameter Tuning (I)

Key Hyperparameters

- Learning rate η
- Batch size for replay
- Replay buffer size
- Target network update frequency C
- ϵ-decay or alternative exploration strategy

Parameter Ranges

- η often in $[10^{-5}, 10^{-3}]$
- ϵ-decay might go from 1.0 to 0.01 over many episodes
- Buffer sizes: from $1,000$ to $1,000,000$, depending on memory and problem complexity

Conflicts

- Large replay buffer increases coverage but slows down sampling.
- Frequent target updates increase stability but can hamper learning speed.
- Overly small batch size leads to high-variance updates.

Finance Implication

Hyperparameters strongly affect performance, especially under regime changes. A robust schedule or online adaptation may be necessary.

Training and Evaluation Process (I)

Typical Workflow

1. **Train Phase**: Learn Q-network on a historical data segment.
2. **Validation Phase**: Evaluate on a subsequent time window, adjust hyperparams or stop early if overfitting.
3. **Test Phase**: Final performance check on truly unseen data.

Walk-Forward Example

- (A) Train on 2010–2014
- (B) Validate on 2015
- (C) Test on 2016
- Then shift window: train on 2011–2015, validate on 2016, test on 2017, etc.

Benefits

- Closer approximation to real deployment.
- Captures how the policy might adapt yearly or monthly.
- Avoids using future data for training at any point.

Result

A more realistic measure of generalization across shifting market conditions, preventing "look-ahead" bias.

Training and Evaluation Process (II)

Metrics for Finance

- **Annualized Return** or total cumulative returns.
- **Sharpe Ratio** $= \frac{\mathbb{E}[R - R_f]}{\mathrm{Std}(R)}$.
- **Sortino / Calmar Ratio**, **Max Drawdown**.
- **Profit Factor** $= \frac{\text{sum of positive returns}}{\text{absolute sum of negative returns}}$.

Why So Many?

- Returns alone can be misleading if volatility is high.
- A stable but slightly lower return might be preferable to a wild high-return strategy.

Implementation

- Track all trades or P&L daily.
- Compute metric post-episode or rolling during training.

In RL Terms

- Episode-level reward might be total P&L.
- For final "test" runs, convert reward logs to finance metrics (Sharpe, etc.) for comparison.

Perspective

A strategy with a high Sharpe ratio but moderate returns can be more desirable than one with huge returns but massive drawdowns.

Scaling Up (I): Parallelization

Need for Parallelization

Financial RL may require millions of steps to converge, especially with large replay buffers and complex state spaces.

Approaches

- **Vectorized Environments:** e.g., run multiple environment instances in parallel, gather transitions quickly.
- **Distributed Training**: separate actors collecting experience from a central learner updating parameters.

Benefits

- Speeds up data collection, essential for large-scale tasks.
- More diverse market conditions can be sampled concurrently (e.g., different assets or time periods).

Implementation Tools

Frameworks like Ray RLlib provide built-in parallel sampling. Or use Python's multiprocessing with stable-baselines3 for vectorized environments.

Scaling Up (II): GPU Acceleration and Libraries

GPU/TPU for Faster NN Training

Neural net forward/backward passes can be accelerated significantly on GPUs, needed for real-time or large data RL.

PyTorch/TensorFlow

- Popular deep learning frameworks.
- Native GPU support for matrix ops.
- Large ecosystem and easy debugging.

Distributed Training

- *Data Parallel*: replicate model on multiple GPUs, each processes a batch slice.
- *Actor-Learner*: multiple actors feed transitions to a central model.

Libraries

- stable-baselines3 (Python)
- Ray RLlib
- TF-Agents

Finance Use Case

High-frequency trading or large portfolio environments benefit most from GPU-based speedups, as iteration counts can be very large.

Common Implementation Pitfalls

Key Pitfalls

- **Improper Data Splits**: Accidental leakage of future data inflates reported performance.
- **Ignoring Transaction Costs**: Leads to unrealistic frequency of trades.
- **Overly Large Neural Nets**: Overfitting due to limited or single-regime data.

Solutions

- Strict chronological train/val/test.
- Incorporate cost in reward or environment logic.
- Use dropout, weight decay, or smaller architectures if data is scarce.

Practical Warnings

- Crash risk: the agent might "discover" a path to unlimited leverage in a naive sim. Real-world constraints must be coded.
- Large memory usage from big replay buffers or parallel processes can exceed system resources.

Advice

Building a robust pipeline that includes thorough validation, performance metrics, and resource checks is essential in financial DRL projects.

Section Summary

Key Points from Sessions 6 and 7

- **Reward Crafting**: Must reflect real trading conditions (costs, risk, partial info).
- **Data Setup**: Proper time-based splits, walk-forward testing to avoid overfitting.
- **Hyperparameter Tuning**: Vital for stable and robust Q-learning solutions.
- **Scaling**: Parallelization, GPU usage, or distributed methods can handle big tasks or real-time requirements.

Impact on Finance

- Reinforcement learning can adapt to changing markets if updated regularly.
- Complexity of real markets demands careful engineering of environment and reward signals.

Looking Forward

- Distributional RL.
- Risk-sensitive RL frameworks.
- Combining advanced or partial market models with DRL.

Conclusion

With proper adaptation, DQN-based approaches can function effectively in finance, but success depends on careful design and testing against real-world complexities.

Case Study 1: Single Asset Discrete Trading

Objective

Explore a **Buy/Sell/Hold** setup on a single asset (e.g., a stock or crypto), using daily or intra-day data. Investigate performance via classical finance metrics.

Implementation Outline

- *States*: Price history window, technical indicators (moving averages, RSI, etc.).
- *Actions*: $\{0 = \text{hold}, 1 = \text{buy}, 2 = \text{sell}\}$.
- *Reward*: Realized P&L minus transaction cost. Possibly integrate a volatility penalty.
- *Algorithm*: DQN or Double DQN with a standard replay buffer.

Evaluation Metrics

- **Sharpe Ratio**:
$$\frac{\mathbb{E}[R - R_f]}{\mathrm{Std}(R)}.$$
- **Max Drawdown**:
$$\max_t\left(\text{peak}_{0\ldots t} - \text{value}(t)\right).$$
- **Net P&L** over the test horizon.

Findings

Often, a basic discrete approach can yield moderate improvements over naive buy-and-hold if the environment is well-tuned. However, large state spaces or regime shifts can still challenge the agent.

Case Study 2: Multi-Asset Allocation

Scenario

Manage a portfolio across multiple assets (e.g., equity, bonds, crypto). The action space might be discrete *allocations*, or expansions of buy/sell/hold for each asset.

Challenges

- State dimensionality grows with number of assets.
- Could use $\mathcal{O}(3^N)$ discrete actions if each asset has $\{$buy,hold,sell$\}$.
- Risk management across correlated assets (drawdown might be more complex).

Potential Methods

- **DQN with large action sets** or hierarchical RL to break down decisions by asset group.
- **Double DQN** for stable Q-values in big, volatile spaces.
- **Dueling architecture** to quickly evaluate "which assets matter in this state?"

Metric Focus

Portfolio-level Sharpe ratio, correlation between assets, and overall *Value-at-Risk* or *Expected Shortfall* can provide deeper insight than per-asset returns alone.

Final Course Summary

Core Takeaways

- **Model-Free DRL** (Q-learning, DQN, etc.) effectively bypasses the need for explicit market modeling.
- **Deep Networks** can handle high-dimensional financial states, but demand careful training (target networks, replay buffers).
- **Finance Nuances**: transaction costs, partial observability, non-stationary data, risk constraints.

Suggested Project

- Implement a Double DQN for a small basket of assets.
- Incorporate transaction costs in the environment.
- Evaluate on multiple years of data with separate train/val/test splits.
- Compare results (Sharpe, max drawdown) to a simple baseline (buy-and-hold or momentum strategy).

Research Directions

- Multi-asset RL with constraints (leverage, margin).
- *Distributional RL*: capturing the entire return distribution.
- *CVaR-based* or risk-sensitive RL: direct control over tail risks.
- *Hierarchical RL* for complex trading pipelines.

Core Q-Learning Equations

Tabular Q-Learning Update

$$Q(s,a) \;\leftarrow\; Q(s,a) \;+\; \alpha\Big[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\Big].$$

Key points:

- α: learning rate.
- γ: discount factor.

Temporal-Difference Error

$$\delta_t = r_{t+1} + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a_t).$$

Off-Policy Aspect

- Behavior policy can be ϵ-greedy.
- Learned policy is $\arg\max_a \, Q(s,a)$.

Overfitting Danger

In finance, exact repeated visits to each state-action pair are unlikely. Large or continuous state spaces motivate function approximation methods.

DQN Loss and Target Network

Deep Q-Network (DQN)

Loss function for each sampled transition (s, a, r, s'):

$$L(\theta) = \big(r + \gamma \max_{a'} Q_{\theta^-}(s', a') \; - \; Q_\theta(s, a)\big)^2.$$

Target Network

$$\theta^- \leftarrow \theta \quad \text{(periodically)}.$$

- Q_{θ^-} is a copy of Q_θ, updated slowly.
- Stabilizes learning by providing a fixed reference.

Gradient Step

$$\theta \leftarrow \theta - \eta \nabla_\theta L(\theta).$$

- Minimizes mean-squared TD error.
- η is the learning rate (Adam, RMSProp, etc.).

Interpretation

Vanilla DQN uses $\max_{a'} Q_{\theta^-}(s', a')$ for the TD target, which can cause *positive bias*. Double DQN modifies this to reduce overestimation.

Overestimation Correction

Double DQN separates action selection from action evaluation:

$$y_{\text{DDQN}} = r + \gamma\, Q_{\theta^-}\Big(s',\, \arg\max_{a'} Q_{\theta}(s', a')\Big).$$

- θ: online network parameters (selecting a^*).
- θ^-: target network parameters (evaluating a^*).
- Addresses the inflated Q-value issue from $\max$ in noisy estimates.

Update Rule

$$Q_\theta(s,a) \leftarrow Q_\theta(s,a) + \alpha\,\big[y_{\text{DDQN}} - Q_\theta(s,a)\big]$$

Benefit

Empirically and theoretically proven to temper overoptimistic Q-estimates, which is especially relevant in volatile financial environments.

Dueling DQN

Value-Advantage Decomposition

$$Q(s,a) = V(s) + A(s,a) \; - \; \frac{1}{|A|}\sum_{a'} A(s,a').$$

Value $V(s)$

- Single scalar for how good state s is, *independent of action*.

Advantage $A(s,a)$

- Measures how much better (or worse) action a is compared to other actions in s.
- $A(s,a)$ can be negative, zero, or positive.

Mean Normalization

- Subtract $\frac{1}{|A|}\sum_{a'} A(s,a')$ to keep $V(s)$ uniquely identified.
- Without it, $V(s)$ and $A(s,a)$ can be confused by additive constants.

Training

A neural network splits into two streams for $V(s)$ and $A(s,a)$, combined for the final $Q(s,a)$ used in a standard TD-loss. This often speeds learning in states where many actions yield similar outcomes.

Policy Gradient Theorem

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

where $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$ is the return.

Policy Parametrization

- $\pi_\theta(a|s)$: distribution over actions given state s.
- θ: typically neural network weights.
- Effective in continuous action spaces or large discrete spaces.

Finance Context

- Continuous controls: portfolio weights, trading volumes.
- Model-free approach if environment transition function is unknown, just observe rewards from market data or simulator.

Variance Reduction

Actor-Critic methods add a baseline $V^\pi(s)$ or $Q^\pi(s,a)$ to the gradient expression, reducing high variance in raw policy gradient estimates.

Advantage Function & Actor-Critic

Advantage-Based Updates

$$A^{\pi}(s,a) = Q^{\pi}(s,a) \; - \; V^{\pi}(s).$$

Replacing the long return G_t with A^{π} improves learning stability.

Actor-Critic Mechanism

- **Actor:** $\pi_\theta(a|s)$ picks actions.
- **Critic:** $V_\phi(s)$ or $Q_\phi(s,a)$ estimates value, providing advantage signals.
- **Update Actor:**

$$\nabla_\theta \sum_t \log \pi_\theta(a_t|s_t)\, A_t.$$

- **Update Critic:**

$$V_\phi(s_t) \;\leftarrow\; \text{TD target} - V_\phi(s_t).$$

Why in Finance?

- Large or continuous action sets (position sizes).
- Advantage function guides the actor to pick actions that outperform baseline.
- Critic focuses on stable value estimation under noisy market data.

Reduction of Variance

Subtracting $V^{\pi}(s)$ from $Q^{\pi}(s,a)$ (or from returns) is a known baseline technique to make policy gradient updates more efficient.

PPO (Proximal Policy Optimization) Objective

Clipped Surrogate

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t\Big[\min\Big(r_t(\theta)\,A_t,\ \text{clip}\big(r_t(\theta),\,1-\epsilon,\,1+\epsilon\big)\,A_t\Big)\Big],$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$.

Interpretation

- $r_t(\theta)$ is the probability ratio between new and old policies.
- PPO "clips" this ratio if it exceeds $(1 \pm \epsilon)$, preventing large updates.
- Improves training stability vs. raw policy gradients.

Finance Angle

- Drastic policy shifts can cause abrupt capital usage changes.
- PPO ensures incremental updates that avoid catastrophic changes in trading behavior.
- Helps in partially observed, volatile markets.

Benefit

PPO remains a popular on-policy actor-critic method, balancing sample efficiency and stable updates, valuable in financial RL tasks.

Risk-Based Reward Formulas

Incorporating Risk into the Objective

- **Penalize Volatility**:
$$R_t \leftarrow R_t - \lambda \cdot \sigma_t$$
- **Drawdown Penalty**:
$$R_t \leftarrow R_t - \lambda_{\text{dd}} \cdot \text{Drawdown}(t).$$

Value-at-Risk (VaR)

$$\text{VaR}_\alpha = \inf\{\, x : P(X \le x) \ge \alpha \}.$$

- RL reward can penalize crossing VaR thresholds.
- Encourages the policy to avoid tail risk events.

Conditional VaR (CVaR)

$$\text{CVaR}_\alpha = \mathbb{E}\big[X \mid X \le \text{VaR}_\alpha\big].$$

- Focuses on expected worst-case losses in the α tail.
- $R'_t = R_t - \lambda \cdot \text{CVaR}_\alpha$ might be used in a risk-sensitive RL approach.

Finance Implementation

Risk terms are often appended to the reward as negative components, controlling the agent's risk appetite. For example, $R'_t = R_t - \lambda_{\text{risk}} \times \text{Risk}(t)$.

Capturing the Full Return Distribution

Rather than learning the mean $Q^\pi(s,a)$, *Distributional RL* estimates $Z^\pi(s,a)$, the distribution of returns.

C51 Approach

$$Z_\theta(s,a) = \sum_{j=1}^{51} p_j \, \delta(z_j),$$

discrete "atoms" in the return range. Minimizes the cross-entropy or KL divergence to a target distribution.

- Encourages the agent to learn about the *entire* reward distribution, not just its expected value.

Finance Relevance

- Tail events matter greatly in trading (risk management).
- Distributional RL can directly focus on high or low quantiles for more cautious or more aggressive strategies.
- CVaR can be integrated more naturally when the distribution is known.

Quantile Regression DQN

Another distributional RL method where Z_θ is approximated by N quantiles. Well-suited to finance for tail risk analysis.

Evaluation Metrics for RL in Finance

- **Sharpe Ratio**:

$$\text{Sharpe} = \frac{\mathbb{E}[R - R_f]}{\text{Std}(R)},$$

- **Sortino Ratio** (downside-focused).
- **Max Drawdown**, **Calmar Ratio**, etc.

Sharpe Ratio

- $(R - R_f)$ is the excess return over risk-free rate.
- $\text{Std}(R)$ is the standard deviation of returns.

Drawdown

$$\text{Drawdown}(t) = \max_{u \le t} \text{Equity}(u) - \text{Equity}(t).$$

- **Max Drawdown**: $\max_t \text{Drawdown}(t)$.

RL Context

- After training, run the policy on a test environment or data set.
- Collect step-by-step returns R_t.
- Compute these finance metrics for final evaluation, ensuring robust performance, not just cumulative reward.

Conclusion

These formulas constitute the mathematical backbone of RL methods and finance metrics. Integrating them properly is vital for a robust DRL strategy in real-world trading.

Minimal Tabular Q-Learning (Setup)

Environment and Q-Table Initialization

Below is a toy environment and Q-table initialization. This demonstrates a purely functional approach.

```
import numpy as np

num_states = 5
num_actions = 3

# Q-table, all zeros initially
Q = np.zeros((num_states, num_actions))

def reset_env():
    return 0  # state=0

def step_env(state, action):
    # Simple environment logic
    reward = 0
    next_state = (state + 1) % num_states
    if action == 1:
        reward = 1  # buy
    done = (next_state == 0)
    return next_state, reward, done
```

Notes

- `step_env` increments the state index, gives a reward if `action==1`.
- `done` is True when the state loops back to 0.
- Real finance tasks would replace this with more complex logic, but the structure remains the same.

Goal

We will illustrate tabular Q-learning in a minimal code snippet.

Minimal Tabular Q-Learning (Loop)

Q-Learning Update Loop

We apply the classic TD rule inside a while loop for multiple episodes.

```
alpha = 0.1
gamma = 0.99
epsilon = 0.1
episodes = 50

for ep in range(episodes):
    state = reset_env()
    done = False
    while not done:
        # Epsilon-greedy
        if np.random.rand() < epsilon:
            action = np.random.randint(num_actions)
        else:
            action = np.argmax(Q[state])

        next_state, reward, done = step_env(state, action)
        td_error = reward + gamma * np.max(Q[next_state]) - Q[state, action]
        Q[state, action] += alpha * td_error

        state = next_state
```

Explanation

- ϵ-greedy action selection balances exploration and exploitation.
- `td_error` is the classic Q-learning TD update.
- Each episode resets the environment to `state=0`.

Result

After enough episodes, `Q` converges for this toy scenario, showing how tabular Q-learning can be coded functionally.

Simple DQN (Network Setup)

Defining a Network

We can define parameters and forward passes manually, illustrating a purely functional style in PyTorch.

```
import torch
import torch.nn.functional as F

# Define weights and biases
W1 = torch.randn((10, 32), requires_grad=True)
b1 = torch.zeros(32, requires_grad=True)
W2 = torch.randn((32, 3), requires_grad=True)
b2 = torch.zeros(3, requires_grad=True)

def dqn_forward(state_tensor):
    # state_tensor shape: [batch_size, 10]
    h = F.linear(state_tensor, W1, b1)
    h = F.relu(h)
    out = F.linear(h, W2, b2)
    return out  # shape: [batch_size, num_actions]
```

Key Points

- `W1,b1,W2,b2` are global variables storing parameters.
- `dqn_forward` is a pure function.
- Usually, we'd wrap this in `nn.Module`, but here we remain strictly functional.

Trade-Off

While this approach works, it can become unwieldy for large networks or frequent saving/loading of models.

Simple DQN (Experience Replay)

Replay Buffer

We'll store $(s, a, r, s', \text{done})$ transitions in a global list and randomly sample mini-batches.

```
import random

replay_buffer = []
max_buffer_size = 10000

def push_replay(transition):
    # transition = (state, action, reward, next_state, done)
    replay_buffer.append(transition)
    if len(replay_buffer)  max_buffer_size:
        replay_buffer.pop(0)

def sample_replay(batch_size):
    return random.sample(replay_buffer, batch_size)
```

Points

- `push_replay` inserts transitions and maintains size limit.
- `sample_replay` returns a random mini-batch for stochastic updates.
- No object-oriented approach, purely functional.

Advantage

Decoupling environment step-by-step correlation via random sampling of past experiences increases stability in DQN.

Simple DQN (Training Step)

Implementing the DQN Update

Compute a TD-target and perform a gradient step. We can do a "single-network" version first, noting overestimation risk.

```
import torch.optim as optim

lr = 1e-3
optimizer = optim.Adam([W1, b1, W2, b2], lr=lr)

def dqn_update(batch_size, gamma=0.99):
    batch = sample_replay(batch_size)
    states, actions, rewards, next_states, dones = zip(*batch)

    # Convert to tensors
    s_t = torch.FloatTensor(states)
    a_t = torch.LongTensor(actions)
    r_t = torch.FloatTensor(rewards)
    ns_t = torch.FloatTensor(next_states)
    d_t = torch.BoolTensor(dones)

    q_vals = dqn_forward(s_t)  # shape [B, num_actions]
    q_action = q_vals.gather(1, a_t.unsqueeze(1)).squeeze(1)

    with torch.no_grad():
        next_q = dqn_forward(ns_t)
        max_q = next_q.max(dim=1).values
        max_q[d_t] = 0.0
        target = r_t + gamma * max_q
```

Mechanics

- `q_action` is the Q-value for the chosen action.
- target includes $\gamma * \max Q(s')$ unless done is True.
- MSE loss w.r.t. `q_action`.
- `optimizer.step()` updates `W1,b1,W2,b2`.

Double DQN (Key Modification)

Decouple Action Selection and Evaluation

We create a "target network" by copying weights to $(W1_t, b1_t, W2_t, b2_t)$, then use the online network to choose a^* and the target network to evaluate $Q(s', a^*)$.

```
W1_t = W1.clone().detach()
b1_t = b1.clone().detach()
W2_t = W2.clone().detach()
b2_t = b2.clone().detach()

def forward_target(s):
    h = F.linear(s, W1_t, b1_t)
    h = F.relu(h)
    return F.linear(h, W2_t, b2_t)

def update_target_network():
    W1_t.copy_(W1.data)
    b1_t.copy_(b1.data)
    W2_t.copy_(W2.data)
    b2_t.copy_(b2.data)
```

Usage

- `forward_target` uses the target parameters.
- `update_target_network` is called periodically (every C steps).
- Double DQN update rule:

$$a^* = \arg\max_a Q_\theta(s', a)$$

then

$Q_{\theta^-}(s', a^*)$ in the TD target.

Result

Overestimation is mitigated by letting θ pick the best action, while θ^- evaluates its value, reducing bias.

Double DQN Update (Pseudo-Code)

Using the Target and Online Nets in One Function

We define dqn_update_double that performs the Double DQN step.

```
def dqn_update_double(batch_size, gamma=0.99):
    batch = sample_replay(batch_size)
    s, a, r, ns, d = zip(*batch)
    s_t = torch.FloatTensor(s)
    a_t = torch.LongTensor(a)
    r_t = torch.FloatTensor(r)
    ns_t = torch.FloatTensor(ns)
    d_t = torch.BoolTensor(d)

    q_vals = dqn_forward(s_t)
    q_chosen = q_vals.gather(1, a_t.unsqueeze(1)).squeeze(1)

    # Action selection with online net
    next_q_online = dqn_forward(ns_t)
    a_star = next_q_online.argmax(dim=1)

    # Evaluation with target net
    next_q_target = forward_target(ns_t)
    max_q_target = next_q_target.gather(1,
                             a_star.unsqueeze(1)).squeeze(1)
    max_q_target[d_t] = 0.0

    target = r_t + gamma * max_q_target
    loss = ((q_chosen - target)**2).mean()

    optimizer.zero_grad()
```

Steps

- a_star: action selection from dqn_forward (online).
- Evaluate that action with forward_target (target network).
- Zero out TD target for done states.
- MSE loss vs. chosen Q-value.
- optimizer updates θ (the online network).

Pseudo-Finance Step Logic

A single-asset environment: a `reset_fin` and `step_fin`.

```
prices = [100, 102, 101, 105, 106, 104, ...]
index = 0

def reset_fin():
    global index
    index = 0
    return [prices[index]]

def step_fin(action):
    global index
    old_price = prices[index]
    index += 1
    if index = len(prices):
        return [old_price], 0.0, True

    new_price = prices[index]
    reward = 0.0

    if action == 1: # buy
        reward = (new_price - old_price) - 0.1
    elif action == 2: # sell
        reward = (old_price - new_price) - 0.1

    done = (index == len(prices)-1)
    return [new_price], reward, done
```

Points

- `action=0` = hold, reward=0, no cost.
- Indices the prices array, increments `index` each time.
- `reward` is difference in price minus transaction cost (0.1).
- This is simplistic, but captures essential structure: next state, reward, done.

Bringing It All Together

Training Loop Example

We can combine the environment, replay buffer, and Double DQN or single DQN updates in a functional approach.

```
episodes = 200
batch_size = 32
epsilon = 0.1

for ep in range(episodes):
    state = reset_fin()  # e.g. [price]
    done = False
    while not done:
        if np.random.rand() < epsilon:
            action = np.random.randint(num_actions)
        else:
            s_t = torch.FloatTensor([state])
            q_out = dqn_forward(s_t)
            action = q_out.argmax(dim=1).item()

        next_state, reward, done = step_fin(action)
        push_replay((state, action, reward, next_state, done))

        state = next_state

        if len(replay_buffer) = batch_size:
            dqn_update_double(batch_size) # or dqn_update

    # Periodically update target network
    if ep % 10 == 0:
```

Steps

- ϵ-greedy action selection.
- `push_replay` storing transitions.
- `dqn_update_double` uses the replay buffer to train.
- `update_target_network()` every 10 episodes for stability.
- `reset_fin()` starts a new "episode" on price data.

Tips for Real Applications

Though we avoided classes here, large projects often benefit from object-oriented organization. Still, the logic remains similar:

State Management

- Manual indexing can get messy for multi-asset or large feature sets.
- Consider structured state arrays or Pandas DataFrames if needed.

Network and Parameters

- Direct manipulation of `W1,b1` etc. is feasible, but saving/loading requires custom solutions.
- Using `torch.save` on param tensors is possible.

Scaling Up

- Parallel environment stepping, bigger replay buffers.
- GPU acceleration by ensuring $state_t$ is on CUDA device.
- May also incorporate advanced exploration or distributional methods.