Dynamic Portfolio Allocation Using Reinforcement Learning

By Jayden Udall

May 25, 2026

View code on GitHub

Introduction

Financial portfolio allocation is fundamentally a sequential decision problem. An investor must repeatedly choose how to allocate wealth across risky assets without knowing future returns. Tomorrow's market move is random and the consequences of today's allocation are only partially observed through noisy daily returns.

The concept of dynamic portfolio allocation through machine learning algorithms is fascinating: can an algorithm learn the best course of action given a current market state and consistently outperform baselines? The hardest part of this problem is choosing an effective model to learn what is random by nature. Traditional supervised learning models, such as linear regression, fail because financial markets operate on nonlinear dynamics, unpredictable human behavior, and the assumption that all public information is instantly priced in (the Efficient Market Hypothesis). To provide context, here are a few primary reasons why regression models fall short in performance:

1. **Random Walk Nature:** Stock prices typically follow a random walk, meaning today's best predictor of tomorrow's price is simply today's closing price. Regression models assume historical trends will mathematically repeat.
2. **Non-Linear Relationships:** Linear regression algorithms assume a constant, straight-line relationship between the variables. Stock markets react to events in highly non-linear, unpredictable, and exponential ways.
3. **Unquantifiable Events:** Regression requires structured, numerical data. It cannot model unexpected macroeconomic events, global crises, unannounced CEO changes, or investor sentiment.
4. **The Outlier Problem:** Outliers (such as sudden flash crashes or extreme market spikes) disproportionately skew regression lines, severely reducing prediction accuracy.
5. **Overfitting:** Models can perfectly fit historical test data, but market conditions change constantly, causing these models to completely fail when making future predictions

In light of this, we must look for alternative methods to portfolio allocation. This project studies reinforcement learning (RL) for daily rebalancing among three exchange-traded funds (ETFs): SPY (tracks the S&P 500), TLT (long-term Treasuries), and GLD (gold), using historical data from 2005–2025 taken from Yahoo Finance. In order to better familiarize myself with RL concepts, I will implement the environment and RL models from scratch.

Problem Framing

The key idea behind the approach I am exploring is what is known as a Markov decision process (MDP), in which an agent interacts with its environment over a series of steps by choosing actions that maximize long-term rewards. The process is defined by states (the current situation), actions (the choices available), transition probabilities (the likelihood of moving to a new state based on an action), and rewards (the feedback given for those transitions). The defining characteristic of an MDP is the Markov Property: the future state depends only on the current state and action, not on the sequence of events that preceded it.

Rather than forecasting returns with a supervised model and optimizing weights in a separate step, we will treat allocation as an MDP: at each trading day the agent observes a state summarizing recent market conditions and its current holdings, selects a discrete portfolio action, receives a stochastic reward based on the next day's realized return (minus transaction costs), and transitions to a new state. The objective is to learn a policy that maximizes long-run cumulative reward, or equivalently, compounded portfolio growth.

While it is true that the Markov property is violated due to the inclusion of recent conditions (rolling returns, volatility, rough regime discretization) in the current state, we are effectively engineering a state that gives us an approximate MDP. The state is treated as a "current state" and used to make allocation decisions for the next day. Our reward depends on the current allocation and next day returns:

$$"what\ I\ held\ today" + "what\ the\ market\ looks\ like\ today" \rightarrow tomorrow's\ wealth\ change$$

I divided the experiment into four separate phases:

Phase I: Design the environment that the agent will explore. Define features, actions, rewards, etc.

Phase II: Implement and train a tabular Q-Learning agent (simplest reinforcement learning model)

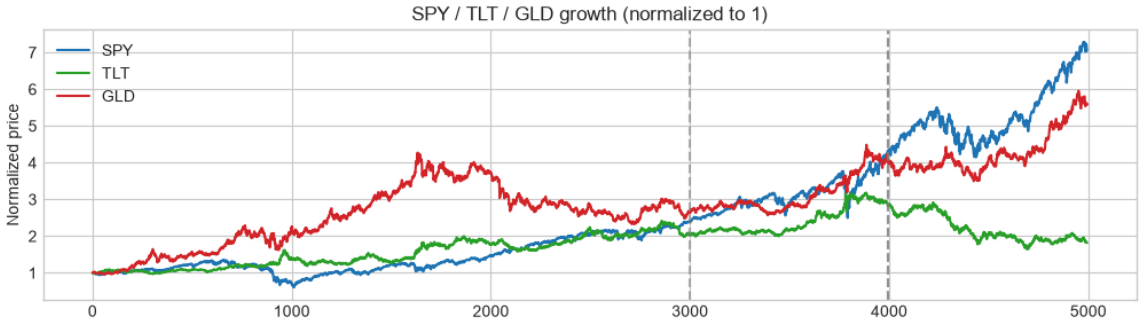
Phase III: Implement and train a Deep Q-Learning (DQN) agent that uses a neural network to approximate expected reward

Phase IV: Experimentation, finetuning, and evaluation against buy-and-hold baselines

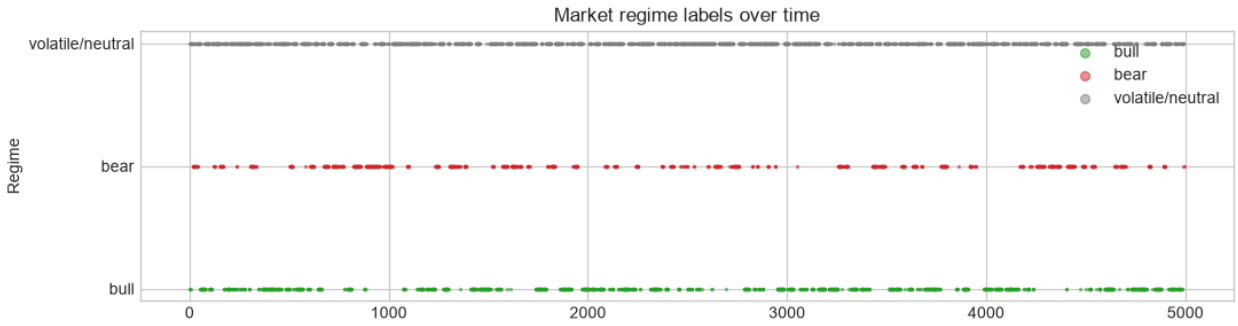
Phase V: Compare the two previous approaches to a Proximal Policy Optimization (PPO) model from Stable Baselines 3

Phase I: Environment Design

I modeled daily portfolio rebalancing as a Markov decision process implemented in a custom PortfolioEnv Python class. The environment steps through historical trading days from a preprocessed dataframe of SPY, TLT, and GLD prices and features. At each timestep the agent observes the current state, chooses a discrete allocation action, receives a reward based on the next day's realized portfolio return, and advances one day. An episode runs from a configurable start date to an end date and the last usable decision day is set to n - 2 because rewards depend on returns on day t + 1.



State. The observation vector has 19 dimensions. Twelve are market features: 1-, 5-, and 20-day returns and 20-day rolling volatilities for each of the three ETFs. Three dimensions encode the current market regime as a one-hot vector (bull, bear, or volatile/neutral), assigned from SPY momentum and volatility thresholds fit on training data only. The remaining four dimensions are the portfolio weights from the previous action: SPY, TLT, GLD, and cash. Those weights are included so the state reflects what the agent currently holds, not just what the market looks like. Before training, market features are normalized using mean and standard deviation computed on the training window; regime one-hot and weights are left as-is.



Actions. Actions are discrete portfolio templates rather than free continuous weights. For example, with four actions the agent can choose all SPY, a 50/50 SPY/TLT mix, all TLT, or all GLD. Each action maps to a fixed 4-vector of weights $[w_{SPY}, w_{TLT}, w_{GLD}, w_{cash}]$ that sum to one. The environment starts in cash unless a different initial action is specified on reset. In this project, we will analyze the results on action vectors of size 4, 6, and 10. The more available actions, the more options the agent will have to choose from to maximize expected reward.

To give extra context, the 10-action vector is defined as:

```
actions = {
     0: [1.0,0.0,0.0,0.0],   # all SPY
     1: [0.75,0.25,0.0,0.0],
     2: [0.50,0.50,0.0,0.0],
     3: [0.25,0.75,0.0,0.0],
     4: [0.0,1.0,0.0,0.0],   # all TLT
     5: [0.50,0.0,0.50,0.0],
     6: [0.0,0.50,0.50,0.0],
     7: [0.0,0.0,1.0,0.0],   # all gold
     8: [0.33,0.33,0.34,0.0],
     9: [0.0,0.0,0.0,1.0],   # all cash
 }
```

Reward. After action a is taken on day t, the reward is the next-day portfolio return minus a transaction cost on turnover:

$$r_t = \sum_{i \in \{SPY, TLT, GLD\}} w_i \cdot r_{i,t+1} - c \cdot \sum_j |w_j - w_{j,prev}|$$

or

$$reward = nextPortfolioReturn - (transactionCost \times turnover)$$

Cash earns approximately zero return. The turnover penalty discourages excessive rebalancing and makes the reward depend on both the new allocation and the previous one.

Episode control. The environment supports slicing the dataframe by start and end indices, so the same class can represent train, validation, or test periods. On reset, the timestep and previous action can be set explicitly, which allows starting episodes at random dates or fixed windows. Each step updates the internal timestep, stores the chosen action as the new holdings, computes reward, and returns whether the episode has terminated.

Together, these pieces define a self-contained simulation: market history becomes states, allocation choices become actions, and realized next-day profit becomes the learning signal that drives the learned decision-making over time.

Phase II: Tabular Q-Learning

Tabular Q-learning is our first and simplest RL method for portfolio allocation. Instead of using the full continuous feature vector, we discretize market conditions into a small set of interpretable states. Each day is labeled with a regime (bull, bear, or volatile/neutral) from SPY momentum and volatility, then momentum and volatility are each bucketed into low, medium, or high. Combining regime (3 levels), momentum (3), and volatility (3) gives 27 market states, stored in the tabular_state column. Because the reward depends on both market conditions and current holdings, the agent uses an augmented state that also includes the previous allocation action. With 4 discrete actions, this yields 108 distinct state indices (27 × 4), each mapped to a single integer via market_state × num_actions + previous_action.

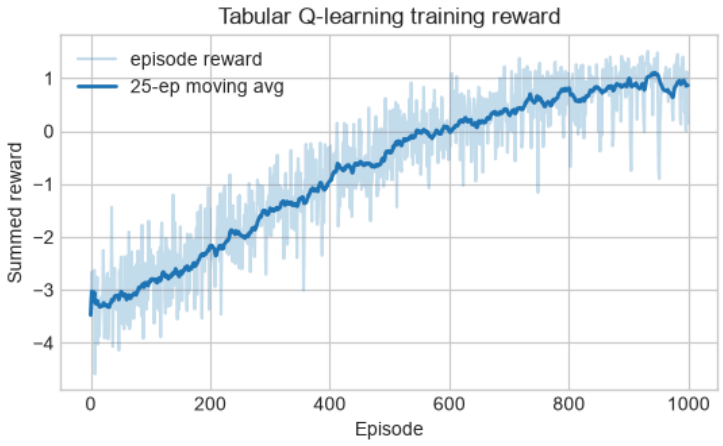
The model itself is a Q-table: a 108 × 4 array where $Q[s, a]$ estimates the expected discounted future return from taking action a in augmented state s. The agent chooses among four fixed portfolio templates, the same allocations used elsewhere in the project (e.g., all SPY, 50/50 SPY/TLT, all TLT, all GLD). Learning follows the standard Q-learning update rule. After observing state s, taking action a, receiving reward r, and transitioning to s', the table is updated as:

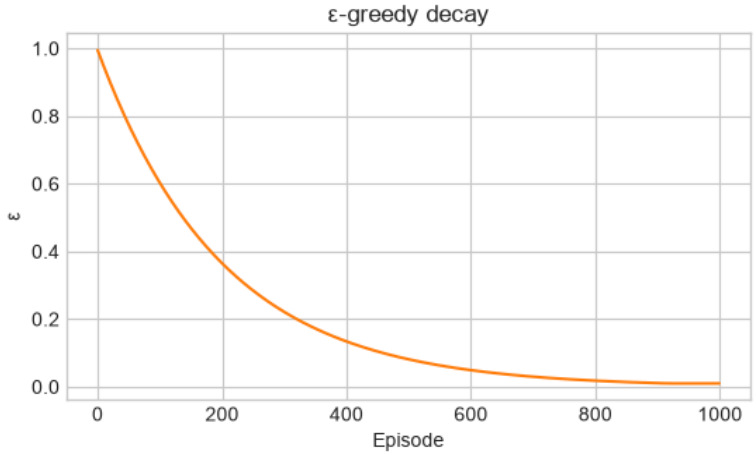
$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha\,[R + \gamma \max_{a'} Q(s', a')]$$

where α = 0.1 is the learning rate and γ = 0.96 is the discount factor.

Initial training process. Training runs for 1,000 episodes over the training window (roughly 2005–2021, an 80/20 train/test split). Each episode walks day-by-day through PortfolioEnv from the start of the training period to the end.

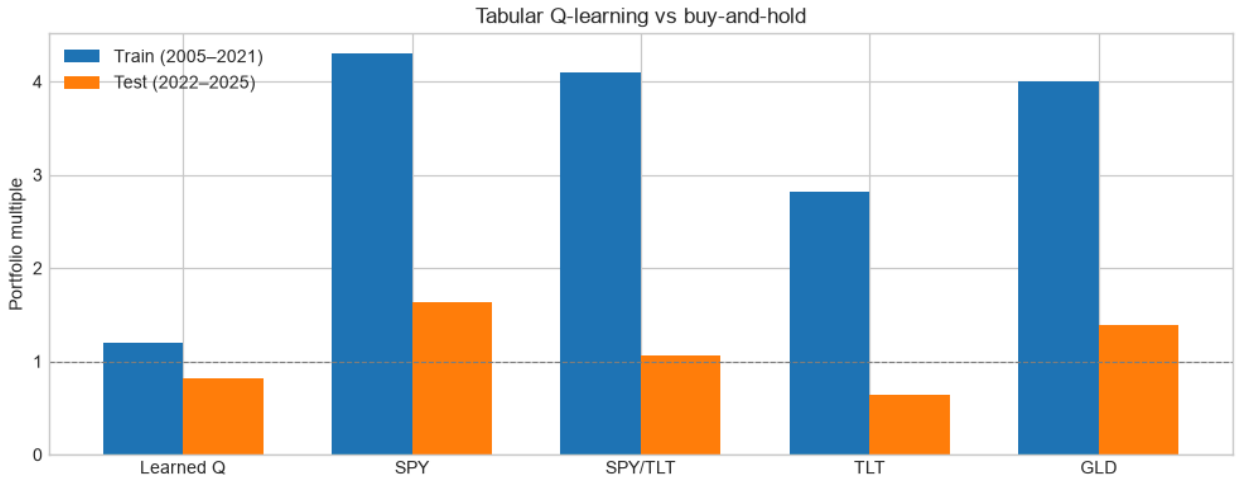
At each step, the agent reads the current tabular_state and previous action, forms the augmented state index, and selects an action using ε-greedy exploration: with probability ε it picks a random action, otherwise it picks the action with the highest Q-value for that state. The environment returns the next-day portfolio reward (return minus transaction costs on turnover), and the Q-table is updated.





After each full pass through the training window, ε is decayed by a factor of 0.995 down to a minimum of 0.01, so early training emphasizes exploration and later training emphasizes exploitation. Episode logging tracks total summed reward and the current ε value.

Evaluation. After training, the learned policy is extracted by taking argmax over actions for each row of the Q-table. Performance is measured with a greedy backtest (backtest_env): the agent walks through train or test environments without exploration, compounds daily rewards as portfolio_value × (1 + r), and reports final portfolio multiple starting from $1. The agent initializes in action 3 (all GLD) at the start of each backtest. This separates learning (noisy, exploratory) from evaluation (deterministic, compounding wealth), which is the standard way to report portfolio performance in this setup.



The main strength of tabular Q-learning here is transparency: every state-action value is stored explicitly and the policy is easy to inspect. The main limitation is discretization, where rich continuous information is collapsed into 27 buckets, so the agent cannot distinguish fine differences within a bucket. That tradeoff keeps the method tractable as a baseline but caps how much detail it can learn from the data. This is evident in the first training run, which gave us a learned final portfolio value of $0.81, meaning we lost about 19 cents in the testing period of 2022-2025 when starting with a portfolio of $1.

Phase III: Deep Q-Network (DQN)

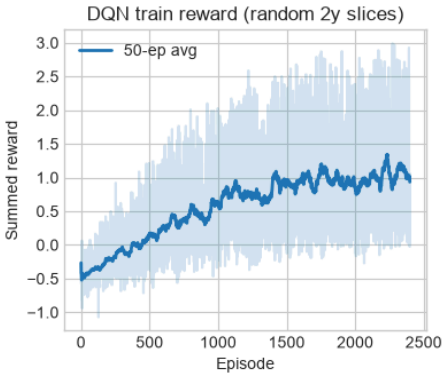
DQN is the second RL method in this project and replaces the tabular Q-table with a neural network that can handle continuous market features. Instead of 27 discrete buckets, the agent observes the full 19-dimensional state from PortfolioEnv: 12 normalized market features (1-, 5-, and 20-day returns and volatilities for SPY, TLT, and GLD), a 3-dimensional one-hot regime label, and the 4 portfolio weights from the previous action. The action space remains discrete so the network's job is to map each continuous state to a Q-value for every available action and pick the best one.

Implementation. The model is a small multilayer perceptron (Q-Network) with two hidden layers of size 64 and ReLU activations. It outputs one Q-value per action. We keep two copies of this network: an online policy network that is updated every step, and a target network that is held fixed for longer stretches and periodically copied from the policy network (every 500 gradient steps). Transitions (s, a, r, s', done) are stored in a replay buffer of capacity 100,000 and sampled in minibatches of 64, which breaks temporal correlation in the sequential market data. The learning update uses Double DQN: the policy network chooses the next action, $a^* = \arg\max_{a'} Q_{policy}(s', a')$, while the target network evaluates that action's value. The loss is Smooth L1 ($Loss = (Q_{policy}(s, a) - y)^2$) between the predicted Q and

$$y = r + \gamma, Q_{target}(s', a^*) \cdot (1 - done)$$

with $\gamma = 0.98$ (roughly a 50-trading-day effective horizon). Gradients are clipped to a max norm of 1.0, and the optimizer is Adam with learning rate (10^{-3}). Actions during training are chosen with ε-greedy exploration; at evaluation time the agent always takes the greedy action.

Data split and initial training process. Unlike tabular Q's 80/20 split, DQN uses 60% train / 20% validation / 20% test (roughly 2005–2016 / 2017–2021 / 2022–2025). Feature normalizers are fit on the train window only and then copied to val and test. We use a variable number of episodes and three different approaches to training within each episode: one where the model is trained on the full test set, one where the model is trained on a random starting point, and one where the model is trained on random windows of time (either 1y or 2y).

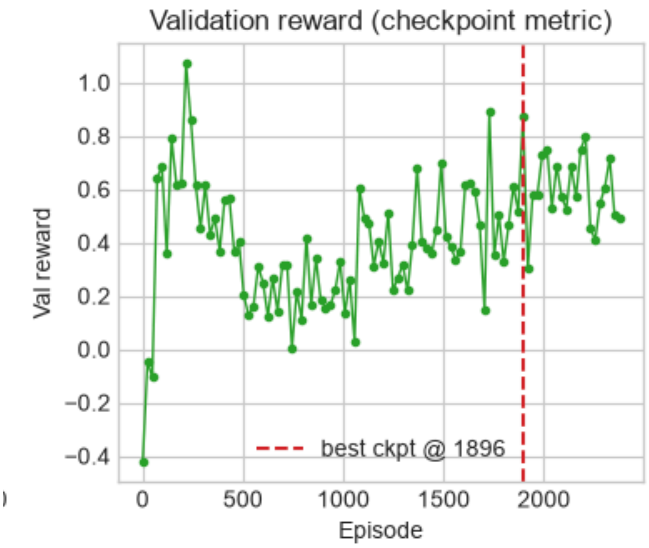


In the 2y slicing window, which was proven to be the most effective, each episode does not walk the full train period; instead it samples a random 504-day (~2-year) slice inside the train window.

This reduces memorization of one chronological path and exposes the agent to many local market regimes. Within an episode, the agent acts, stores transitions, and calls train_step() after each environment step. After every episode, ε is multiplied by 0.9985 and floored at 0.01, so exploration fades gradually over training. This is consistent with the epsilon-decay in Phase II.

Checkpointing and evaluation.

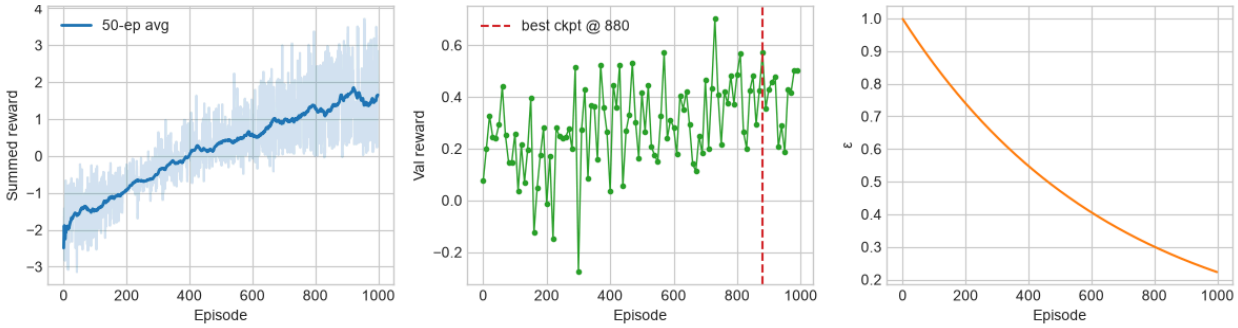
Every 20 episodes (1% of training), we run a greedy evaluate() pass on the validation environment and record summed validation reward. After the agent has completed 75% of training, we save the policy weights whenever validation reward improves. At the end of training we restore the best checkpoint before touching the test set. Final reporting uses run_policy, which compounds daily rewards into a portfolio multiple starting from $1 on train, val, and test windows, then compares those multiples to SPY, TLT, GLD, and SPY/TLT buy-and-hold baselines on the same dates.



Relative to tabular Q-learning, DQN's main advantage is that it can use the full continuous state without discretization, and experience replay plus a target network make learning more stable in noisy financial data. The tradeoff is less interpretability, heavier computation, and a larger risk of overfitting the train path, which is why validation checkpointing and held-out test evaluation at the 75% mark are essential parts of this phase.

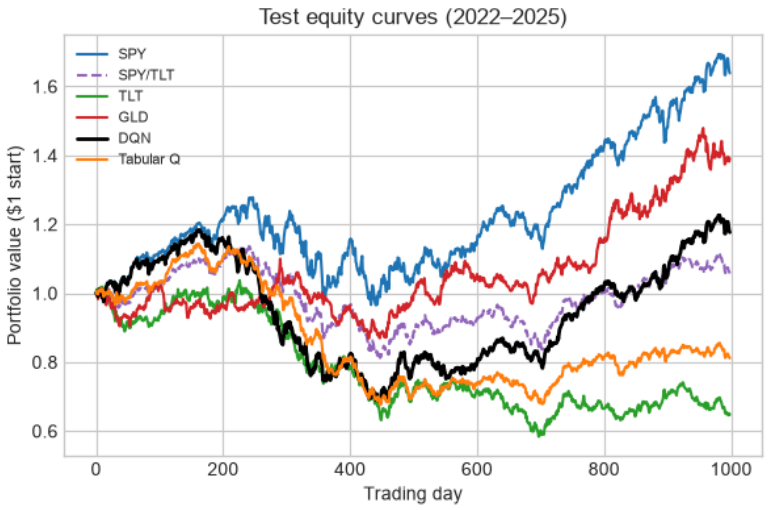
Phase IV: Experimentation and Results

I started the training process by treating it similar to the tabular Q-learning phase, where the agent looked at the entire test set during each episode. I initially trained the DQN model with 4 actions, a decay rate of 0.994, and 600 episodes. This produced subpar results as the model was unable to turn the initial dollar into any profit during the test period upon backtesting the learned policy. While the model is very efficient on the training period (135x portfolio value) it is simply overfitting and is unable to generalize on unseen data. This is a common issue with DQNs as they suffer from memorization and bootstrap error accumulation.



Because I wanted to keep the model from memorizing the chronological patterns of 2005 to 2016, which is the training period, I then had the model sample a random start time for each episode. I increased the number of episodes from 600 to 1000 to account for the model seeing less throughout each episode, around 50% on average. This produced slightly better results as the model was able to achieve a 3% profit on the test data.

I then took it a step further and turned each episode into a training window. In each iteration, the model will choose a random 2y window within the training period and learn from that. We increase the number of episodes here to allow the model to see each time period a consistent amount, which introduces the need to increase the epsilon decay so we hit an epsilon of around 0.01 at the end of the training process. Even with 4 actions, this produced much better results, indicating that the model was indeed memorizing the training time period.



Results from random 2y slicing

I then experimented with 6 and 10 actions, increasing the number of episodes to 2400 and 2800, respectively, for each so the model can learn in a higher dimension. I played around with a variety of parameters in the finetuning process, including α, γ, and ϵ decay. I also tried 1y windows with a smaller γ, but this didn't seem to improve the model's performance. Below are the results of the experimentation process, each done with a seed value of 42 to control the randomness.

ID	Method	Actions	Episodes	ε decay	Episode sampling
Q-1	Tabular Q	4	1,000	0.995	Full train window
DQN-1	DQN	4	600	0.994	Full train window
DQN-2	DQN	4	1000	0.994	Random start
DQN-3	DQN	4	2,000	0.998	2y random slices
DQN-4	DQN	6	2,400	0.9985	2y random slices
DQN-5	DQN	10	2,800	0.99885	2y random slices

Table 1. Experiment ID and attributes

Creating a portfolio with higher value than the buy-and-hold baselines turned out to be a challenge. Most of the agents underperformed on the testing set, evident in this log:

ID	Learned	SPY	SPY/TLT	TLT	GLD	vs SPY
Q-1	0.95×	1.64×	1.06×	0.65×	1.39×	-42%
DQN-1	0.98x	1.64x	1.06x	0.65x	1.39x	-40%
DQN-2	1.03x	1.64x	1.06x	0.65x	1.39x	-37%
DQN-3	0.90×	1.64×	1.06×	0.65×	1.39×	-45%
DQN-4	0.88×	1.64×	1.06×	0.65×	1.39×	-46%
DQN-5	1.55×	1.64×	1.06×	0.65×	1.39×	-5%

Table 2. Results on the testing period (2022-2025)

Each final policy was evaluated greedily (no exploration) on train, validation, and test. We logged portfolio multiples next to the baselines and used equity curves and action histograms to check for overfitting and policy collapse. Across runs, large train multiples often failed to carry over to 2022–2025, which confirmed that out-of-sample test results, not training reward, should drive conclusions.

ID	Train (2005–2016)	Val (2017–2021)	Test (2022–2025)	Full (2005–2025)
Q-1	3.04×	—	0.95×	—
DQN-1	60.06x	2.28x	0.98x	135.55x
DQN-2	108.42x	1.64x	1.04x	187.07
DQN-3	98.89×	1.38×	0.90×	123.07×
DQN-4	23.97×	1.72×	0.88×	36.21×
DQN-5	39.91×	2.11×	**1.55×**	132.07×
SPY	2.38x	1.79x	1.64x	7.04x
SPY/TLT	2.51x	1.64x	1.06x	4.34x
TLT	2.06x	1.37x	0.65x	1.82x
GLD	2.68x	1.50x	1.39x	5.60x

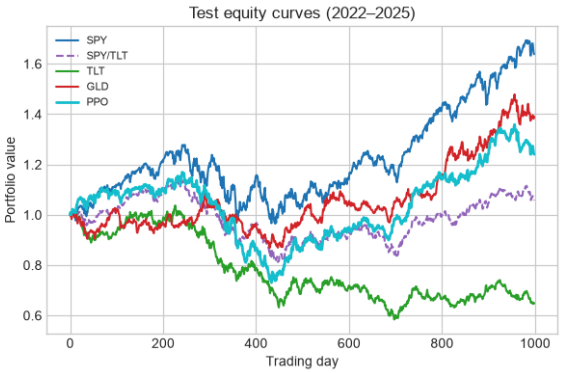
Table 3. Total experimentation process

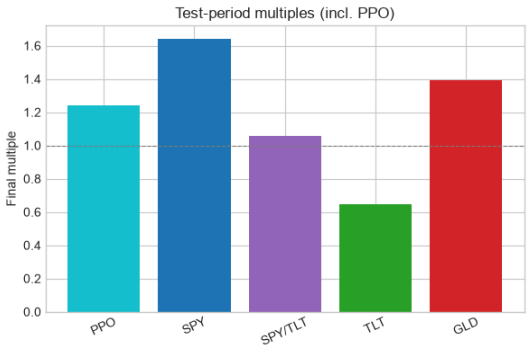
Results show that the model is able to memorize the training period very well, achieving over 2000% portfolio growth and even up to 10000%. This, however, is useless if the model cannot learn to outperform the SPY in unseen scenarios. The only model that got close to the SPY was the final model with 10 distinct actions and 2y training windows with a 50 day horizon, which outperformed all other buy-and-hold baselines and hit a 55% return. It is still worth noting that all trained DQN models were able to over 30x the portfolio value on the full time window, meaning the agent learned how to dynamically allocate funds based on market states. The key problem is generalization, which needs much more advanced methods. Future modifications could include increasing the selection of stocks or ETFs, which would inflate the dimensions of the environment but give the model more flexibility and potential for high-risk high-reward growth.

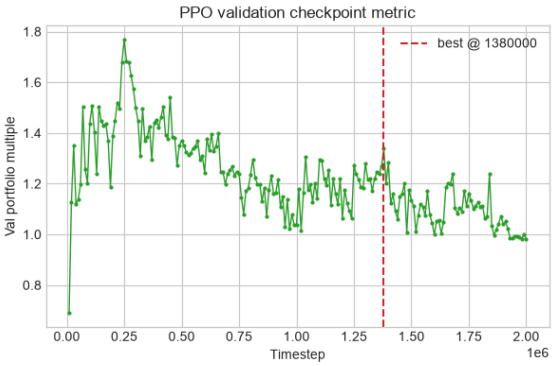
Phase V: Compare to SB3 PPO

As a third method I trained Proximal Policy Optimization (PPO) with Stable-Baselines3 on the same portfolio environment used by tabular Q-learning and DQN. PPO is an on-policy actor-critic algorithm: it samples trajectories from the current policy, estimates advantages with GAE, and updates a stochastic policy with a clipped objective so each update cannot move too far from the previous policy. I wrapped my portfolio environment class in a thin Gymnasium adapter, used the same 60/20/20 splits and train-only normalization as DQN, trained on random 504-day slices of 2005–2016, and selected checkpoints by validation portfolio multiple before evaluating on 2022–2025. Reporting used the same compounded $1 → final multiple as DQN's run policy, so the three methods are comparable on an equal footing.

Relative to the earlier methods, PPO plays a different role. Tabular Q-learning is simple and interpretable but limited by 27 discrete states; it underperformed buy-and-hold on test (~0.95× vs SPY 1.64×). DQN replaces the table with a neural Q-network, reuses experience through replay, and, with a richer 10-action space and longer training, produced our best out-of-sample result (1.55× on test, close to SPY and above GLD). PPO, despite being a strong general-purpose method, did not beat that DQN baseline on the held-out window. It tended to fit the training period more aggressively and generalize less well to 2022–2025, consistent with PPO's weaker sample efficiency when data are limited to one historical market path and the action space is small and discrete.







Overall, value-based DQN remained the strongest learned policy, while PPO illustrates that a more modern on-policy algorithm is not automatically better for this allocation task. The main shared lesson across all three methods is that in-sample portfolio multiples can look strong while test performance stays near or below simple SPY buy-and-hold, so validation checkpointing and held-out evaluation are essential.

Conclusion

This project analyzed daily portfolio allocation among SPY, TLT, and GLD as an approximate MDP. It compared three reinforcement learning approaches (tabular Q-learning, Deep Q-Networks, and PPO) against simple buy-and-hold strategies using data from 2005 to 2025. A shared environment, discrete allocation actions, and compounded portfolio multiples allowed for direct comparison of the methods. Train, validation, and test splits, along with validation checkpointing, helped avoid looking at the final out-of-sample data.

The main finding is that learning a policy to outperform the SPY buy-and-hold strategy on held-out data is tough. Tabular Q-learning served as a useful baseline but struggled as market conditions changed. DQN emerged as the best method: with a broader action space and careful training, it nearly matched SPY during 2022 to 2025, generating about 1.55 times returns compared to SPY's 1.64 times, and it outperformed several passive alternatives. PPO, a recent on-policy algorithm, did not improve on this result. It often overfit the training phase without carrying its success into the test years, indicating that newer methods aren't necessarily superior in this discrete, data-limited finance setting.

In all experiments, large returns within the sample were deceptive. Models that performed well from 2005 to 2016 frequently failed during 2022 to 2025, especially when bonds acted differently. Key takeaways include that engineered states only approximate a true Markov market. Off-policy learning with experience replay worked better for this task than PPO. Also, thorough out-of-sample evaluation is more important than studying training curves. Deep reinforcement learning can create effective allocation policies, but any real-world use should approach in-sample gains with skepticism and consider simple index buy-and-hold as a tough benchmark to surpass.