Insight: AI Stock Prediction

By: Naif Alzmami, Mohammed Alharbi, Naif Alqhatani

Abstract

This report explores the application of Deep Reinforcement Learning (RL), specifically the Deep Deterministic Policy Gradient (DDPG) algorithm, in predicting and optimizing trading decisions in the Saudi stock market. The goal is to assess the model's ability to outperform traditional strategies like buy-and-hold. The experiments reveal varied outcomes: in the best scenarios, the model mimicked the buy-and-hold strategy, achieving similar returns; in good scenarios, it managed to make profitable trades, although still close to buy-and-hold performance; in bad scenarios, it underperformed, highlighting the model's volatility; and in some cases, it failed to learn any effective trading strategy, resulting in poor performance. These results indicate that while DDPG shows promise, its current form is unstable and inconsistent, requiring further research to enhance its reliability and effectiveness in real-world trading environments.

Report on "Insight: AI Stock Prediction" using Deep Reinforcement Learning (RL) and DDPG

Background

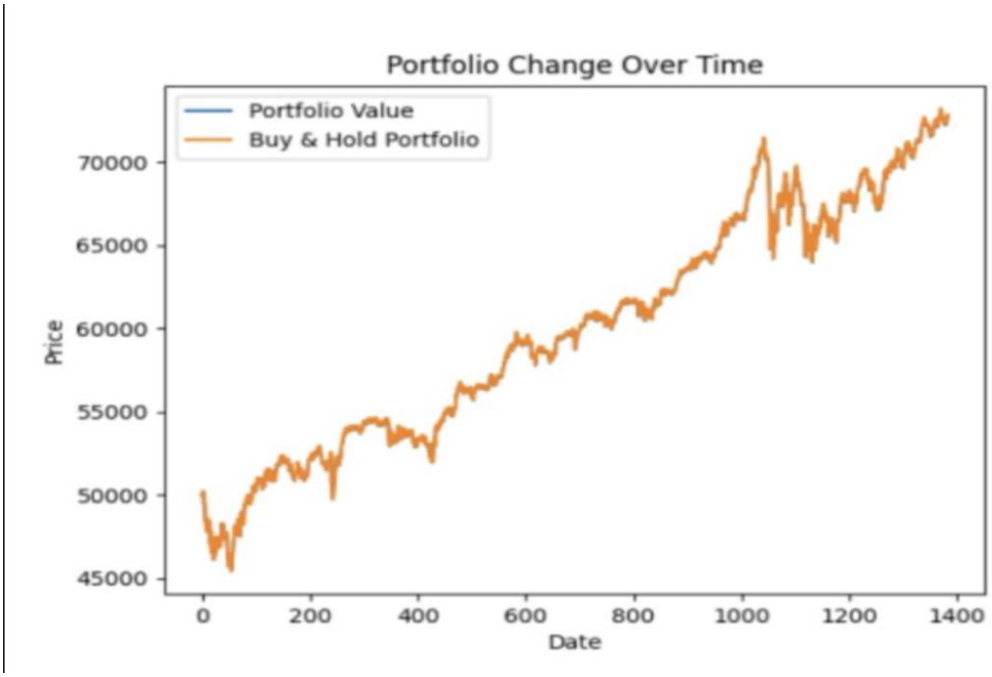
Artificial Intelligence (AI) has made significant strides in various industries, with its application in financial markets, particularly stock trading, attracting considerable interest. Traditional stock trading methods often depend on historical data and human judgment, which can be prone to errors and biases. Reinforcement Learning (RL), especially through the Deep Deterministic Policy Gradient (DDPG) algorithm, offers an advanced approach where an agent learns optimal trading strategies by interacting with the market environment. This report investigates the feasibility and effectiveness of using DDPG for automated stock trading in the Saudi stock market, comparing its performance against the widely used buy-and-hold strategy.

Method

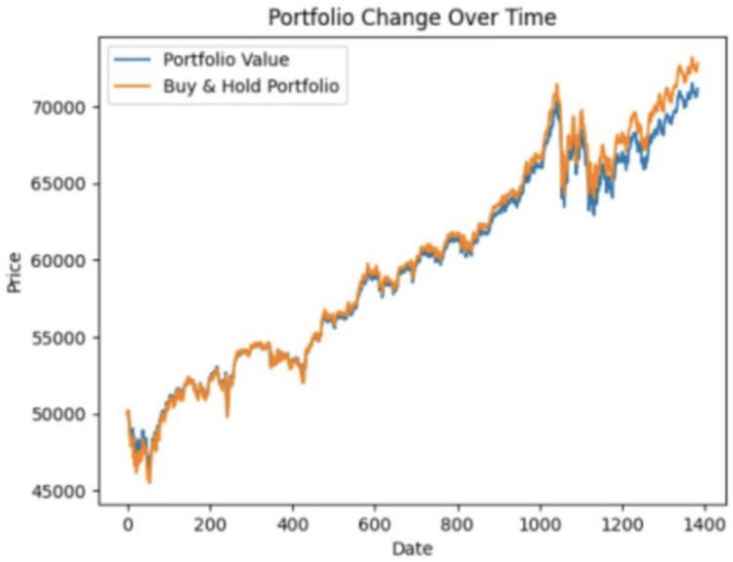
1.**Data Collection**: The model was trained on historical stock prices and financial data from the Saudi stock market, capturing essential features necessary for making informed trading decisions.
2.**Data Preprocessing**: The collected data was normalized and structured to serve as inputs for the RL model, ensuring that the model could learn effectively from the available information.
3.**Model Development**: The DDPG algorithm was implemented to create a trading agent. The agent was designed to maximize cumulative rewards by selecting the best possible actions (buy, sell, hold) based on the current state of the market.
4.**Testing and Evaluation**: The model's performance was evaluated in different scenarios, focusing on its ability to outperform the buy-and-hold strategy, which served as a baseline.
Results

The outcomes of the experiments with the DDPG algorithm varied significantly:

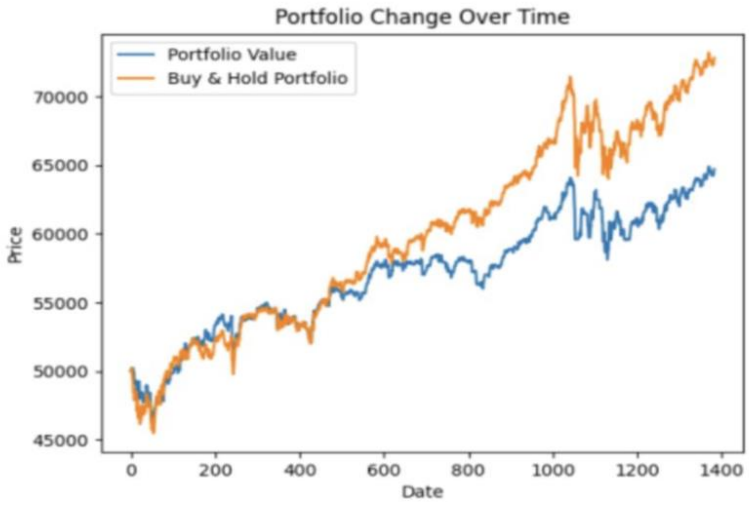
•**Best Result (Mimicked Buy-and-Hold Strategy)**: In the best-case scenario, the model's performance closely mirrored the buy-and-hold strategy, achieving similar returns. This result suggests that while the model could match the baseline strategy, it did not offer any substantial improvement in terms of generating higher returns.



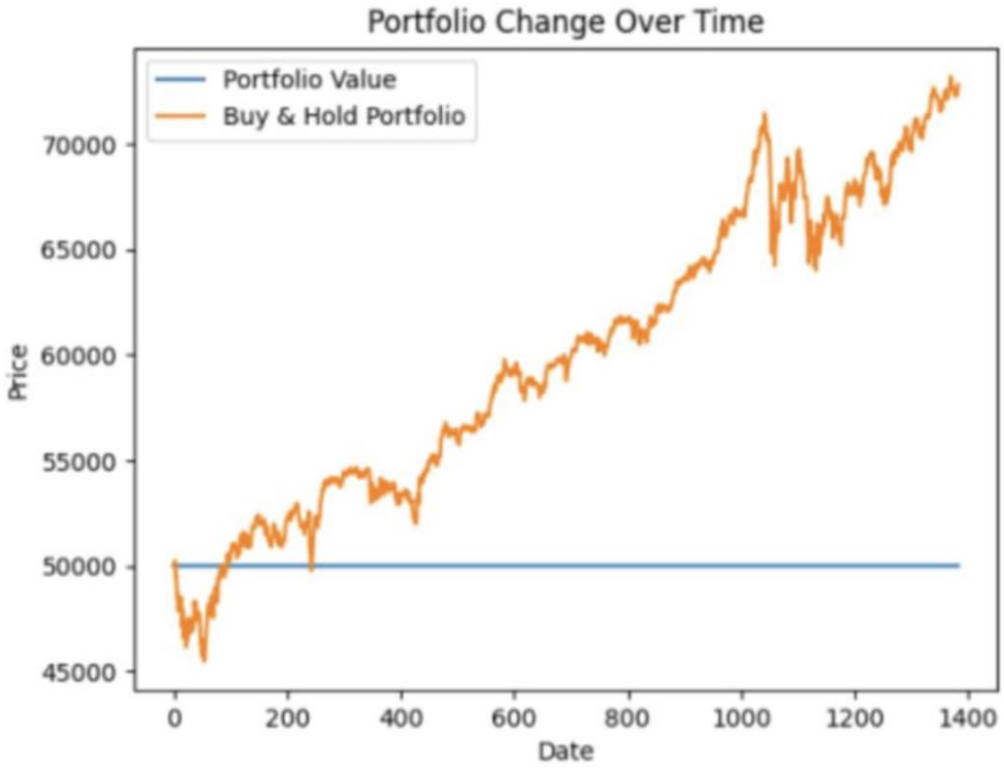
•**Good Result (Profitable but Similar to Buy-and-Hold)**: The model managed to execute trades that led to profits in some cases. However, these profits were very close to what could have been achieved by simply following the buy-and-hold strategy. This indicates that while the model could make some informed trading decisions, it was not significantly better than a passive investment approach.



•**Bad Result (Underperformance Compared to Buy-and-Hold)**: In certain scenarios, the model underperformed compared to the buy-and-hold strategy, highlighting the volatility and unpredictability of the model. These outcomes demonstrate that the model sometimes made poor trading decisions, leading to lower returns.



•**No Learning (Failure to Develop a Useful Strategy)**: In some instances, the model failed to learn any effective trading strategy, resulting in poor performance. This suggests that the model struggled to identify patterns in the data that could be leveraged for profitable trading, leading to random or ineffective decisions.



Discussion

The varied outcomes of the experiments indicate that while the DDPG algorithm has potential, it faces significant challenges in reliably outperforming a simple buy-and-hold strategy. The model's instability and sensitivity to initial conditions and hyperparameters were major drawbacks. Furthermore, the lack of long-term memory in the DDPG algorithm limited its ability to capture temporal dependencies in stock price movements, which are crucial for making informed trading decisions.

Conclusion

The DDPG algorithm, in its current form, is not yet a reliable tool for automated stock trading when compared to traditional strategies like buy-and-hold. Although it shows promise, particularly when enhanced with technical indicators and more sophisticated state representations, the model's instability and inconsistent performance limit its practical application in real-world trading environments.

Future Work

To improve the reliability and performance of RL-based trading models, future research should focus on several key areas:

- **Incorporation of Memory Mechanisms**: Introducing models like GRU or LSTM that can capture long-term dependencies in stock price data may enhance the agent's decision-making abilities.
- **Algorithm Refinement**: Exploring advanced RL algorithms like TD3 or combining multiple RL strategies could yield better results.
- **Diverse Data Integration**: Expanding the model's state space to include a broader range of market indicators and using more extensive datasets could improve its generalizability and robustness.

In conclusion, while the application of Deep Reinforcement Learning to stock trading is a promising field, significant advancements are needed before these models can consistently outperform traditional strategies and be deployed in real-world trading scenarios.

```
import copy
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import utility.utils as utils
from math import floor

device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")

# Implementation of Twin Delayed Deep Deterministic Policy
Gradients (TD3)
# Paper: https://arxiv.org/abs/1802.09477
# Original Implementation found on https://github.com/sfujim/
TD3/blob/master/TD3.py
# Regular DDPG: https://github.com/ghliu/pytorch-ddpg
class FirstBlock(nn.Module):
    def __init__(self, in_channel, out_channel):
        super().__init__()
        self.conv = nn.Conv2d(in_channel, out_channel,
kernel_size=3, stride=1, padding=1, bias=False)
        torch.nn.init.kaiming_normal_(self.conv.weight,
mode='fan_in')
        self.bn = nn.BatchNorm2d(out_channel)
        self.prelu = nn.PReLU()

    def forward(self, x):
        out = self.conv(x)
        out = self.bn(out)
        out = self.prelu(out)
        return out


class InnerBlock(nn.Module):
    def __init__(self, in_channel, out_channel, stride=1):
        super().__init__()
        assert stride == 1 or stride == 2
        self.conv = nn.Conv2d(in_channel, out_channel,
kernel_size=3, stride=stride, padding=1)
        torch.nn.init.kaiming_normal_(self.conv.weight,
mode='fan_out')
        self.bn = nn.BatchNorm2d(out_channel)

        self.conv2 = nn.Conv2d(out_channel, out_channel,
```

```
kernel_size=3, stride=1, padding=1)
        torch.nn.init.kaiming_normal_(self.conv2.weight,
mode='fan_out')
        self.bn2 = nn.BatchNorm2d(out_channel)

        self.prelu = nn.PReLU()

        if stride == 1 and in_channel == out_channel:
            self.shortcut = nn.Identity()
        else:
            self.shortcut = nn.Conv2d(in_channel, out_channel,
kernel_size=1, stride=stride)
            torch.nn.init.kaiming_normal_(self.shortcut.weight,
mode='fan_out')


    def forward(self, x):
        out = self.conv(x)
        out = self.bn(out)
        out = self.prelu(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.prelu(self.shortcut(x) + out)
        return out


class CNN(nn.Module):
    def __init__(
        self,
        inchannel,
        outchannel,
        activation=nn.PReLU,
    ):
        super(CNN, self).__init__()
        self.layers = nn.Sequential(
            FirstBlock(inchannel, 32),

            InnerBlock(32, 64),
            InnerBlock(64, 64),

            InnerBlock(64, 128, stride=2),
            InnerBlock(128, 128),
            nn.Dropout(0.15),
            InnerBlock(128, 128),
```

```
            InnerBlock(128, 256, stride=2),
            InnerBlock(256, 256),
            nn.Dropout(0.15),
            InnerBlock(256, 256),

            InnerBlock(256, 512, stride=2),
            InnerBlock(512, 512),
            nn.Dropout(0.15),
            InnerBlock(512, 512),
            nn.AdaptiveAvgPool2d((1, 1))
        )

        self.flatten = nn.Flatten()
        self.output = nn.Linear(512,  outchannel)


    def forward(self, X, X_immediate=None):
        out = self.layers(X)
        out = self.flatten(out)
        if X_immediate != None:
            out2 = self.layers(X_immediate.unsqueeze(1))
            out2 = self.flatten(out2)
            out = self.output(torch.cat((out, out2), -1))
        return out

class Actor(nn.Module):
    def __init__(self, state_dim, action_dim, max_action):
        super(Actor, self).__init__()
        self.conv = CNN(state_dim, 600)
        self.l1 = nn.Linear(600, 400)
        self.l2 = nn.Linear(400, 300)
        self.l3 = nn.Linear(300, action_dim)

        self.prelu = nn.PReLU()
        self.max_action = max_action
        self.init_weights()

    def init_weights(self):
        layers = [self.l1, self.l2, self.l3]
        for layer in layers:
            torch.nn.init.kaiming_uniform_(layer.weight)

    def forward(self, state):
        state = self.conv(state)
```

```
        a = self.prelu(self.l1(state))
        a = self.prelu(self.l2(a))
        a = self.max_action * torch.tanh(self.l3(a))
        return a



class Critic(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(Critic, self).__init__()
        # Q1 architecture
        self.cnn = CNN(state_dim, 600)
        self.l1 = nn.Linear(600 + action_dim, 400)
        self.l2 = nn.Linear(400, 200)
        self.l3 = nn.Linear(200, 1)
        # Q2 architecture
        self.cnn2 = CNN(state_dim, 600)
        self.l4 = nn.Linear(600 + action_dim, 400)
        self.l5 = nn.Linear(400, 200)
        self.l6 = nn.Linear(200, 1)

        self.prelu = nn.PReLU()

    def forward(self, state, action):
        sa1 = self.cnn(state)
        sa1 = torch.cat([sa1, action], 1)
        q1 = self.prelu(self.l1(sa1))
        q1 = self.prelu(self.l2(q1))
        q1 = self.l3(q1)
        sa2 = self.cnn2(state)

        sa2 = torch.cat([sa2, action], 1)
        q2 = self.prelu(self.l4(sa2))
        q2 = self.prelu(self.l5(q2))
        q2 = self.l6(q2)
        return q1, q2

    def Q1(self, state, action):
        sa = self.cnn(state)
        sa = torch.cat([sa, action], 1)
        q1 = self.prelu(self.l1(sa))
        q1 = self.prelu(self.l2(q1))
        q1 = self.l3(q1)
        return q1
```

```
class TD3(object):
    def __init__(
        self,
        state_dim,
        action_dim,
        max_action,
        discount=0.99,
        tau=0.005,
        policy_noise=0.2,
        noise_clip=0.5,
        policy_freq=2,
        lr=3e-4,
    ):
        self.actor = Actor(state_dim, action_dim,
max_action).to(device)
        self.actor_target = copy.deepcopy(self.actor)
        self.actor_optimizer =
torch.optim.Adam(self.actor.parameters(), lr=lr)
        self.critic = Critic(state_dim, action_dim).to(device)
        self.critic_target = copy.deepcopy(self.critic)
        self.critic_optimizer =
torch.optim.Adam(self.critic.parameters(), lr=lr)
        self.max_action = max_action
        self.discount = discount
        self.tau = tau
        self.policy_noise = policy_noise
        self.noise_clip = noise_clip
        self.policy_freq = policy_freq
        self.total_it = 0

        self.actor_scheduler =
torch.optim.lr_scheduler.ReduceLROnPlateau(self.actor_optimizer
, factor=0.5, patience=20,  verbose=True)
        self.critic_scheduler =
torch.optim.lr_scheduler.ReduceLROnPlateau(self.critic_optimize
r, factor=0.5, patience=20,  verbose=True)

    def select_action(self, state_tup):
        ind_state, imm_state = state_tup
        if len(ind_state.shape) == 2:
            ind_state =
torch.FloatTensor([ind_state]).to(device)
            imm_state =
```

```
torch.FloatTensor([imm_state]).to(device)
        else:
            ind_state = torch.FloatTensor(ind_state).to(device)
            imm_state = torch.FloatTensor(imm_state).to(device)
        action = self.actor(ind_state,
imm_state).cpu().data.numpy()
        return action

    def train(self, replay_buffer, batch_size=100):
        self.total_it += 1
        # Sample replay buffer
        (
            state,
            action,
            next_state,
            reward,
            not_done,
        ) = replay_buffer.sample(batch_size)

        with torch.no_grad():
            # Select action according to policy
            noise = (torch.randn_like(action) *
self.policy_noise).clamp(
                -self.noise_clip, self.noise_clip
            )
            next_action = (
                self.actor_target(next_state) + noise
            ).clamp(-self.max_action, self.max_action)
            # Compute the target Q value
            target_Q1, target_Q2 =
self.critic_target(next_state, next_action)
            target_Q = torch.min(target_Q1, target_Q2)
            target_Q = reward + not_done * self.discount *
target_Q
        # Get current Q estimates
        current_Q1, current_Q2 = self.critic(state, action)
        # Compute critic loss
        critic_loss = F.mse_loss(current_Q1, target_Q) +
F.mse_loss(
            current_Q2, target_Q
        )
        # Optimize the critic
        self.critic_optimizer.zero_grad()
        critic_loss.backward()
```

```
torch.nn.utils.clip_grad_norm_(self.critic.parameters(), 1)
        self.critic_optimizer.step()

        # Delayed policy updates
        if self.total_it % self.policy_freq == 0:
            # Compute actor losse
            actor_loss = -self.critic.Q1(state,
self.actor(state)).mean()
            # Optimize the actor
            self.actor_optimizer.zero_grad()

            actor_loss.backward()

torch.nn.utils.clip_grad_norm_(self.actor.parameters(), 1)

            self.actor_optimizer.step()
            # self.actor_scheduler.step(actor_loss)
            # Update the frozen target models
            for param, target_param in zip(
                self.critic.parameters(),
self.critic_target.parameters()
            ):
                target_param.data.copy_(
                    self.tau * param.data + (1 - self.tau) *
target_param.data
                )
            for param, target_param in zip(
                self.actor.parameters(),
self.actor_target.parameters()
            ):
                target_param.data.copy_(
                    self.tau * param.data + (1 - self.tau) *
target_param.data
                )

    def save(self, filename):
        torch.save(self.critic.state_dict(), filename +
"_critic")
        torch.save(self.critic_optimizer.state_dict(), filename
+ "_critic_optimizer")
        torch.save(self.actor.state_dict(), filename +
"_actor")
        torch.save(self.actor_optimizer.state_dict(), filename
+ "_actor_optimizer")
```

```
    def load(self, filename):
        self.critic.load_state_dict(torch.load(filename +
"_critic"))
        self.critic_optimizer.load_state_dict(
            torch.load(filename + "_critic_optimizer")
        )
        self.critic_target = copy.deepcopy(self.critic)
        self.actor.load_state_dict(torch.load(filename +
"_actor"))

self.actor_optimizer.load_state_dict(torch.load(filename +
"_actor_optimizer"))
        self.actor_target = copy.deepcopy(self.actor)




class ReplayBuffer(object):
    def __init__(self, state_dim, action_dim,
max_size=int(1e6)):
        self.max_size = max_size
        self.ptr = 0
        self.size = 0
        self.action = np.zeros((max_size, action_dim))
        if type(state_dim) == tuple or type(state_dim) == list:
            full_state_dim = [max_size] + [x for x in
state_dim]
        else:
            full_state_dim = (max_size, state_dim)
        self.state = np.zeros(full_state_dim)
        self.next_state = np.zeros(full_state_dim)
        self.reward = np.zeros((max_size, 1))
        self.not_done = np.zeros((max_size, 1))
        self.device = torch.device("cuda" if
torch.cuda.is_available() else "cpu")

    def add(self, state, action, next_state, reward, done):
        self.next_state[self.ptr] = next_state
        self.action[self.ptr] = action
        self.state[self.ptr] = state
        self.reward[self.ptr] = reward
        self.not_done[self.ptr] = 1.0 - done
        self.ptr = (self.ptr + 1) % self.max_size
        self.size = min(self.size + 1, self.max_size)
```

```
    def sample(self, batch_size):
        ind = np.random.randint(0, self.size, size=batch_size)
        return (
            torch.FloatTensor(self.state[ind]).to(self.device),

torch.FloatTensor(self.action[ind]).to(self.device),

torch.FloatTensor(self.next_state[ind]).to(self.device),

torch.FloatTensor(self.reward[ind]).to(self.device),

torch.FloatTensor(self.not_done[ind]).to(self.device),
        )
```