­­**Hybrid CNN-Transformer Deep Reinforcement Learning Agent for Cryptocurrency Trading**

1. **Introduction**

This project builds a reinforcement learning (RL) trading agent for cryptocurrencies that combines two sequence processors:

* **Causal 1-D Convolutional Neural Network (CNN):** CNN is used in an unsupervised manner. Acts as a front-end feature extractor that detects short-term patterns, micro-trends and also helps reduce input noise.
* **Transformer Encoder**: A transformer is used as a backbone for the agent. Using self-attention, it models long-term dependencies, enabling the agent to capture trends, patterns and market momentum over extended time windows.

The transformer receives not only the raw time-series input but also additional features extracted by the CNN. This hybrid approach produces a denser, noise-filtered representation of the market state, enhancing the agent’s ability to make informed trading decisions.

1. **Data Preparation**

To build a trading agent we need historical data for training and backtesting. The data will consist of raw candlesticks data (OHLCV) as well as technical indicators. Traders often use indicators from different time frames to make more informed decisions, and we will also use this strategy in our agent. We will follow a day trading strategy, therefore we will use data from an 1-hour and a 1-day time frame and the corresponding technical indicators:

* **Candlestick data (OHLCV)** can be fetched directly from the exchange APIs, for this project we used Binance’s API.
* **Technical indicators** will be calculated using Pandas TA library.

***binance\_scraper.py***

**Arguments:**

***--coin***  - "List of coins to fetch”

***--interval*** - "List of time intervals”

***--start\_time*** - "Start time for the data in ISO format”

***--end\_time*** - "End time for the data in ISO format”

***--config*** - "Path to the YAML indicator config file”

***--save\_folder*** - "Folder to save the scraped data”

This script automates the retrieval of historical OHLCV data from Binance for specified cryptocurrencies and time intervals. It processes the data by calculating customizable technical indicators (e.g., RSI, MACD) defined in the *config\_indicators.yaml* file, structures it into pandas DataFrames with datetime indexing, and saves each dataset as a CSV file in the specified folder.

***config\_indicators.yaml***

Use this file to set configuration for Pandas TA’s indicators, the timeframes and their parameters.

* Each key under "indicators" represents a timeframe (e.g., "1h", "1d").
* Under each timeframe, specify the indicator names and their parameters.
* For indicators of the same type (such as multiple EMAs), include a "kind" field.

A screenshot of a computer program

Description automatically generated

***data\_preparation.ipynb***

This jupyter notebook utilizes the other files to generate the dataset . Running each cell in order will retrieve the data from Binance, calculate technical indicators and save the dataset in .npy files, to be used for training.

**Dataset**

The retrieved data and calculated indicators are comparable to TradingView’s

Examples:

A graph showing the price of bitcoin

Description automatically generated

A graph of stock prices

Description automatically generated with medium confidence

A screenshot of a graph

Description automatically generated

The dataset covers hourly price data for a major cryptocurrency from late 2020 through 2024. This time span includes a variety of market regimes: the bull market of 2021, the bear market of 2022, prolonged sideways movement in 2023, and the renewed uptrend in 2024.

Instead of a simple chronological split (where the last months serve as the validation set), we divide the data into intervals of 5,000 timesteps, using 4,000 steps for training and the subsequent 1,000 for validation. This allows each training/validation pair to sample from different sections of the full market cycle, reducing the risk of regime bias in evaluation.

All input features are normalized using the *Robust Scaler* from scikit-learn. Rather than fitting the scaler to the entire training dataset (which could introduce data leakage and regime dependency), we fit it *locally* within each rolling lookback window.

The agent operates in a simulated environment for **perpetual futures contracts**, instead of spot trading. This setup allows for both long and short positions, as well as leverage.

1. **Trading Environment**

For training and evaluation, we use an open-source cryptocurrency trading environment based on the [Gymnasium](https://gymnasium.farama.org/index.html) interface, originally developed by [Alex K](https://medium.com/coinmonks/deep-reinforcement-learning-for-crypto-trading-72c06bb9b04c). The environment closely simulates perpetual futures trading and provides a realistic testbed for RL-based trading agents.

Environment configuration

A computer screen shot of a program

Description automatically generated

The environment simulates interactions on the exchange. It has three main attributes:

*observation space, action space* and *reward.*

* **Action space**

The agent chooses from four discrete actions at each step:

1. Do nothing
2. Increase position by *order\_size*
3. Decrease position by *order\_size*
4. Close the entire position

(This will change later to simplify the task:

1. Do nothing
2. Close position and open long
3. Close position and open short
4. Close position)



* **Observation space**

Each observation consists of a rolling window (1 week, 168 hours) of features, flattened into a vector of shape (76 x 168,).

* **74 static features:** day, hour, OHLCV, technical indicators
* **2 dynamic features:** available balance, unrealized pnl

A screenshot of a computer program

Description automatically generated

* **Reward function**

In financial trading, the reward function naturally reflects the agent’s realized profit and loss (PnL). We normalize the reward by the initial account balance, ensuring that reward values remain in a meaningful range:

Although the environment is a simplification and not a one-to-one replica of a real exchange, it faithfully captures key trading mechanics, including:

* + Bid/ask spread
  + Opening/closing fees
  + All calculations between exchange account parameters such as: *equity, wallet\_balance, available\_balance, margin, position\_value, unrealized\_pnl,* etc
  + Liquidation process
  + Trading constraints, such as: the agent cannot open a new position if *available\_balance* is lower than *order\_size*, etc

The environment doesn’t consider funding fees and slippage. This is compensated by a fee twice as large as the real one, which the agent pays for opening/closing positions.

**4. Τraining**

During training, episodes begin at randomly sampled timesteps within the training intervals. Each episode lasts for a maximum of *episode\_max\_len* steps (corresponding to 2 weeks, or 336 timesteps), after which the environment is reset. If liquidation occurs before the maximum episode length, the agent receives a significant negative reward, and the episode terminates early.

In reset, the agent is returned to its initial state and the environment first picks a random training internal (4000-step range), then it draws a random start index that leaves room for a whole episode, and the episode rolls out.

During validation, the same logic is applied, but with the *test* interval list.

Some enhancements to increase realism and challenge the agent were implemented:

* **Prioritizing recent data**: More recent timesteps are sampled more frequently during training, focusing learning on the most relevant market regimes.
* **Randomized starting states:** Upon reset, there is a probability that the agent starts with an open long or short position at specific price levels. As training progresses, the likelihood of unfavorable starting conditions increases, encouraging the agent to develop robust policies.
* **Flexible interval-based cross-validation:** Users can define multiple, non-contiguous training and validation intervals within the dataset. This cross-validation approach yields more meaningful evaluation, as it tests agent performance across a variety of market conditions, not just a single time period.

These features ensure that the RL agent is exposed to a diverse set of trading scenarios, supporting more robust learning and fairer evaluation of generalization to unseen market regimes.

1. **Proximal Policy Optimization (PPO)**

We use the Proximal Policy Optimization (PPO) algorithm, a widely used reinforcement learning algorithm. PPO consists of two neural networks operating together in an Actor-Critic fashion:

* **Actor (Policy Network):** Executes the policy, taking actions based on the current observation, aiming to maximize the cumulative reward.
* **Critic (Value Network):** Evaluates the quality of the actor’s actions by estimating the value of the current state or the advantage of the action.

We configure PPO through several hyperparameters defined in *config.py*:

* **Learning Rate (lr)**: Controls how much the network weights adjust at each iteration.
  + **High value**: May cause instability and divergence.
  + **Low value**: Slow convergence or insufficient learning.
* **Gradient Clipping (grad\_clip)**: Limits the magnitude of updates to avoid drastic changes in the policy, ensuring stable training.
  + **High value**: Risk of unstable updates.
  + **Low value**: Excessively slow or restricted learning progress.
* **SGD Iterations (num\_sgd\_iter)**: Number of gradient descent passes through the data batches collected per iteration.
  + **High value**: Potential overfitting.
  + **Low value**: Insufficient model fitting.
* **Training Batch Size (train\_batch\_size)**: Size of the collected batch data from all workers used per gradient update.
  + **Small batch**: Noisy gradient estimates, but more frequent updates.
  + **Large batch**: More stable gradient estimates, but computationally intensive.
* **Discount Factor (gamma)**: Determines the importance of future rewards.
  + **High γ**: Prioritizes long-term rewards.
  + **Low γ**: Prioritizes immediate rewards.
* **Entropy Coefficient (entropy\_coeff)**: Encourages exploration by rewarding the agent for randomness in action selection.
  + **High value**: Promotes more exploration.
  + **Low value**: Promotes exploitation (focused on the current policy).
* **KL Divergence Coefficient (kl\_coeff)**: Controls how heavily deviations between the old and updated policy (measured by KL divergence) penalize the updates, ensuring policy stability.
  + **High value**: Restrictive, small policy updates.
  + **Low value**: Larger updates, risk of instability.
* **KL Divergence Target (kl\_target)**: The ideal KL divergence between old and new policies. The PPO algorithm dynamically adjusts *kl\_coeff* to keep the policy updates close to this target.
  + **Low target**: Conservative policy updates.
  + **High target**: Permits larger policy shifts.
* **Probability Clipping (clip\_param)**: Caps the ratio of probabilities between new and old policies, preventing excessively large updates that might harm training stability.

1. **Transformer-only training**

We use PyTorch and Ray to build a custom Transformer model as the backbone for our reinforcement learning agent. The presented architecture is pretty straightforward; however, it can easily accommodate more advanced Transformer variations.

**Model Components** (*simple\_transformer.py*):

1. **Input Processing:**

Embeds raw observations into a learned vector space representation and adds positional encodings to preserve sequence order.

1. **Transformer Model:**

Consists of multiple stacked encoder layers utilizing PyTorch’s pre-built multi-head self-attention mechanism. It implements normalization layers, dropout regularization, and the GELU activation function.

1. **Output Heads:**

Extracts the dynamic environment features (wallet balance and unrealized PnL) from the most recent timestep. Then, it combines these dynamic features with the transformer’s final timestep output embedding to form an enriched representation and feed it to two separate networks, the policy (actor) and the value (critic) networks. Both networks share an identical structure:

**Linear(embed\_size + 2 → 256) → LayerNorm → GELU → Linear(256 → action\_space\_size)**

**Runs**

**Run 1#:**

Training time: 20.84 hours

config:

A screenshot of a computer program

Description automatically generated

A graph showing a graph

Description automatically generated with medium confidence

A graph showing the time of a stock market

Description automatically generatedA graph showing a wave of data

Description automatically generated with medium confidenceA graph showing the growth of the stock market

Description automatically generatedA graph showing a wave of data

Description automatically generated with medium confidenceA graph showing a graph

Description automatically generated with medium confidence