Detailed Explanation of LSTM + DQN (and PPO) Trading Workflow

Your Name

February 24, 2025

Contents

1 Introduction

This document provides a step-by-step explanation of the Python program that trains:

1. An LSTM neural network (for price prediction) using Optuna for hyperparameter tuning.
2. A Deep Q-Network (DQN) agent (from Stable-Baselines3) in a custom Stock Trading Gym environment for reinforcement learning.

Additionally, it will cover:

- The benefits of potentially substituting the DQN algorithm with Proximal Policy Optimization (PPO).
- A detailed guide on how to modify the existing code to implement a PPO agent instead of a DQN agent.

2 Full Source Code

In this section, we display the full source code. Each component will be described in detail in subsequent sections.

Listing 1: Full program source code.

```
import os
import sys
import argparse
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import logging
from tabulate import tabulate

from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout, Bidirectional
from tensorflow.keras.optimizers import Adam, Nadam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.losses import Huber
from tensorflow.keras.regularizers import l2

import xgboost as xgb
import optuna
from optuna.integration import KerasPruningCallback

# For Reinforcement Learning
import gym
from gym import spaces
from stable_baselines3 import DQN
from stable_baselines3.common.vec_env import DummyVecEnv

# Suppress TensorFlow warnings
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'  # Suppress INFO/WARNING

# Configure logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')


###############################
# 1. Data Loading / Indicators
###############################
def load_data(file_path):
    logging.info(f"Loading data from: {file_path}")
    try:
        df = pd.read_csv(file_path, parse_dates=['time'])
    except FileNotFoundError:
```

```
            logging.error(f"File not found: {file_path}")
            sys.exit(1)
        except pd.errors.ParserError as e:
            logging.error(f"Error parsing CSV file: {e}")
            sys.exit(1)
        except Exception as e:
            logging.error(f"Unexpected error: {e}")
            sys.exit(1)

        rename_mapping = {
            'time': 'Date',
            'open': 'Open',
            'high': 'High',
            'low': 'Low',
            'close': 'Close'
        }
        df.rename(columns=rename_mapping, inplace=True)

        logging.info(f"Data columns after renaming: {df.columns.tolist()}")

        df.sort_values('Date', inplace=True)
        df.reset_index(drop=True, inplace=True)
        logging.info("Data loaded and sorted successfully.")
        return df


def compute_rsi(series, window=14):
    delta = series.diff()
    gain = delta.where(delta > 0, 0).rolling(window=window).mean()
    loss = -delta.where(delta < 0, 0).rolling(window=window).mean()
    RS = gain / (loss + 1e-9)
    return 100 - (100 / (1 + RS))


def compute_macd(series, span_short=12, span_long=26, span_signal=9):
    ema_short = series.ewm(span=span_short, adjust=False).mean()
    ema_long = series.ewm(span=span_long, adjust=False).mean()
    macd_line = ema_short - ema_long
    signal_line = macd_line.ewm(span=span_signal, adjust=False).mean()
    return macd_line - signal_line  # histogram


def compute_obv(df):
    signed_volume = (np.sign(df['Close'].diff()) * df['Volume']).fillna(0)
    return signed_volume.cumsum()


def compute_adx(df, window=14):
    """Pseudo-ADX approach using rolling True Range / Close."""
    df['H-L'] = df['High'] - df['Low']
    df['H-Cp'] = (df['High'] - df['Close'].shift(1)).abs()
    df['L-Cp'] = (df['Low'] - df['Close'].shift(1)).abs()
    tr = df[['H-L','H-Cp','L-Cp']].max(axis=1)
    tr_rolling = tr.rolling(window=window).mean()

    adx_placeholder = tr_rolling / (df['Close'] + 1e-9)
    df.drop(['H-L','H-Cp','L-Cp'], axis=1, inplace=True)
    return adx_placeholder


def compute_bollinger_bands(series, window=20, num_std=2):
    sma = series.rolling(window=window).mean()
    std = series.rolling(window=window).std()
    upper = sma + num_std * std
    lower = sma - num_std * std
    bandwidth = (upper - lower) / (sma + 1e-9)
    return upper, lower, bandwidth


def compute_mfi(df, window=14):
    typical_price = (df['High'] + df['Low'] + df['Close']) / 3
    money_flow = typical_price * df['Volume']
    prev_tp = typical_price.shift(1)
```

```python
    flow_pos = money_flow.where(typical_price > prev_tp, 0)
    flow_neg = money_flow.where(typical_price < prev_tp, 0)
    pos_sum = flow_pos.rolling(window=window).sum()
    neg_sum = flow_neg.rolling(window=window).sum()
    mfi = 100 - (100 / (1 + pos_sum/(neg_sum+1e-9)))
    return mfi


def calculate_technical_indicators(df):
    logging.info("Calculating technical indicators...")

    df['RSI'] = compute_rsi(df['Close'], window=14)
    df['MACD'] = compute_macd(df['Close'])
    df['OBV'] = compute_obv(df)
    df['ADX'] = compute_adx(df)

    upper_bb, lower_bb, bb_width = compute_bollinger_bands(df['Close'], window=20, num_std=2)
    df['BB_Upper'] = upper_bb
    df['BB_Lower'] = lower_bb
    df['BB_Width'] = bb_width

    df['MFI'] = compute_mfi(df, window=14)

    df['SMA_5'] = df['Close'].rolling(window=5).mean()
    df['SMA_10'] = df['Close'].rolling(window=10).mean()
    df['EMA_5'] = df['Close'].ewm(span=5, adjust=False).mean()
    df['EMA_10'] = df['Close'].ewm(span=10, adjust=False).mean()

    df['STDDEV_5'] = df['Close'].rolling(window=5).std()
    df.dropna(inplace=True)
    logging.info("Technical indicators calculated successfully.")
    return df


###############################
# 2. ARGUMENT PARSING
###############################
def parse_arguments():
    parser = argparse.ArgumentParser(description='Train LSTM and DQN models for stock trading.
        ')
    parser.add_argument('csv_path', type=str, help='Path to the CSV data file (with columns
        time,open,high,low,close,volume).')
    return parser.parse_args()


###############################
# 3. MAIN
###############################
def main():
    # 1) Parse args
    args = parse_arguments()
    csv_path = args.csv_path

    # 2) Load data & advanced indicators
    data = load_data(csv_path)
    data = calculate_technical_indicators(data)

    # EXCLUDE 'Close' from feature inputs
    feature_columns = [
        'SMA_5', 'SMA_10', 'EMA_5', 'EMA_10', 'STDDEV_5',
        'RSI', 'MACD', 'ADX', 'OBV', 'Volume', 'Open', 'High', 'Low',
        'BB_Upper','BB_Lower','BB_Width','MFI'
    ]
    target_column = 'Close'
    data = data[['Date'] + feature_columns + [target_column]]
    data.dropna(inplace=True)

    # 3) Scale
    scaler_features = MinMaxScaler()
    scaler_target = MinMaxScaler()

    X_all = data[feature_columns].values
    y_all = data[[target_column]].values
```

```
    X_scaled = scaler_features.fit_transform(X_all)
    y_scaled = scaler_target.fit_transform(y_all).flatten()

    # 4) Create LSTM Sequences
    def create_sequences(features, target, window_size=15):
        X_seq, y_seq = [], []
        for i in range(len(features) - window_size):
            X_seq.append(features[i:i+window_size])
            y_seq.append(target[i+window_size])
        return np.array(X_seq), np.array(y_seq)

    window_size = 15
    X, y = create_sequences(X_scaled, y_scaled, window_size)

    # 5) Train/Val/Test Split
    train_size = int(len(X)*0.7)
    val_size = int(len(X)*0.15)
    test_size = len(X) - train_size - val_size

    X_train = X[:train_size]
    y_train = y[:train_size]
    X_val   = X[train_size:train_size+val_size]
    y_val   = y[train_size:train_size+val_size]
    X_test  = X[train_size+val_size:]
    y_test  = y[train_size+val_size:]

    logging.info(f"Scaled training features shape: {X_train.shape}")
    logging.info(f"Scaled validation features shape: {X_val.shape}")
    logging.info(f"Scaled testing features shape: {X_test.shape}")
    logging.info(f"Scaled training target shape: {y_train.shape}")
    logging.info(f"Scaled validation target shape: {y_val.shape}")
    logging.info(f"Scaled testing target shape: {y_test.shape}")

    # 6) GPU/CPU Config
    def configure_device():
        gpus = tf.config.list_physical_devices('GPU')
        if gpus:
            try:
                for gpu in gpus:
                    tf.config.experimental.set_memory_growth(gpu, True)
                logging.info(f"{len(gpus)} GPU(s) detected and configured.")
            except RuntimeError as e:
                logging.error(e)
        else:
            logging.info("No GPU detected, using CPU.")

    configure_device()

    # 7) Build LSTM
    def build_advanced_lstm(input_shape, hyperparams):
        model = Sequential()
        for i in range(hyperparams['num_lstm_layers']):
            return_seqs = (i < hyperparams['num_lstm_layers'] - 1)
            model.add(Bidirectional(
                LSTM(hyperparams['lstm_units'],
                     return_sequences=return_seqs,
                     kernel_regularizer=tf.keras.regularizers.l2(0.001)
                ), input_shape=input_shape if i==0 else None))
            model.add(Dropout(hyperparams['dropout_rate']))

        model.add(Dense(1, activation='linear'))

        # Optimizer
        if hyperparams['optimizer'] == 'Adam':
            opt = Adam(learning_rate=hyperparams['learning_rate'], decay=hyperparams['decay'])
        elif hyperparams['optimizer'] == 'Nadam':
            opt = Nadam(learning_rate=hyperparams['learning_rate'])
        else:
            opt = Adam(learning_rate=hyperparams['learning_rate'])

        model.compile(optimizer=opt, loss=Huber(), metrics=['mae'])
        return model
```

```python

    # 8) Optuna Tuning
    def objective(trial):
        num_lstm_layers = trial.suggest_int('num_lstm_layers', 1, 3)
        lstm_units = trial.suggest_categorical('lstm_units', [32, 64, 96, 128])
        dropout_rate = trial.suggest_float('dropout_rate', 0.1, 0.5)
        learning_rate = trial.suggest_loguniform('learning_rate', 1e-5, 1e-2)
        optimizer_name = trial.suggest_categorical('optimizer', ['Adam', 'Nadam'])
        decay = trial.suggest_float('decay', 0.0, 1e-4)

        hyperparams = {
            'num_lstm_layers': num_lstm_layers,
            'lstm_units': lstm_units,
            'dropout_rate': dropout_rate,
            'learning_rate': learning_rate,
            'optimizer': optimizer_name,
            'decay': decay
        }

        model_ = build_advanced_lstm((X_train.shape[1], X_train.shape[2]), hyperparams)

        early_stop = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
        lr_reduce  = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=5, min_lr=1e-6)

        cb_prune = KerasPruningCallback(trial, 'val_loss')

        history = model_.fit(
            X_train, y_train,
            epochs=100,
            batch_size=16,
            validation_data=(X_val, y_val),
            callbacks=[early_stop, lr_reduce, cb_prune],
            verbose=0
        )
        val_mae = min(history.history['val_mae'])
        return val_mae

    logging.info("Starting hyperparameter optimization with Optuna...")
    study = optuna.create_study(direction='minimize')
    study.optimize(objective, n_trials=50)  # might take a long time

    best_params = study.best_params
    logging.info(f"Best Hyperparameters from Optuna: {best_params}")

    # 9) Train Best LSTM
    best_model = build_advanced_lstm((X_train.shape[1], X_train.shape[2]), best_params)
    early_stop2 = EarlyStopping(monitor='val_loss', patience=20, restore_best_weights=True)
    lr_reduce2  = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=5, min_lr=1e-6)

    logging.info("Training the best LSTM model with optimized hyperparameters...")
    history = best_model.fit(
        X_train, y_train,
        epochs=300,
        batch_size=16,
        validation_data=(X_val, y_val),
        callbacks=[early_stop2, lr_reduce2],
        verbose=1
    )

    # 10) Evaluate
    def evaluate_model(model, X_test, y_test):
        logging.info("Evaluating model...")
        # Predict scaled
        y_pred_scaled = model.predict(X_test).flatten()
        y_pred_scaled = np.clip(y_pred_scaled, 0, 1)  # clamp if needed
        # Inverse
        y_pred = scaler_target.inverse_transform(y_pred_scaled.reshape(-1,1)).flatten()
        y_test_actual = scaler_target.inverse_transform(y_test.reshape(-1,1)).flatten()

        mse = mean_squared_error(y_test_actual, y_pred)
        rmse = np.sqrt(mse)
        mae = mean_absolute_error(y_test_actual, y_pred)
```

```python
    r2  = r2_score(y_test_actual, y_pred)

    # Directional accuracy
    direction_actual = np.sign(np.diff(y_test_actual))
    direction_pred   = np.sign(np.diff(y_pred))
    directional_accuracy = np.mean(direction_actual == direction_pred)

    logging.info(f"Test MSE: {mse}")
    logging.info(f"Test RMSE: {rmse}")
    logging.info(f"Test MAE: {mae}")
    logging.info(f"Test R2 Score: {r2}")
    logging.info(f"Directional Accuracy: {directional_accuracy}")

    # Plot
    plt.figure(figsize=(14, 7))
    plt.plot(y_test_actual, label='Actual Price')
    plt.plot(y_pred, label='Predicted Price')
    plt.title('Actual vs Predicted Prices')
    plt.xlabel('Time Step')
    plt.ylabel('Price')
    plt.legend()
    plt.grid(True)
    plt.savefig('actual_vs_predicted.png')
    plt.close()
    logging.info("Actual vs Predicted plot saved as 'actual_vs_predicted.png'")

    # Tabulate first 40 predictions
    table_data = []
    for i in range(min(40, len(y_test_actual))):
        table_data.append([i, round(y_test_actual[i],2), round(y_pred[i],2)])
    headers = ["Index", "Actual Price", "Predicted Price"]
    print(tabulate(table_data, headers=headers, tablefmt="pretty"))

    return mse, rmse, mae, r2, directional_accuracy

mse, rmse, mae, r2, directional_accuracy = evaluate_model(best_model, X_test, y_test)

# 11) Save
best_model.save('optimized_lstm_model.h5')
import joblib
joblib.dump(scaler_features, 'scaler_features.save')
joblib.dump(scaler_target, 'scaler_target.save')
logging.info("Model and scalers saved as 'optimized_lstm_model.h5', 'scaler_features.save', and 'scaler_target.save'.")

#########################################
# 12) Reinforcement Learning Environment
#########################################
class StockTradingEnv(gym.Env):
    """
    A simple stock trading environment for OpenAI Gym
    """
    metadata = {'render.modes': ['human']}

    def __init__(self, df, initial_balance=10000):
        super().__init__()
        self.df = df.reset_index()
        self.initial_balance = initial_balance
        self.balance = initial_balance
        self.net_worth = initial_balance
        self.max_steps = len(df)
        self.current_step = 0
        self.shares_held = 0
        self.cost_basis = 0

        # We re-use feature_columns from above
        # (Excluding 'Close' from the observation)
        # Actions: 0=Sell, 1=Hold, 2=Buy
        self.action_space = spaces.Discrete(3)

        # Observations => advanced feature columns + 3 additional (balance, shares, cost_basis)
        self.observation_space = spaces.Box(
```

```
            low=0,
            high=1,
            shape=(len(feature_columns) + 3,),
            dtype=np.float32
        )

    def reset(self):
        self.balance = self.initial_balance
        self.net_worth = self.initial_balance
        self.current_step = 0
        self.shares_held = 0
        self.cost_basis = 0
        return self._next_observation()

    def _next_observation(self):
        obs = self.df.loc[self.current_step, feature_columns].values
        # Simple normalization by max to keep it [0,1]
        obs = obs / np.max(obs) if np.max(obs)!=0 else obs

        additional = np.array([
            self.balance / self.initial_balance,
            self.shares_held / 100.0,
            self.cost_basis / self.initial_balance
        ])
        return np.concatenate([obs, additional])

    def step(self, action):
        current_price = self.df.loc[self.current_step, 'Close']

        if action == 2:  # Buy
            total_possible = self.balance // current_price
            shares_bought = total_possible
            if shares_bought > 0:
                self.balance -= shares_bought * current_price
                self.shares_held += shares_bought
                self.cost_basis = (
                    (self.cost_basis * (self.shares_held - shares_bought)) +
                    (shares_bought * current_price)
                ) / self.shares_held

        elif action == 0:  # Sell
            if self.shares_held > 0:
                self.balance += self.shares_held * current_price
                self.shares_held = 0
                self.cost_basis = 0

        self.net_worth = self.balance + self.shares_held * current_price
        self.current_step += 1

        done = (self.current_step >= self.max_steps - 1)
        reward = self.net_worth - self.initial_balance

        obs = self._next_observation()
        return obs, reward, done, {}

    def render(self, mode='human'):
        profit = self.net_worth - self.initial_balance
        print(f"Step: {self.current_step}")
        print(f"Balance: {self.balance}")
        print(f"Shares held: {self.shares_held} (Cost Basis: {self.cost_basis})")
        print(f"Net worth: {self.net_worth}")
        print(f"Profit: {profit}")

def train_dqn_agent(env):
    logging.info("Training DQN Agent...")
    try:
        model = DQN(
            'MlpPolicy',
            env,
            verbose=1,
            learning_rate=1e-3,
            buffer_size=10000,
            learning_starts=1000,
```

```
                batch_size=64,
                tau=1.0,
                gamma=0.99,
                train_freq=4,
                target_update_interval=1000,
                exploration_fraction=0.1,
                exploration_final_eps=0.02,
                tensorboard_log="./dqn_stock_tensorboard/"
            )
            model.learn(total_timesteps=100000)
            model.save("dqn_stock_trading")
            logging.info("DQN Agent trained and saved as 'dqn_stock_trading.zip'.")
            return model
        except Exception as e:
            logging.error(f"Error training DQN Agent: {e}")
            sys.exit(1)

    # Initialize RL environment
    logging.info("Initializing and training DQN environment...")
    trading_env = StockTradingEnv(data)
    trading_env = DummyVecEnv([lambda: trading_env])

    # Train
    dqn_model = train_dqn_agent(trading_env)

    logging.info("All tasks complete. Exiting.")


if __name__ == "__main__":
    main()
```

3 Step-by-Step Explanation

3.1 Imports and Configuration

- `numpy, pandas, sklearn` are used for data manipulation and scaling.
- `matplotlib` and `seaborn` are used for plotting.
- `tensorflow.keras` is used for building the LSTM model.
- `optuna` is used for hyperparameter tuning.
- `gym` and `stable_baselines3` are used for the reinforcement learning (RL) portion of the code.
- The environment variable `TF_CPP_MIN_LOG_LEVEL` suppresses TensorFlow messages to keep logs clean.

3.2 Data Loading and Preprocessing (Functions in Step 1)

1. **`load_data(file_path)`**:
   - Reads a CSV file, expecting columns `time`, `open`, `high`, `low`, `close`, `volume`.
   - Renames these columns to `Date`, `Open`, `High`, `Low`, `Close`.
   - Sorts the DataFrame by date, resets the index, and returns the processed DataFrame.
2. **Technical Indicators Functions**: The code defines several functions to compute widely used technical indicators:
   - `compute_rsi(series, window=14)`: Calculates the Relative Strength Index over a given window.
   - `compute_macd(series, ...)`: Computes the Moving Average Convergence Divergence (MACD) histogram.
   - `compute_obv(df)`: Computes On-Balance Volume.
   - `compute_adx(df, window=14)`: Returns a placeholder ADX-like measure.

- `compute_bollinger_bands(series, ...)`: Returns Bollinger Bands and the bandwidth.
- `compute_mfi(df, window=14)`: Computes the Money Flow Index.

3. **`calculate_technical_indicators(df)`**:
   - Applies all of the above functions on the `df`.
   - Creates short/medium moving averages like SMA, EMA.
   - Removes any resulting NaN values from the DataFrame.
   - Returns the DataFrame with all computed indicators.

3.3 Argument Parsing (Step 2)

- Uses `argparse` to read a single argument, `csv_path`, which specifies the path to the CSV data file.

3.4 Main Function (Step 3)

The **main** function orchestrates the entire workflow:

1. **Parse arguments**: Calls `parse_arguments()` to get `csv_path`.
2. **Load and process data**:
   - Calls `load_data(csv_path)` to load the CSV.
   - Calls `calculate_technical_indicators(data)` to augment it with technical features.
3. **Setup feature and target columns**: The code excludes `Close` from the features (since it is the prediction target) and identifies the target as the `Close` column.
4. **Scaling**:
   - Instantiates two `MinMaxScalers`: one for features and one for the target.
   - Fits and transforms the data to a 0-1 range.
5. **Sequence Creation** (`create_sequences`):
   - Converts a time series of scaled features/targets into 3D sequences (samples $\times$ window_size $\times$ number_of_features).
   - Correspondingly, the targets become a vector of length `(original_length - window_size)`.
6. **Train-Validation-Test Split**: Splits the resulting 3D array (and 1D targets) as 70% train, 15% val, and 15% test.
7. **GPU/CPU Configuration**: The function `configure_device()` attempts to set memory growth on any detected GPUs.

3.5 Building and Tuning the LSTM (Steps 7–9)

1. **`build_advanced_lstm`**:
   - Takes `input_shape` (the shape of the input sequence) and a `hyperparams` dictionary.
   - Builds a stacked or repeated LSTM architecture, each wrapped in a Bidirectional layer, with dropout in between.
   - A final Dense layer of size 1 is added to output the price prediction.
   - Compilation uses `Huber` loss and one of the possible optimizers (`Adam` or `Nadam`) with various learning rate/decay settings.
2. **Optuna Tuning** (`objective` function):

- Suggests a search space for the number of LSTM layers, units, dropout rate, learning rate, etc.
- Builds and trains the model with early stopping, learning rate reduction, and a pruning callback.
- Returns the best validation `MAE` found.
- This tuning is done in `study.optimize(..., n_trials=50)`.

3. **Train the best model**:

   - Retrieves the best hyperparameters from the study, builds a new model, and trains it again for up to 300 epochs (with patience).

3.6 Evaluation and Saving the Model (Steps 10–11)

1. **Evaluation**:

   - Predicts on test data.
   - Inverse-transforms these predictions using `scaler_target`.
   - Computes MSE, RMSE, MAE, R$\hat{2}$, and a directional accuracy metric.
   - Saves a plot of Actual vs. Predicted.
   - Prints a table of the first 40 predictions for inspection.

2. **Saving**:

   - Saves the trained model as `optimized_lstm_model.h5`.
   - Saves the fitted `MinMaxScalers` via `joblib`.

3.7 Reinforcement Learning Environment and DQN Training (Step 12)

- **`StockTradingEnv`**:
  - Inherits from `gym.Env`.
  - Observations are a combination of scaled technical features and some additional info (balance ratio, shares held, cost basis).
  - Action space has three discrete actions: `Sell (0)`, `Hold (1)`, and `Buy (2)`.
  - `step(action)` adjusts balance, shares, and net worth accordingly and returns a reward of `net_worth - initial_balance`.
- **`train_dqn_agent(env)`**:
  - Creates a DQN model with MLP policy from `stable_baselines3`.
  - Trains for 100,000 timesteps.
  - Saves the resulting model to disk as `dqn_stock_trading`.

4 Replacing DQN with PPO

In many cases, **Proximal Policy Optimization (PPO)** can be more robust or sample-efficient than DQN for continuous or more complex action spaces (and sometimes even for discrete spaces). It uses a different approach to update the policy in a clipped fashion, preserving stability. Here is a step-by-step guide on how to replace the current DQN approach with PPO in your code:

4.1 Potential Benefits of PPO over DQN

1. **Policy Gradient Method**: PPO is an on-policy algorithm that directly optimizes a policy. This can be more stable in some environments.

2. **Clipping Mechanism**: PPO uses a clipping in the objective function that prevents large updates from one iteration to the next, making training more stable.

3. **Sample Efficiency**: In certain setups, PPO can reuse on-policy samples effectively, especially if you carefully manage the rollout steps.

4.2 Modifications in the Code

To make the switch, you only need to modify a few lines where DQN is used:

1. **Import PPO**:

```
# From:
from stable_baselines3 import DQN

# To:
from stable_baselines3 import PPO
```

2. **Create the PPO Model Instead of DQN**:

```
def train_ppo_agent(env):
    logging.info("Training PPO Agent...")
    try:
        model = PPO(
            'MlpPolicy',
            env,
            verbose=1,
            learning_rate=3e-4,    # typical default for PPO
            n_steps=2048,         # number of steps to run for each environment
            batch_size=64,
            gamma=0.99,
            gae_lambda=0.95,
            clip_range=0.2,
            ent_coef=0.0,
            tensorboard_log="./ppo_stock_tensorboard/"
        )
        model.learn(total_timesteps=100000)
        model.save("ppo_stock_trading")
        logging.info("PPO Agent trained and saved as 'ppo_stock_trading.zip'.")
        return model
    except Exception as e:
        logging.error(f"Error training PPO Agent: {e}")
        sys.exit(1)
```

3. **Instantiate the PPO Training**:

   - Instead of calling `train_dqn_agent`, call `train_ppo_agent`:

```
logging.info("Initializing and training PPO environment...")
trading_env = StockTradingEnv(data)
trading_env = DummyVecEnv([lambda: trading_env])

# Train
ppo_model = train_ppo_agent(trading_env)
```

Everything else regarding the environment remains the same. The main difference is in how the policy is updated and how the rollout buffer is handled internally.

5 Conclusion

By following the explanations in this document, you should now:

- Understand the structure and purpose of each code block in the LSTM + DQN trading workflow.
- Know how to easily switch from DQN to PPO by changing the import statements and the agent creation and training code.
- Appreciate some of the benefits (and potential pitfalls) of using PPO instead of DQN for reinforcement learning tasks in stock trading environments.

Disclaimer: Reinforcement learning models in algorithmic trading should be carefully tested under realistic constraints, including transaction costs, slippage, and risk management rules. Past performance does not guarantee future results.