Teja Sree Parasa October 27th, 2024

CSE 584: Machine Learning Homework 2

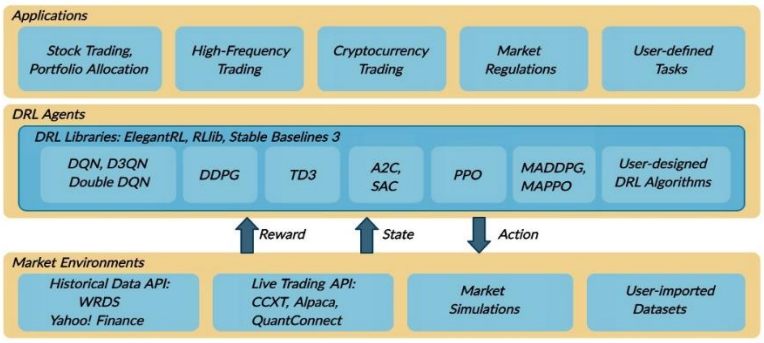
Reinforcement Learning

AI4Finance Foundation FinRL

Source: https://github.com/AI4Finance-Foundation/FinRL

Question 1 - About the Project and its Workflow:

This was a Deep Reinforcement Learning framework from GitHub, applied to financial market applications. It is designed to develop autonomous trading agents that make data-driven decisions in dynamic financial environments. Organization of code is subdivided into three layers: Applications, DRL Agents, and Market Environments-each serving a different purpose that together allows functionality for this framework (Represented in the figure below).



The Applications layer defines real-world financial tasks that the agents are trained to tackle, such as stock trading, high-frequency trading, cryptocurrency trading, and portfolio allocation. It is also flexible enough to adapt to everything from user-defined tasks to compliance with market regulations for various financial use cases. Each of the applications mentioned in the above figure represents a situation where an agent has to perform complex decisions based on market data, representing the challenges that face agents in live trading environments.

The DRL Agents layer hosts the core reinforcement learning algorithms that enable the agents to learn optimal strategies for different trading tasks. There is a variety of integrated DRL libraries with foundation implementations of popular DRL algorithms in the code. It supports algorithms such as DQN, PPO, SAC, TD3, MADDPG, and multi-agent setups such as MAPPO. In other words, one can try both single-agent and collaborative trading strategies. Agents are trained via observing the state of the market and choosing an appropriate action-for instance, buying, holding, or selling assets-with respect to their policy and receiving a feedback about that as some reward. In this case, it's a feedback loop where agents iterate over their decision policies in search of maximum returns by adapting to changes in markets.

The Market Environments layer provides all the data sources and simulation environments that are necessary for training and testing these agents. The historical data APIs, such as WRDS and Yahoo! Finance, provide market data agents utilize in training to simulate real trading without exposing them to financial risks. The execution of live trading APIs for CCXT, Alpaca, and QuantConnect also opens the doors for real-time trading with those agents that are adequately trained and ready for deployment. The environment layer also includes market simulations and user-imported data sets to let the agents generalize their learning to a variety of market conditions. With these resources at hand, agents can be extensively trained and tested before making their way into live markets.

The process of reinforcement learning here considers the code to be a sequence of interactions: the agent perceives the current state of the market environment, executes an action, and receives a reward for such an outcome. For example, in this core implementation, functions are responsible for basically observing the state, choosing actions, and calculating rewards. Comments in these functions are extensive, explaining each step-from Q-value computation to the update of the policies via gradient descent-thus being illustrative for how exactly the agents will learn and optimize their trading behavior. In this manner, through iterations of actions and feedback, the agents build their ability to maximize cumulative rewards and refine strategies that could realize consistent profitability in volatile markets.

Reinforcement Learning Aspect of the Project:

This codebase, when viewed from the point of view of reinforcement learning implementation, is quite well-structured and modular in handling such a complex task as training agents in dynamic environments. The RL part was designed around a series of training loops where agent-environment interactions were managed, enabling the use of popular reinforcement learning algorithms like Deep Q-Networks, Proximal Policy Optimization, and direct Policy Gradient methods. Each of the various algorithms is packaged in its own module; this makes the process of experimenting with different algorithms - and indeed switching between them - very easy. The core of the iterative refinement of the agent's policy through interaction with the environment lies in the RL implementation's training loop. At each time step, the agent gets an observation of the current state of the environment, courtesy of the underlying ODE description. This state is then fed into the agent's neural network, which predicts an action according to the current policy. The environment will then react to this actionInternal by updating its internal state with the ODEs and returning a reward signal that quantifies effectively how good this action was. This reward, in combination with the new state, is again used to update the policy of the agent. The loop continues until a maximum number of episodes is reached or a specific performance threshold is achieved.

From the point of view of algorithm-specific details, each RL algorithm takes a little bit of a different approach towards policy updates. As an example, the DQN implementation should contain code with an experience replay buffer, storing past experiences used for training by means of random sampling. This mitigates the issue of correlated data and results in more stable learning. The DQN module also makes use of a target network, updated periodically, to further stabilize learning by providing one fixed point toward which the updates in Q-values are projected.

The PPO implementation is an actor-critic framework where two neural networks-the actor and the critic-are both being trained in parallel. The actor network is updated to maximize the expected cumulative reward, while the critic network learns to predict the value function, which estimates future rewards from a given state. The PPO's clipped objective function implemented in the code prevents excessively large policy updates, creating stability during training and avoiding any destructive updates that can result in the derivation of suboptimal policies.

Each of the algorithm modules exposes tunable hyperparameters that have to be set for the fine-tuning of the agent. The code also supports adaptive learning rates and exploration decay schedules, hence respective adjustments could be done during training to obtain good convergence. That is certainly useful in ODE-driven dynamics when the state space might change unpredictably with time.

Observe the logging and evaluation functions that form part of the RL code: The code will log performance metrics-cumulative reward, episode length, and policy loss-at each train episode. These metrics are important to diagnose the progress of training and determine when an agent has converged to an optimal or near-optimal policy. This is done periodically in the environment without exploration, only taking deterministic actions of the agent to evaluate the policy under optimum conditions. It basically sets up an evaluation function that helps in realizing whether the

policy adopted by the agent is appropriate and how well the agent will work in cases where fine-tuning might be needed.

In summary, this repository is a good example of how to build an RL code that is comprehensive and robust, clear in its separation between algorithmic logic, agent configuration, and environment interactions. Its implementation provides efficient ways of training and evaluating the agents; it is highly adaptable for a wide range of RL tasks, especially those with environments modeled by differential equations. This modular approach allows fast turnarounds of algorithms and hyperparameters, therefore fostering experimentation and optimization that caters to the peculiar demands of complex continuous-state environments.

Question 2 - Code Implementation: Commenting and Understanding (https://github.com/AI4Finance-Foundation/FinRL/blob/master/finrl/agents/stablebaselines3/models.py)

```
def _train_window(
    self,
    model_name,
    model_kwargs,
    sharpe_list,
    validation_start_date,
    validation_end_date,
    timesteps_dict,
    i,
    validation,
    turbulence_threshold,
):
    """
    Train the model for a single time window.
    """
    # Check if model kwargs are provided, if not, stop the function
    if model_kwargs is None:
        return None, sharpe_list, -1  # No model trained, return empty values

    # Print the model being trained
    print(f"======{model_name} Training========")

    # Initialize the model with training environment and settings
    model = self.get_model(
        model_name, self.train_env, policy="MlpPolicy", model_kwargs=model_kwargs
    )

    # Train the initialized model with specified parameters like timesteps and log names
    model = self.train_model(
        model,
        model_name,
        tb_log_name=f"{model_name}_{i}",
```

```
    iter_num=i,
    total_timesteps=timesteps_dict[model_name],
)

# Print the validation date range for tracking
print(
    f"======{model_name} Validation from: ",
    validation_start_date,
    "to ",
    validation_end_date,
)

# Create a new validation environment
val_env = DummyVecEnv(
    [
        lambda: StockTradingEnv(
            df=validation,
            stock_dim=self.stock_dim,
            hmax=self.hmax,
            initial_amount=self.initial_amount,
            num_stock_shares=[0] * self.stock_dim,
            buy_cost_pct=[self.buy_cost_pct] * self.stock_dim,
            sell_cost_pct=[self.sell_cost_pct] * self.stock_dim,
            reward_scaling=self.reward_scaling,
            state_space=self.state_space,
            action_space=self.action_space,
            tech_indicator_list=self.tech_indicator_list,
            turbulence_threshold=turbulence_threshold,
            iteration=i,
            model_name=model_name,
            mode="validation",
            print_verbosity=self.print_verbosity,
        )
    ]
)

# Reset the environment to get the initial state for validation
val_obs = val_env.reset()

# Run validation on the model to get the results
self.DRL_validation(
    model=model,
    test_data=validation,
    test_env=val_env,
```

```
            test_obs=val_obs,
        )

        # Calculate and print the Sharpe ratio for the model in this validation window
        sharpe = self.get_validation_sharpe(i, model_name=model_name)
        print(f"{model_name} Sharpe Ratio: ", sharpe)

        # Store the Sharpe ratio in the sharpe list
        sharpe_list.append(sharpe)

        # Return the model, updated sharpe list, and the latest Sharpe ratio
        return model, sharpe_list, sharpe


    def run_ensemble_strategy(
        self,
        A2C_model_kwargs,
        PPO_model_kwargs,
        DDPG_model_kwargs,
        SAC_model_kwargs,
        TD3_model_kwargs,
        timesteps_dict,
    ):
        """
        Ensemble strategy combining A2C, PPO, DDPG, SAC, and TD3 models.
        """
        # Dictionary holding all model kwargs
        kwargs = {
            "a2c": A2C_model_kwargs,
            "ppo": PPO_model_kwargs,
            "ddpg": DDPG_model_kwargs,
            "sac": SAC_model_kwargs,
            "td3": TD3_model_kwargs,
        }

        # Dictionary storing Sharpe ratios for each model
        model_dct = {k: {"sharpe_list": [], "sharpe": -1} for k in MODELS.keys()}

        # Track starting message for the ensemble strategy
        print("============Start Ensemble Strategy============")

        # Initialize empty list to store last state for the ensemble
        last_state_ensemble = []
```

```
# Track models used and validation dates
model_use = []
validation_start_date_list = []
validation_end_date_list = []
iteration_list = []

# Get the 90th quantile of the turbulence data during the in-sample period
insample_turbulence = self.df[
   (self.df.date < self.train_period[1])
   & (self.df.date >= self.train_period[0])
]
insample_turbulence_threshold = np.quantile(
   insample_turbulence.turbulence.values, 0.90
)

start = time.time()  # Track time for process duration

# Loop through all trade dates with specified rebalance window
for i in range(
   self.rebalance_window + self.validation_window,
   len(self.unique_trade_date),
   self.rebalance_window,
):
   # Set validation start and end dates
   validation_start_date = self.unique_trade_date[
      i - self.rebalance_window - self.validation_window
   ]
   validation_end_date = self.unique_trade_date[i - self.rebalance_window]

   # Store validation dates in lists
   validation_start_date_list.append(validation_start_date)
   validation_end_date_list.append(validation_end_date)
   iteration_list.append(i)

   # Set initial flag for the very first window
   initial = i - self.rebalance_window - self.validation_window == 0

   # Calculate turbulence threshold using historical turbulence values
   end_date_index = self.df.index[
      self.df["date"]
      == self.unique_trade_date[
         i - self.rebalance_window - self.validation_window
      ]
   ].to_list()[-1]
```

```
start_date_index = end_date_index - 63 + 1
historical_turbulence = self.df.iloc[
   start_date_index : (end_date_index + 1), :
]
historical_turbulence = historical_turbulence.drop_duplicates(
   subset=["date"]
)
historical_turbulence_mean = np.mean(
   historical_turbulence.turbulence.values
)

# Set turbulence threshold based on in-sample turbulence threshold
turbulence_threshold = (
   insample_turbulence_threshold
   if historical_turbulence_mean > insample_turbulence_threshold
   else np.quantile(insample_turbulence.turbulence.values, 1)
)
turbulence_threshold = np.quantile(
   insample_turbulence.turbulence.values, 0.99
)
print("turbulence_threshold: ", turbulence_threshold)

# Environment setup: create train and validation data splits
train = data_split(
   self.df,
   start=self.train_period[0],
   end=self.unique_trade_date[
      i - self.rebalance_window - self.validation_window
   ],
)
self.train_env = DummyVecEnv(
   [
      lambda: StockTradingEnv(
         df=train,
         stock_dim=self.stock_dim,
         hmax=self.hmax,
         initial_amount=self.initial_amount,
         num_stock_shares=[0] * self.stock_dim,
         buy_cost_pct=[self.buy_cost_pct] * self.stock_dim,
         sell_cost_pct=[self.sell_cost_pct] * self.stock_dim,
         reward_scaling=self.reward_scaling,
         state_space=self.state_space,
         action_space=self.action_space,
         tech_indicator_list=self.tech_indicator_list,
```

```
            print_verbosity=self.print_verbosity,
        )
    ]
)

validation = data_split(
    self.df,
    start=self.unique_trade_date[
        i - self.rebalance_window - self.validation_window
    ],
    end=self.unique_trade_date[i - self.rebalance_window],
)

print(
    "======Model training from: ",
    self.train_period[0],
    "to ",
    self.unique_trade_date[
        i - self.rebalance_window - self.validation_window
    ],
)

# Train each model and validate its performance
for model_name in MODELS.keys():
    model, sharpe_list, sharpe = self._train_window(
        model_name,
        kwargs[model_name],
        model_dct[model_name]["sharpe_list"],
        validation_start_date,
        validation_end_date,
        timesteps_dict,
        i,
        validation,
        turbulence_threshold,
    )
    model_dct[model_name]["sharpe_list"] = sharpe_list
    model_dct[model_name]["model"] = model
    model_dct[model_name]["sharpe"] = sharpe

# Select the model with the highest Sharpe ratio
sharpes = [model_dct[k]["sharpe"] for k in MODELS.keys()]
max_mod = list(MODELS.keys())[np.argmax(sharpes)]
model_use.append(max_mod.upper())
model_ensemble = model_dct[max_mod]["model"]
```

```
    # Trading period after retraining the best model
    last_state_ensemble = self.DRL_prediction(
        model=model_ensemble,
        name="ensemble",
        last_state=last_state_ensemble,
        iter_num=i,
        turbulence_threshold=turbulence_threshold,
        initial=initial,
    )

end = time.time()  # End timing
print("Ensemble Strategy took: ", (end - start) / 60, " minutes")

# Summary DataFrame of results with Sharpe ratios and selected models
df_summary = pd.DataFrame(
    [
        iteration_list,
        validation_start_date_list,
        validation_end_date_list,
        model_use,
        model_dct["a2c"]["sharpe_list"],
        model_dct["ppo"]["sharpe_list"],
        model_dct["ddpg"]["sharpe_list"],
        model_dct["td3"]["sharpe_list"],
        model_dct["sac"]["sharpe_list"],
    ]
).T
df_summary.columns = [
    "Iter",
    "Val Start",
    "Val End",
    "Ensemble Model",
    "A2C Sharpe",
    "PPO Sharpe",
    "DDPG Sharpe",
    "TD3 Sharpe",
    "SAC Sharpe",
]

return df_summary
```