

```

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import DataLoader, TensorDataset
from sklearn.model_selection import train_test_split
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import r2_score

import torch

print("Is CUDA available: ", torch.cuda.is_available())
print("CUDA version: ", torch.version.cuda)
print("Number of CUDA devices: ", torch.cuda.device_count())
print("CUDA Device Name: ", torch.cuda.get_device_name(0) if torch.cuda.device_count() > 0 else "No CUDA device")

Is CUDA available: True
CUDA version: 12.1
Number of CUDA devices: 1
CUDA Device Name: Tesla T4

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

print(device)

cuda

row_data_file = 'TSLA_stock_data_2023.csv'
predict_data_file = 'TSLA_stock_data_2024.csv'

# Creating sliding windows
def slide_windows(features):

    # We need to ensure we have data for t-1, t, t+1 without index errors
    windowed_data = []
    predict_prices = [] # List to store the target 'Close' prices
    cycle_data=[]

    # Handling daily cycles
    cycle_length = 7
    for cycle in range(len(features) - cycle_length + 1):
        start_index = cycle
        end_index = start_index + cycle_length
        cycle_datas = features.iloc[start_index:end_index,1:6].values
        cycle_data.append(cycle_datas)

    # Creating sliding windows within the cycle
    for i in range(14,len(cycle_data)-7): # Avoiding index error by stopping before the last day
        pre_previous = cycle_data[i - 14]
        previous = cycle_data[i- 7]
        current_state = cycle_data[i]

        combined_features = np.concatenate([pre_previous, previous, current_state]).reshape(1,-1).squeeze()
        windowed_data.append(combined_features)
        predict_prices.append(cycle_data[i+7][-1][3])

    return windowed_data, predict_prices

```

```

# Load data
row_data = pd.read_csv(row_data_file)

windowed_data, predict_prices = slide_windows(row_data)
# Convert to PyTorch tensors
state = torch.tensor(windowed_data, dtype=torch.float32).to(device)
predict_price = torch.tensor(predict_prices, dtype=torch.float32).to(device)
# state[0]

predict_price[0]

    tensor(113.0300, device='cuda:0')

# divide the data into training part and test part
state_train, state_test, predict_price_train, predict_price_test = train_test_split(state, predict_price, test_size=0.2, random_state=42)
train_loader = DataLoader(TensorDataset(state_train, predict_price_train), batch_size=200, shuffle=True)
test_loader = DataLoader(TensorDataset(state_test, predict_price_test), batch_size=300)
len(train_loader)

7

# construct NN
class NN(nn.Module):
    def __init__(self, n_observations):
        super(NN, self).__init__()
        self.layer1 = nn.Linear(n_observations, 128)
        self.layer2 = nn.Linear(128, 128)
        self.layer3 = nn.Linear(128, 1)

    def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        return self.layer3(x)

# Optimazation function, it do one step of gredient decent,
def optimize_model():
    for state, target in train_loader:
        current_value = value_net(state).squeeze()
        criterion = nn.SmoothL1Loss()
        loss = criterion(current_value, target)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    for state, target in test_loader:
        with torch.no_grad():
            test_valur = value_net(state).squeeze()
            l_test = criterion(test_valur, target)
    return loss.item(), l_test.item()

```

```

learning_rates = [1e-3] # Possible learning rates
weight_decays = [1e-2] # Possible weight decay values
n_observations = len(state[0])
epochs=1000

best_lr = None
best_weight_decay = None
lowest_test_loss = float('inf') # Initialize with infinity to ensure any first result is better

# value_net is your prediction function, take state as input and output is the prediction price
value_net = NN(n_observations).to(device)

for LR in learning_rates:
    for wd in weight_decays:
        print(f"Testing with LR = {LR} and Weight Decay = {wd}")
        # 'optimize' is an easy package to do Gradient decent, 'Adam' is a method to let learning rate decay as step go.
        optimizer = optim.Adam(value_net.parameters(), lr=LR, weight_decay=wd)

        # Run the training loop for this combination of parameters
        for epoch in range(epochs):
            l_train, l_test = optimize_model() # Use the current lr and wd in your optimization
            if epoch % 100 == 0: # Report every 100 epochs
                print(f'Epoch [{epoch+1}/{epochs}], L_train: {l_train}, L_test: {l_test}')

            # Save parameters if this is the best we've seen
            if l_test < lowest_test_loss:
                lowest_test_loss = l_test
                best_lr = LR
                best_weight_decay = wd

# Output the best parameters and the test loss achieved with them
print(f"Best Learning Rate: {best_lr}, Best Weight Decay: {best_weight_decay}, Lowest L_test: {lowest_test_loss}")
#

Testing with LR = 0.001 and Weight Decay = 0.01
Epoch [1/1000], L_train: 162461.265625, L_test: 276378.8125
Epoch [101/1000], L_train: 29093.708984375, L_test: 22728.095703125
Epoch [201/1000], L_train: 5718.72265625, L_test: 9528.4375
Epoch [301/1000], L_train: 7027.38671875, L_test: 7680.0390625
Epoch [401/1000], L_train: 2199.861083984375, L_test: 5758.205078125
Epoch [501/1000], L_train: 1447.7686767578125, L_test: 3436.920166015625
Epoch [601/1000], L_train: 1023.7678833007812, L_test: 2589.6552734375
Epoch [701/1000], L_train: 664.7922973632812, L_test: 2153.30517578125
Epoch [801/1000], L_train: 1064.9866943359375, L_test: 1575.3363037109375
Epoch [901/1000], L_train: 251.06939697265625, L_test: 826.0755615234375
Best Learning Rate: 0.001, Best Weight Decay: 0.01, Lowest L_test: 352.7765197753906

# continue optimize
# start from last training result
epochs=1000
LR = 1e-15
Weight_decay=0.01
optimizer = optim.Adam(value_net.parameters(), lr=LR, weight_decay=Weight_decay)
value_net.train()

run_count = 0

# while True:
for epoch in range(epochs):
    l_train, l_test = optimize_model()
    if epoch % 10 == 0:
        print(f'Epoch [{epoch+1}/{epochs}], L_train: {l_train}, L_test: {l_test}')
    # Save parameters if this is the best we've seen
    if l_test < lowest_test_loss:
        lowest_test_loss = l_test
        torch.save(value_net.state_dict(), 'model_parameters.pth')
        print("store result from ", f'Epoch [{epoch+1}/{epochs}], L_train: {l_train}, L_test: {l_test}')
    # run_count += 1

    # # Check the stopping condition
    # if l_test <= 10 or run_count > 2:
    #     print("Stopping the loop.")
    #     break
print(f"lowest L_test: {lowest_test_loss}")

```

```

epoch [461/1000], L_train: 7.199551918029785, L_test: 6.814588069915715
Epoch [471/1000], L_train: 7.3317389488220215, L_test: 6.814602851867676
Epoch [481/1000], L_train: 7.366457939147949, L_test: 6.814671039581299
Epoch [491/1000], L_train: 7.333654880523682, L_test: 6.814602851867676
Epoch [501/1000], L_train: 7.574687957763672, L_test: 6.814589500427246
Epoch [511/1000], L_train: 7.507099151611328, L_test: 6.814543724060059
Epoch [521/1000], L_train: 7.600534915924072, L_test: 6.814563274383545
Epoch [531/1000], L_train: 7.561635494232178, L_test: 6.814645290374756
Epoch [541/1000], L_train: 7.577398777008057, L_test: 6.814754009246826
Epoch [551/1000], L_train: 7.513708591461182, L_test: 6.8148698806762695
Epoch [561/1000], L_train: 7.762601375579834, L_test: 6.814798831939697
Epoch [571/1000], L_train: 7.594850063323975, L_test: 6.814935684204102
Epoch [581/1000], L_train: 7.111621856689453, L_test: 6.815001487731934
Epoch [591/1000], L_train: 6.797135353088379, L_test: 6.814938545227051
Epoch [601/1000], L_train: 7.957949638366699, L_test: 6.815152645111084
Epoch [611/1000], L_train: 8.018989562988281, L_test: 6.815042018890381
Epoch [621/1000], L_train: 7.075555801391602, L_test: 6.815109729766846
Epoch [631/1000], L_train: 7.907120227813721, L_test: 6.815171241760254
Epoch [641/1000], L_train: 6.420556545257568, L_test: 6.815177917480469
Epoch [651/1000], L_train: 6.727065086364746, L_test: 6.81523323059082
Epoch [661/1000], L_train: 7.902218818664551, L_test: 6.815180778503418
Epoch [671/1000], L_train: 7.954980373382568, L_test: 6.815237998962402
Epoch [681/1000], L_train: 7.735865592956543, L_test: 6.81520414352417
Epoch [691/1000], L_train: 7.608218669891357, L_test: 6.8152055740356445
Epoch [701/1000], L_train: 7.67433500289917, L_test: 6.815258026123047
Epoch [711/1000], L_train: 6.930333137512207, L_test: 6.815212726593018
Epoch [721/1000], L_train: 6.815947532653809, L_test: 6.815201282501221
Epoch [731/1000], L_train: 8.009689331054688, L_test: 6.815311908721924
Epoch [741/1000], L_train: 7.853996753692627, L_test: 6.815319538116455
Epoch [751/1000], L_train: 6.910910129547119, L_test: 6.815262794494629
Epoch [761/1000], L_train: 6.946164131164551, L_test: 6.815268516540527
Epoch [771/1000], L_train: 7.743074417114258, L_test: 6.815254211425781
Epoch [781/1000], L_train: 7.859787940979004, L_test: 6.815242290496826
Epoch [791/1000], L_train: 8.072708129882812, L_test: 6.81524658203125
Epoch [801/1000], L_train: 7.646492958068848, L_test: 6.815240859985352
Epoch [811/1000], L_train: 8.139105796813965, L_test: 6.815210342407227
Epoch [821/1000], L_train: 7.160065174102783, L_test: 6.815184116363525
Epoch [831/1000], L_train: 6.862547397613525, L_test: 6.815182685852051
Epoch [841/1000], L_train: 8.097982406616211, L_test: 6.8152289390563965
Epoch [851/1000], L_train: 6.882308483123779, L_test: 6.815268039703369
Epoch [861/1000], L_train: 7.2434401512146, L_test: 6.8152971267700195
Epoch [871/1000], L_train: 7.494387626647949, L_test: 6.815329074859619
Epoch [881/1000], L_train: 7.522837162017822, L_test: 6.8152594566345215
Epoch [891/1000], L_train: 6.978073596954346, L_test: 6.815281867980957
Epoch [901/1000], L_train: 7.04850959777832, L_test: 6.815302848815918
Epoch [911/1000], L_train: 7.125743865966797, L_test: 6.815364360809326
Epoch [921/1000], L_train: 7.850288391113281, L_test: 6.815340042114258
Epoch [931/1000], L_train: 7.664575576782227, L_test: 6.815395832061768
Epoch [941/1000], L_train: 6.84632682800293, L_test: 6.815346717834473
Epoch [951/1000], L_train: 7.1336774826049805, L_test: 6.815299987792969
Epoch [961/1000], L_train: 7.930482387542725, L_test: 6.815272331237793
Epoch [971/1000], L_train: 7.133889675140381, L_test: 6.815192699432373
Epoch [981/1000], L_train: 7.781347751617432, L_test: 6.8152875900268555
Epoch [991/1000], L_train: 7.050561904907227, L_test: 6.815263748168945
lowest L_test: 6.761529445648193

```

```

# # load the trained parameter to the model
loaded_model = NN(n_observations).to(device)
loaded_model.load_state_dict(torch.load('model_parameters.pth'))
value_net=loaded_model

```

```
<All keys matched successfully>
```

```

# Load data to predict
windowed_data, predict_prices = slide_windows(pd.read_csv(predict_data_file))
# Convert to PyTorch tensors
state_new = torch.tensor(windowed_data, dtype=torch.float32).to(device)
real_price = torch.tensor(predict_prices, dtype=torch.float32).to(device)

l_test = []
predict_price=[]
replace = 0
n=0
# predict price
value_net.eval()
with torch.no_grad():
    criterion = nn.SmoothL1Loss()
    predict_price=loaded_model(state_new).squeeze() #predict price
    adjusted_predict_price = predict_price.clone()

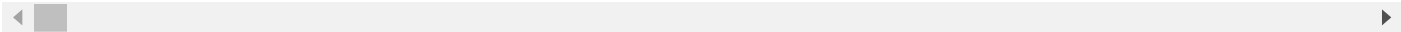
# Calculate the difference for each hour
for i in range(real_price.size(0)): # Assuming real_price and predict_price are of the same length
    # Calculate loss for each pair
    loss = criterion(real_price[i], predict_price[i])

    # Check if loss is greater than 15, replace predicted price with real price if condition is met
    if loss.item() > 15:
        predict_price[i] = real_price[i-7]
        replace += 1
    n+=1
    loss = criterion(real_price[i], predict_price[i])
    l_test.append(loss.item())

# print(predict_price)
print(l_test)
# print(state_new)
print(replace,n)

[87.41099548339844, 86.489990234375, 89.58999633789062, 89.80450439453125, 90.70999145507812, 92.08489990234375, 93.18800354003906, 2.44
505 505

```



```

real_price_data = real_price.cpu().numpy()
predict_price_data = predict_price.cpu().numpy()

plt.figure(figsize=(8, 5))
plt.plot(real_price_data, color='black', label='Real Stock Price')
plt.plot(predict_price_data, color='green', label='Predicted Stock Price in NN')
plt.title('Stock Price Prediction')
plt.xlabel('Time')
plt.ylabel('Stock Price')
plt.legend()
plt.show()

```

```
plt.figure(figsize=(8, 5))
calculate = np.abs((real_price_data - predict_price_data) / real_price_data) * 100
plt.plot(calculate, color='red', label='Prediction Error (%)')
plt.title('Prediction Error')
plt.xlabel('Time')
plt.ylabel('Error (%)')
plt.legend()
plt.show()
```

