DQN Trading Agent Report

Project description and methodology:

The Deep Reinforcement Learning Trading Agent project is focused on the development of the Deep Q-Network (DQN) which is designed to autonomously trade the s&p500 or any other stock. The objective was to apply Deep Reinforcement Learning to trading by giving the agent many different technical analysis tools. The agent interacts with the environment we built to learn the most optimal trading strategies and gives the agent the opportunity to Buy, Sell, or Hold. This is all based on the technical analysis and the different market states. The system is built by using PyTorch for the neural networks as well as a custom OpenAi Gym environment, and the DQN agent. The model uses historical data from finance from 2025 to 2024, as well as other economic factors such as volatility index, the 10year treasure tiles and other technical indicators such as RSI, MACD, Bollinger Bands, and SMA. The agent utilizes a feedforward neural network with 2 hidden layers of 128 neurons each which uses ReLU activation. To ensure stable learning we also used a Replay Buffer to store and sample past experiences and we also used a target network that updates periodically to stabilize the Q-value targets.

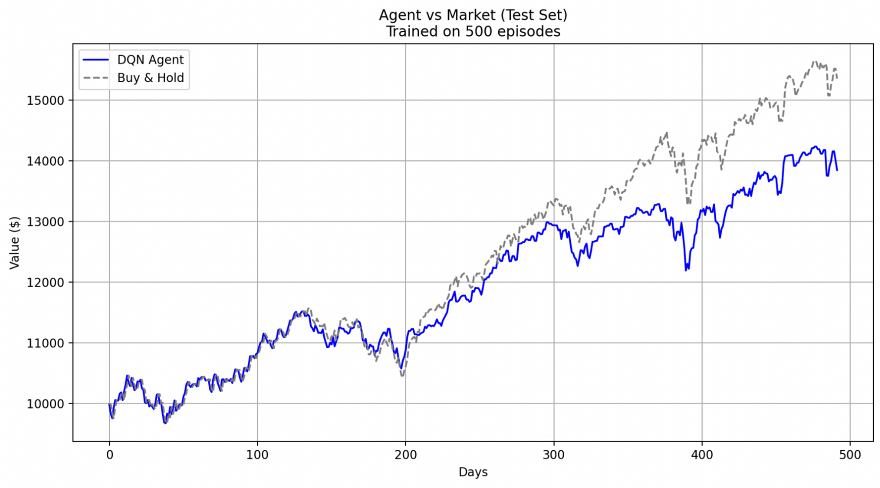
Steps taken and challenges encountered:

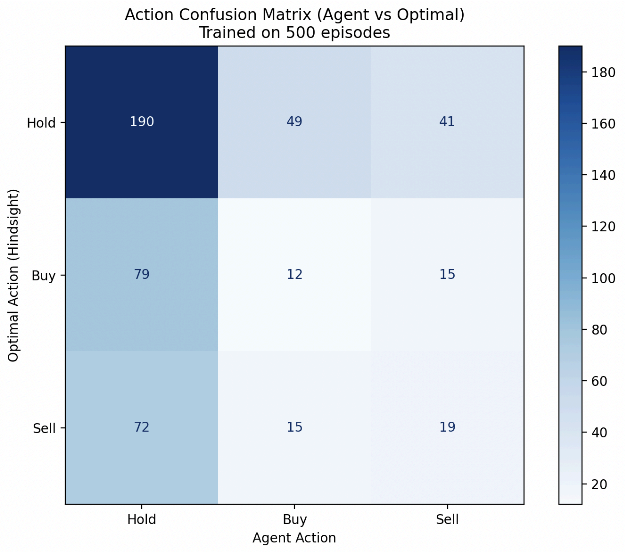
When developing, the progress was relatively linear. First we merged the separate datasets and features using Standard Scalar to ensure that the neural network received all its inputs within the 0-1 range (to prevent outlier ranges to stick out). Afterwards, we implemented the logic for calculating the portfolio values, transaction costs and setting the rewards for the agent. Then, we coded the training loop using epsilon greedy exploration and gradient clipping for stability. Lastly, I created the evaluation script to calculate the different evaluation metrics such as Sharpe ratio, cumulative return, alpha, beta etc… There were a couple of challenges I ran into. The first one was about invalid actions.The problem was that the agent would attempt to “BUY” when it did not have any cash and “SELL” when the agent had no shares. In order to resolve this in the training loop and evaluate the code check the position. If the agent is not holding a position the Q-value for “SELL” is set to negative infinity which forces the agent to either hold or buy. Another problem I ran into was how I can reward and scale the agent. A raw return of 1% is a small number for a neural network to learn from effectively. So I changed the reward function to multiply the percentage by 100, which makes the learning signal stronger. Another challenge I ran into was finding a sweet spot between overfitting and underfitting the market. To solve this the reset function ensures the agent learns actual recognizable patterns instead of memorizing the dates (sets a valid random start).

Results and Analysis:

To evaluate the stability and learning of the DQN, I used several different training runs all with different numbers of episodes. The behavior and results of the agent changes significantly as the exposure to the environment increases. I started off with smaller training sessions at 50 episodes, the agent's behavior wasn't the best. In this attempt the epsilon decay had not fully decayed as the agent was exploring too much at the end which leads to higher than wanted random actions. In terms of performance the agent underperformed the benchmark (buy and hold). With 500 episodes the agent started to demonstrate better behaviors, the agent started to learn that holding positions are really good during strong uptrends, instead of taking short term profits which seems more like the goldilocks zone as the 1000 eps training performed significantly worse. When trained with 100 and 250 episodes the agent was underfitted, the model did a lot of random things and performed low. In 100eps the agent likely overoptimized the training data, so instead of learning actual market trends, it memorized specific patterns that were in training and not testing. The model tested on the unseen 20% of data, the provided confusion matrix shows what the agent chose vs what it should have done. The “Hold” is the one that stood the most out, as the agent heavily biased towards it, as well as the “should have done” benchmark recommended to Hold nearly 70% of the time. The agent had many misclassifications (271) which was about 55% of the time. The Market’s Buy and Hold had a very strong bull run throughout those 10 years. The implementation of all the actions were 100% valid as we implemented Action Masking which eliminated any of the technical errors.

Metric	**100 eps**	**250 eps**	**500 eps**	**1000 eps**
Agent Return (%)	8.85%	8.25%	38.46%	0.85%
Buy & Hold Return (%)	51.77%	51.77%	51.77%	51.77%
Final Value ($)	$10,885.20	$10,825.23	$13,845.77	$10,085.03
Max Drawdown (%)	-1.70%	-2.33%	-8.24%	-5.51%
Sharpe Ratio (Agent)	1.205	1.084	1.442	0.115
Sharpe Ratio (Benchmark)	1.793	1.793	1.793	1.793
Total Trades	12	48	151	112





	Hold	Buy	Sell
Hold	190	49	41
Buy	79	12	15
Sell	72	15	19

Effort Justification:
The project meets the criteria for 1.5x effort when compared to a normal project, because of all the integration and learning that must've been done in order to understand and build a complex system. In normal assignments we start off with a skeleton code, with some helpful explanations during class and other content on elc. I built a custom trading environment that simulates what a realistic brokerage account goes through including cash balancing, share tracking, transaction costs. While HW11 we used a simple q-table to help me robot navigate through a simple maze, this project required implementing a full deep learning using neural networks. This included building the different states, actions, tuning and playing around with the hyperparameters, implementing a replay buffer and target network to stabilize the model, and having long train times. There were a couple main skills that I learned, starting off with getting much needed experience with PyTorch, as this is the first time I've used a deep learning framework. I also had to learn deep reinforcement theory and architecture, I learned how to stabilize the model using experience replay, I also learned that we needed a target network that helps stabilize the model to allow convergence. I then learned to calculate some complex financial metrics with the ta library. I learned to architect a custom reinforcement learning environment. I had to design the different states and perform action masking to only allow valid trades. Since this was a solo project I had to develop and learn everything myself from data processing to neural network architecture.

Rubric Checklist:

Criteria	**Points**	**Description**	**How I fulfilled criteria**
DQN Implementation & Code quality	30	Neural Network, experience play buffer, clean PyTorch Code	Neural Network was implemented in the DQN class using PyTorch's torch.nn. Experience replay was created in ReplayBuffer class to store and sample the transitions, breaking time series correlation Utilized a separate target network with periodic updates to stabilize training
Trading Environment Design	5	Realistic constraints such as Transaction costs, position sizing	The TrainingEnv class simulates a brokerage account, tracking cash, shares, and a portfolio value separately. Transaction costs were implemented at 0.1%.
Learning	10	Evidence of Learning	There was demonstrated learning from 100 episodes to 500 episodes
Visualizations, Insights and Comparative Analysis	25	Trading behavior visualization, action distributions, performance across market conditions, clear plots	Generated a confusion matrix to analyze the agent's decision making against an optimal strategy. I also created Plots comparing benchmark to agent's portfolio. I calculated various metrics to have a deeper insight.
Total	70		Completed all the requirements