# Stochastic Process–Based Regime Detection and Adaptive Switching via Meta-Reinforcement Learning

Cole Krudwig

3rd November 2025

## Contents



**Abstract**

This project explores regime-aware reinforcement learning for financial trading by combining classical stochastic calculus tools with modern machine learning. Market regimes, such as bull, bear, and high-volatility phases, are identified using diffusion-based diagnostics, including rolling drift and volatility estimates derived from quadratic variation and related measures. A set of baseline trading strategies (momentum, mean reversion, defensive) is compared against an RL meta-controller that dynamically selects strategies conditional on the

detected regime. Using liquid equity and crypto market data, we evaluate performance across risk-adjusted metrics, highlighting the benefits of regime awareness for adaptive algorithmic trading.

In the current stage of the project, we have implemented a Gymnasium-compatible trading environment, defined a Hamilton–Jacobi–Bellman (HJB)–motivated reward based on power utility, and trained a Deep Q-Network (DQN) agent using the Stable-Baselines3 framework. We also debugged observation-shape issues induced by a lookback filtration and introduced reward scaling to make the HJB objective numerically compatible with deep RL.

# 1 Introduction

A central challenge in quantitative trading is that financial markets are not stationary: volatility, drift, and microstructure effects evolve over time and often cluster into regimes. A strategy that performs well in a low-volatility trending environment may perform poorly in a choppy, mean-reverting regime, and vice versa. This motivates *regime-aware* trading systems that adapt to the current state of the market rather than committing to a single fixed strategy.

In this project, we study a meta-control architecture in which:

1. A stochastic feature engine computes regime-relevant diagnostics from historical price data (e.g., drift, realized volatility, residuals from mean reversion, and price velocity, all standardized).
2. A small set of interpretable base strategies is implemented:
   - a momentum strategy,
   - a mean-reversion strategy,
   - and a defensive (low-risk) strategy.
3. A reinforcement learning (RL) agent acts as a meta-controller that selects which strategy to deploy at each time step, based on a filtration of stochastic features and current portfolio wealth.

Unlike many RL-in-finance setups that directly optimize terminal wealth or cumulative log-returns, we adopt an objective inspired by continuous-time optimal control. Specifically, we work with constant relative risk aversion (CRRA) power utility and shape the reward as the *change in utility*, mimicking the Hamilton–Jacobi–Bellman (HJB) objective used in classical portfolio optimization. This connects the project to the Merton-type literature, while still leveraging function approximation via deep RL.

The work reported here focuses on the implementation layer: building a stable Gymnasium environment, integrating it with Stable-Baselines3, debugging observation and reward scaling issues, and running preliminary DQN training on liquid equity data (SPY).

# 2 Related Work

## RL in Algorithmic Trading

Reinforcement learning has been applied to a range of trading problems, including single-asset market making, optimal execution, and portfolio allo-

cation. Many early works optimize simple reward signals such as per-step profit and loss (P&L) or log-returns. More recent models incorporate risk via Sharpe-ratio-like objectives or explicit drawdown penalties. Our work differs by explicitly using a utility-based objective motivated by stochastic control theory and the HJB equation, while still operating in a discrete-time, function-approximation regime.

## Regime Detection and Stochastic Features

Regime-based models are common in time series and finance, from Markov switching models to hidden semi-Markov and stochastic volatility models. In practice, regime detection often relies on engineered features such as:

- rolling estimates of drift and volatility,
- realized variance and quadratic variation,
- residuals from mean-reversion fits,
- and momentum or trend indicators.

In this project, a separate `StochasticFeatureEngine` computes a set of standardized features that serve as the observable state of the underlying stochastic process. These are not hard *labels* of regimes, but smooth diagnostics that allow an RL agent to infer regimes implicitly.

## Utility Maximization and HJB

In continuous-time portfolio theory, a classical setup is to choose a trading strategy that maximizes the expected utility of terminal wealth,

$$\mathbb{E}\left[U(W_T)\right], \qquad U(W) = \begin{cases} \dfrac{W^{1-\gamma}}{1-\gamma}, & \gamma \neq 1, \\ \log W, & \gamma = 1, \end{cases}$$

where $\gamma > 0$ is the risk aversion parameter. The associated HJB equation characterizes the value function and optimal control in a diffusion setting.

We borrow this structure by setting $\gamma = 2$ and using the CRRA power utility

$$U(W) = \frac{W^{1-2}}{1-2} = -\frac{1}{W}. \tag{1}$$

Rather than solving the HJB equation analytically, we approximate the optimal meta-control policy with a Deep Q-Network. To make this numerically feasible, we shape the per-step reward as a scaled change in utility, which avoids degenerately small gradients during RL training.

# 3 Proposed Approach

At a high level, the approach can be summarized as:

1. Preprocess raw price data (currently SPY) and construct a set of stochastic features that summarize local drift, volatility, and mean-reversion structure.
2. Implement three base trading strategies (momentum, mean reversion, defensive) that map the current price trajectory into a position or return.
3. Wrap these components in a Gymnasium environment that:
   - tracks portfolio wealth,
   - exposes a filtered observation of features and wealth to the agent,
   - and uses an HJB-inspired utility change as the reward.
4. Train a Deep Q-Network that, at each time step, chooses one of the three base strategies as the control action.

## 3.1 Observation Design and Filtration

The environment state is built to approximate a filtration $\mathcal{F}_t$ generated by the underlying stochastic process. We do this via:

- A vector of standardized stochastic features at time $t$:

  $X_t = \big($Drift_Short_Z$_t$, Drift_Long_Z$_t$, RV_Signal_Z$_t$, Vol_Trend_Z$_t$, MR_Residual_Z$_t$, Price_V

  constructed by the `StochasticFeatureEngine`.

- A normalized wealth feature

$$w_t^{\text{norm}} = \frac{W_t}{W_0},$$

  where $W_0$ is the initial wealth.

We define a base observation at time $t$ as

$$o_t^{\text{base}} = \big(X_t,\, w_t^{\text{norm}}\big) \in \mathbb{R}^7, \tag{2}$$

and then construct a lookback stack of length $L$ to encode a short history of the process:

$$o_t = \big(o_{t-L+1}^{\text{base}}, \ldots, o_t^{\text{base}}\big) \in \mathbb{R}^{7L}. \tag{3}$$

In our current implementation, we use a lookback of $L = 5$, so the observation dimension is $7 \times 5 = 35$. This temporal stacking is implemented as an internal history buffer in the `HJBTradingEnv`, which is updated on every step and initialized at reset by repeating the initial base observation.

## 3.2 Meta-Controller over Base Strategies

The action space of the environment is discrete with three actions:

- $a = 0$: choose the momentum strategy,
- $a = 1$: choose the mean-reversion strategy,
- $a = 2$: choose the defensive strategy.

These actions are implemented in a separate `TradingStrategies` class via a method:

$$r_t^{\pi(a)} = \texttt{calculate\_strategy\_returns}(t, a), \tag{4}$$

which returns the portfolio return for the selected strategy at step $t$. The environment updates the wealth process according to

$$W_{t+1} = W_t\big(1 + r_t^{\pi(a)}\big), \tag{5}$$

and uses the updated wealth to compute utility and reward.

### HJB-Motivated Reward and Scaling

We define CRRA power utility with risk aversion parameter $\gamma = 2$:

$$U(W) = -\frac{1}{W}, \qquad W > 0. \tag{6}$$

At each step, the raw HJB-inspired reward is the change in utility,

$$\Delta U_t = U(W_{t+1}) - U(W_t). \tag{7}$$

For realistic magnitudes of wealth and small per-step returns, $\Delta U_t$ is extremely small (on the order of $10^{-8}$), which created numerical issues for DQN training. To address this, we introduce a scalar reward multiplier $\alpha > 0$:

$$r_t = \alpha\, \Delta U_t = \alpha\big[U(W_{t+1}) - U(W_t)\big], \tag{8}$$

with $\alpha$ currently set to $10^8$. This preserves the HJB structure but rescales the signal into a numerically meaningful range for gradient-based RL.

Additionally, the environment imposes an early termination penalty if wealth falls below 10% of the initial wealth:

$$\text{if } W_t < 0.1\, W_0, \quad \text{then } r_t \leftarrow r_t - 100, \quad \text{episode terminates.} \tag{9}$$

# 4 Implementation

## 4.1 Gymnasium Environment Design

We implement an environment `HJBTradingEnv` that subclasses `gym.Env` from the Gymnasium API. Key components are:

- **Action space:** `spaces.Discrete(3)` corresponding to the three base strategies.
- **Observation space:** `spaces.Box` with dimension 35 and type `float32`, representing the lookback stack of seven base features over five steps. The bounds are chosen as

$$\text{features} \in [-5, 5], \qquad \text{wealth} \in [0, \infty),$$

  replicated across the lookback window.
- **Internal state:**
  - `current_step`: integer index into the merged data frame,
  - `current_wealth`: current portfolio value,
  - `last_utility`: $U(W_t)$ from the previous step,
  - `obs_history`: list storing the last $L$ base observations.

The `reset` method initializes the environment by:

1. Selecting a starting index `current_step` within the data after a burn-in period, to avoid edge effects from feature computation.
2. Setting `current_wealth` to `INITIAL_WEALTH`.
3. Computing the initial base observation and repeating it $L$ times to initialize `obs_history`.
4. Returning the stacked observation and an empty `info` dict.

The `step` method:

1. Increments `current_step`.
2. If the end of the data is reached, terminates the episode and returns the final observation with zero reward.

3. Otherwise, computes the strategy return via `TradingStrategies.calculate_strategy_ret` updates wealth, and computes the scaled HJB reward.
4. Checks for ruin (wealth below a threshold) and applies a large penalty with early termination if triggered.
5. Updates the observation history by dropping the oldest base observation and appending the newest one.
6. Returns the stacked observation, reward, termination flag, no truncation, and an empty `info` dict.

We also added run-time assertions to verify that the shape of the returned observation matches the declared `observation_space`, which helped debug mismatches when modifying the lookback logic.

## 4.2 Stable-Baselines3 Integration

For training, we use Stable-Baselines3's DQN implementation:

- The environment is wrapped in a `Monitor` for logging and then in a `DummyVecEnv` to provide the vectorized interface expected by SB3.
- We construct separate training and testing slices of the data within a factory function `make_trading_env`, although for the current working setup we primarily use the training slice for both learning and quick evaluation.

The DQN model is instantiated as:

```
model = DQN(
    policy="MlpPolicy",
    env=train_env,
    learning_rate=1e-4,
    buffer_size=100000,
    learning_starts=1000,
    batch_size=64,
    tau=0.1,
    gamma=0.99,
    train_freq=4,
    target_update_interval=1000,
    exploration_fraction=0.2,
    exploration_final_eps=0.05,
    verbose=1,
)
```

We use a `CheckpointCallback` to periodically save model checkpoints and the replay buffer during training. After training for a fixed number of time steps, we:

1. Save the final model to disk.
2. Run a single evaluation rollout using the same vectorized environment `train_env`, taking greedy actions with `model.predict`.
3. Accumulate the total reward and extract the final wealth by unwrapping the underlying `HJBTradingEnv` from the `Monitor`.

This produces a concise summary of the agent's performance at the end of training:

```
=== Evaluation Summary (train slice) ===
Final Wealth: 10000.00
Cumulative Reward: -0.000013
```

Prior to introducing reward scaling, the cumulative reward was effectively numerical noise; after scaling, the signal becomes more interpretable, though further tuning and evaluation are still needed.

# 5 Future Work

The current implementation establishes the core infrastructure for HJB-guided meta-reinforcement learning in a single-asset setting. Several directions are planned:

- **Richer Evaluation:** Move beyond a single evaluation rollout on the training slice to:
  - evaluate on a held-out test period,
  - compare against fixed baselines (always-momentum, always-mean-reversion, always-defensive),
  - and estimate risk-adjusted metrics such as Sharpe ratio and maximum drawdown.
- **Improved Reward Design:** Experiment with:
  - alternative risk-sensitive rewards (e.g., log-wealth increments, drawdown penalties),
  - different CRRA parameters $\gamma$,
  - and multi-objective formulations combining utility and variance penalties.
- **Multi-Asset Extension:** Generalize the environment to a portfolio of assets, where the meta-controller allocates across multiple strategies and symbols rather than selecting a single strategy for a single asset.
- **Hyperparameter and Architecture Tuning:** Systematically tune the DQN architecture and training hyperparameters, and compare with other algorithms such as PPO or SAC that may better handle continuous-valued rewards and potentially continuous action spaces.
- **Explicit Regime Labeling:** Augment the current feature-based approach with explicit regime labels (e.g., via clustering or hidden Markov models) to study whether explicit or implicit regime representations lead to better performance.

# 6 Conclusion

We have constructed a regime-aware trading environment that combines stochastic feature engineering, interpretable base strategies, and a meta-controller trained with deep reinforcement learning. The key contribution

at this stage is the implementation of an HJB-inspired reward signal based on power utility, and its integration into a Gymnasium environment compatible with Stable-Baselines3.

In designing the environment, we encountered and resolved several practical issues that are typical in RL-for-finance projects: aligning observation shapes with a temporal lookback filtration, correctly interfacing custom environments with vectorized RL libraries, and scaling theoretically motivated but numerically tiny reward signals. The current DQN agent, trained on SPY data, does not yet exhibit strong wealth-maximizing behavior, but the infrastructure is now in place to iterate on reward design, strategy specification, and algorithmic choices.

Overall, this work lays the groundwork for a more ambitious exploration of meta-reinforcement learning for regime-aware trading, where tools from stochastic calculus and HJB theory inform both the feature and reward design, and modern deep RL provides a flexible function-approximation engine for complex market environments.