Certainly, monitoring the performance and training progress of individual agents is crucial. Here are several metrics and visualizations you can consider:

1. **Drawdown**: This measures the largest drop from a peak in the portfolio value. It can provide insight into the potential risks an agent is taking.
2. **Position Duration**: How long does the agent typically hold a position? This can help you understand if your agent is behaving more like a high-frequency trader, a day trader, or a longer-term trader.
3. **Total Profit vs. Total Loss**: Separately summing up positive and negative rewards can give insights into the balance between profitable and non-profitable actions.
4. **Action Distribution**: A histogram or pie chart of buy, sell, and hold actions. This can show if the agent has a bias towards a particular action.
5. **Trade Win/Loss Ratio**: The ratio of profitable trades to unprofitable ones.
6. **Average Profit per Winning Trade vs. Average Loss per Losing Trade**: This can help you understand the risk-reward profile of the agent.
7. **Portfolio Value Over Time**: Plotting the agent's portfolio value over time can give insights into its consistency and potential drawdowns.
8. **Consecutive Losses**: The number of consecutive losing trades. If this number grows, it might be a sign that the agent is in a "rut" or that the market regime has changed.
9. **Value at Risk (VaR)**: This is a measure of the risk of investments. It estimates how much a set of investments might lose, given normal market conditions.
10. **Learning Curves**: Plot the mean reward and loss over episodes. This can help in understanding if the agent is still learning or has converged.
11. **Q-Value or Critic Loss (for algorithms like DDPG, A2C)**: Monitoring the learning process of the Q-value can provide insights into the stability of the learning process.
12. **Entropy of the Policy (for policy gradient methods)**: This can show how deterministic or exploratory the policy is. A decreasing entropy might show that the agent is becoming too deterministic, which might not always be a good thing.
13. **Exploration Rate (for methods with epsilon-greedy exploration)**: How much the agent is exploring versus exploiting.
14. **Parameter Updates**: If you're using adaptive learning rates, it can be useful to monitor how often and by how much parameters (like learning rates) are being updated.
15. **Correlation with Market**: If the agent's returns are highly correlated with the broader market, then it might not be adding as much value as one that has a lower correlation.

Remember, the key is to not just look at one metric in isolation but to consider a combination to get a holistic view of the agent's performance and behavior.

The warnings you're seeing are from NumPy when it encounters operations that result in NaN (Not a Number). This can happen, for example, when you try to compute the mean of an empty array or when you divide by zero.

To figure out where this happened in your environment, you can:

1. **Add Logging**: Add print statements or logging in your environment where you suspect operations related to these warnings might be happening. This might be places where you compute the mean, divide numbers, or compute other statistics.
2. **Check the Stacks**: The warnings give you a clue about where they are raised (**fromnumeric.py**, **\_methods.py**). However, these are internal NumPy files. What you'd ideally want is to see the stack trace leading to your environment's code. You can modify your environment to catch the **RuntimeWarning** and print a full stack trace. Here's an example:

pythonCopy code

import warnings import traceback with warnings.catch\_warnings(): warnings.filterwarnings('error') try: # Your suspected code here, e.g., np.mean(empty\_array) except RuntimeWarning: traceback.print\_exc()

This will convert the warning into an error, catch it, and then print the full traceback, which should include the location in your environment that caused the warning.

1. **Inspect Values**: In parts of your environment code where you compute statistics, like the mean or the Sortino ratio (as the logs suggest), inspect the values you're computing with. Ensure that arrays aren't empty and denominators aren't zero before performing operations.
2. **Environment Reset and Step**: If the above steps don't give you a direct answer, you might want to closely inspect the **reset** and **step** methods of your environment, as these are the most likely places where such computations might be happening.

Lastly, while these are warnings and might not halt your program, they indicate potential issues in your computations, so it's a good idea to address them.

Current best trial: d64a6\_00003 with episode\_reward\_mean=74844.76906045801 and parameters={'extra\_python\_environs\_for\_driver': {}, 'extra\_python\_environs\_for\_worker': {}, 'num\_gpus': 0, 'num\_cpus\_per\_worker': 1, 'num\_gpus\_per\_worker': 0, '\_fake\_gpus': False, 'num\_learner\_workers': 0, 'num\_gpus\_per\_learner\_worker': 0, 'num\_cpus\_per\_learner\_worker': 1, 'local\_gpu\_idx': 0, 'custom\_resources\_per\_worker': {}, 'placement\_strategy': 'PACK',