Hybrid CNN-Transformer Deep Reinforcement Learning Agent for Cryptocurrency Trading

Boulionis Athanasios – AI 189 7/6/2025

Abstract

This project explores the design and evaluation of a hybrid reinforcement learning (RL) agent for cryptocurrency trading, combining a Causal 1-D Convolutional Neural Network (CNN) for short-term feature extraction with a transformer encoder to capture long-term dependencies. The agent is trained and evaluated on a multi-year dataset of hourly and daily market data, using a custom environment simulating perpetual futures trading with cross-validation across different market regimes.

Despite extensive experimentation with model architectures, reward functions, and training protocols, the agent did not achieve consistent profitability or stable policies. The main challenges encountered included training instability, reward scaling issues, and the tendency of the agent to converge on trivial strategies such as "doing nothing." These outcomes underscore the real-world difficulty of applying deep RL in noisy, non-stationary financial environments.

Nevertheless, the project provided valuable insights into environment design, preprocessing, and model selection for RL-based trading systems. The findings suggest several promising avenues for future research, including modular transformer architectures, improved reward shaping, and longer training horizons. While the agent's performance did not meet expectations, the experience gained lays the groundwork for further experimentation and refinement.

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.

2. 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.

```yaml
# Example config YAML file
indicators:
  # Example for 1-hour timeframe
  "1h":
    # Indicator: Average True Range (ATR)
    atr:
      length: 14       # Period for ATR calculation
    # Indicator: Relative Strength Index (RSI)
    rsi:
      length: 14       # Period for RSI calculation

  # Example for 1-day timeframe
  "1d":
    # Multiple indicators of the same type require a "kind" field.
    # Exponential Moving Averages (EMA)
    ema_200:
      kind: "ema"      # Specify the type of indicator
      length: 200      # Period for the 200 EMA
    ema_50:
      kind: "ema"      # Specify the type of indicator
      length: 50       # Period for the 50 EMA
```

data_preparation.ipynb

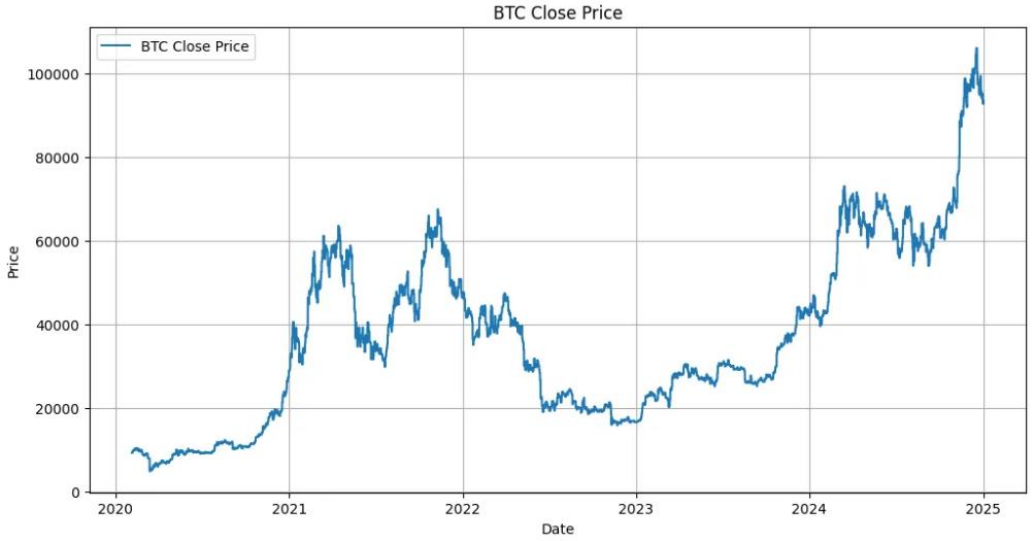
This jupyter notebook utilizes the other files to generate the dataset . Running each cell in order will retrieve the data from Binance, calculate technical indicators and save the dataset in .npy files, to be used for training.

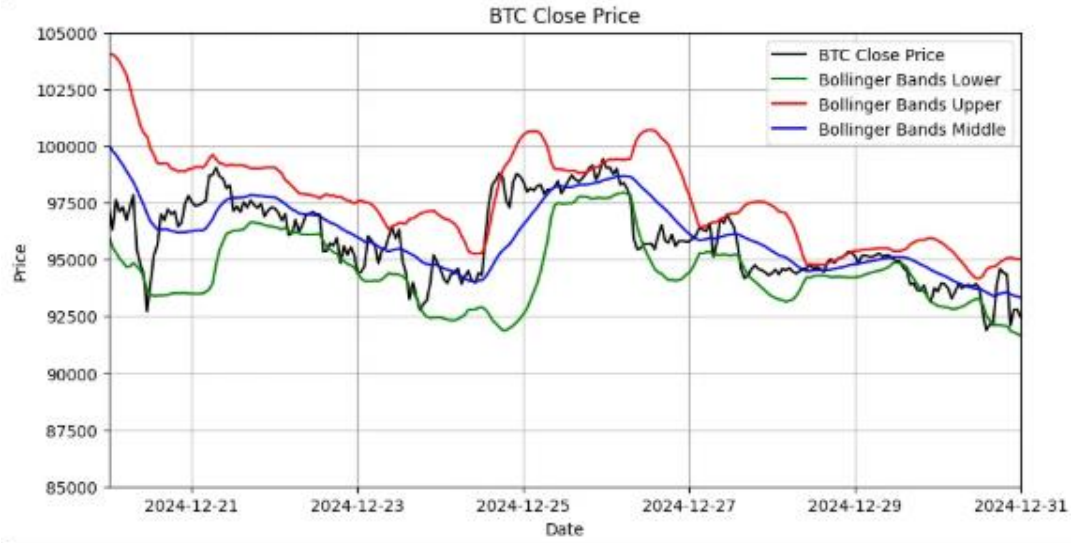
Dataset

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

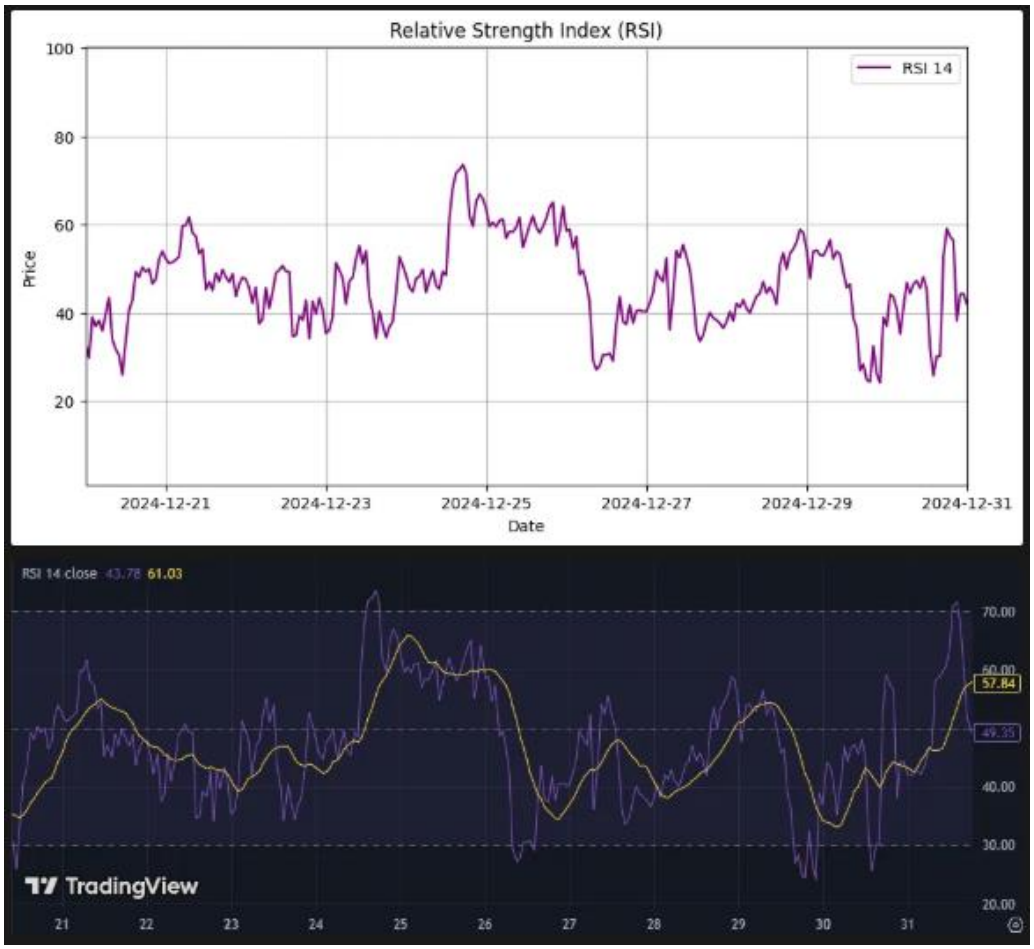
---

Examples:


BTC Close Price
BTC Close Price
100000
80000
60000
40000
20000
0
Price
2020
2021
2022
2023
2024
2025
Date


BTC Close Price
BTC Close Price
Bollinger Bands Lower
Bollinger Bands Upper
Bollinger Bands Middle
105000
102500
100000
97500
95000
92500
90000
87500
85000
Price
2024-12-21
2024-12-23
2024-12-25
2024-12-27
2024-12-29
2024-12-31
Date


100,000.00
98,000.00
96,000.00
95,039.12
94,000.00
93,413.84
92,784.75
91,788.55
TradingView
21
22
23
24
25
26
27
28
29
30



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.

3. Trading Environment

For training and evaluation, we use an open-source cryptocurrency trading environment based on the Gymnasium interface, originally developed by Alex K. The environment closely simulates perpetual futures trading and provides a realistic testbed for RL-based trading agents.

Environment configuration

```
  env_config={
      "dataset_name": "dataset",  # .npy files should be in ./data/dataset/
      "leverage": 2, # leverage for perpetual futures
      "episode_max_len": 168 * 2, # train episode length, 2 weeks
      "lookback_window_len": 168, # 1week
      "train_start": [2000, 7000, 12000, 17000, 22000],
      "train_end": [6000, 11000, 16000, 21000, 26000],
      "test_start": [6000, 11000, 16000, 21000, 26000],
      "test_end": [7000, 12000, 17000, 22000, 29377-1],
      "order_size": 50, # dollars
      "initial_capital": 1000, # dollars
      "open_fee": 0.12e-2, # taker_fee
      "close_fee": 0.12e-2, # taker_fee
      "maintenance_margin_percentage": 0.012, # 1.2 percent
      "initial_random_allocated": 0, # opened initial random long/short position up to in
itial_random_allocated $
      "regime": "training",
      "record_stats": False, # True for backtesting
  },
```

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)

```
action_space = gymnasium.spaces.Discrete(4)
```

- **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

```
observation_space=gymnasium.spaces.Box(
  low=-np.inf,
  high=np.inf,
  shape=(76 * 168,),
  dtype=np.float32
)
```

- **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:

$$reward = \frac{pnl\_realised\_short + pnl\_realised\_long}{initial\_balance}$$

  After experimentation, we will change this reward into a more fitting one that accounts for unrealized PnL as well (otherwise the agent could just buy and hold forever):

  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. Training

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.

5. 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.

6. 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.
2. **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.
3. **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

In each training run, screenshots of the TensorBoard metrics were taken. Due to the large number of images, they are not included in this document to keep it concise. However, the full collection can be found in the *tensorboard_screenshots/* directory.

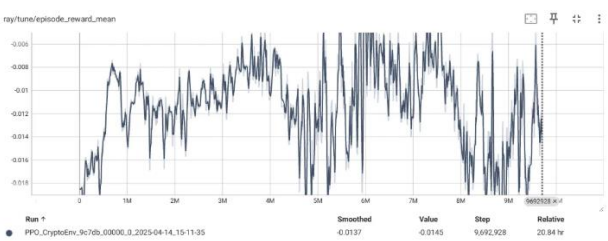
Some screenshots are presented to highlight relevant issues.

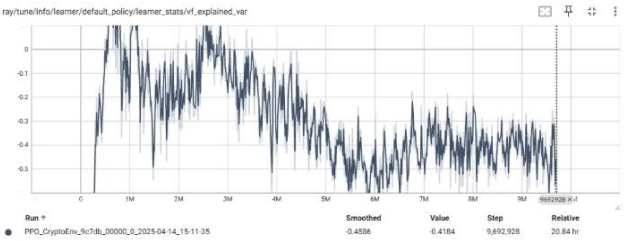
Run #1:

Training time: 20.84 hours

config:

```
.training(
        lr=5e-5,
        gamma=0.995, # 1.
        grad_clip=30.,
        entropy_coeff=0.03,
        kl_coeff=0.05,
        kl_target=0.01, # not used if kl_coeff == 0.
        num_sgd_iter=10,
        use_gae=True,
        # lambda=0.95,
        clip_param=0.3, # larger values for more policy change
        vf_clip_param=10,
        train_batch_size=8 * 6 * 168, # num_rollout_workers * num_envs_per_worker * rollo
ut_fragment_length
        shuffle_sequences=True,
```





Results:

The initial run shows substantial instability during training:

1. **Mean episodic reward:** It initially climbs, but after 4m steps it becomes really unstable.
2. **Value function:** Explained variance of the value function remains negative throughout training. This means that the critic's predictions are worse than a naïve baseline.

Run #2:

Training time: 17.61 hours

Adjustments in PPO parameters:

1. ***lr_schedule = [[0, 1e-4], [2e6, 5e-5], [4e6, 1e-5]] :*** Learning rate might be too high, causing unstable updates. We will use a scheduler to gradually reduce learning rate during training. This way we can have a large learning rate initially to explore, and later reduce it for more stable learning.
2. ***kl_coeff=0.05 → 0:*** Disable KL divergence. It might be causing conficts with *clip_param* when used at the same time.
3. **clip_param=0.3 → 0.2:** Reducing the clip range tightens policy updates, helping prevent large, destabilizing changes between updates.

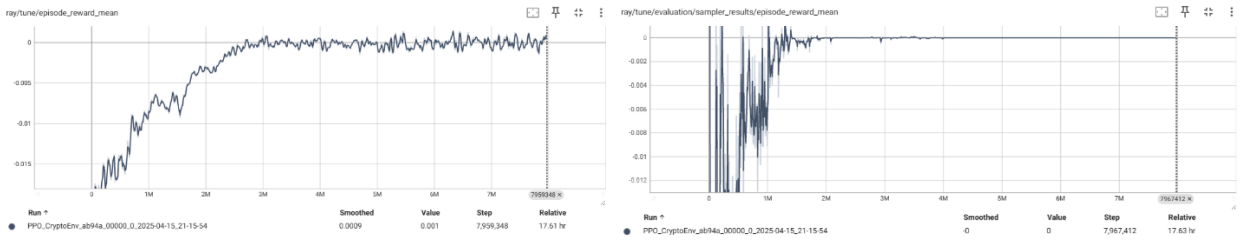
Adjustments in transformer model:

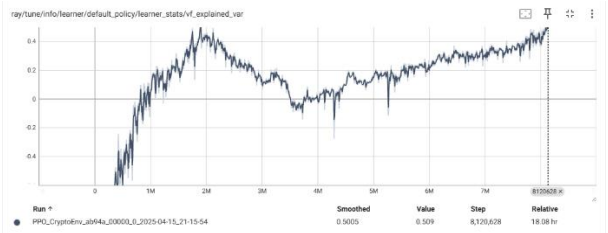
1. ***dropout=0.3:*** Increase dropout in the transformer to reduce overfitting.
2. ***LayerNorm:*** Add normalization layers in the transformer and in both the policy and value heads.
3. Wider layers in policy and value neural networks (64 → 256)

Config:

```
    .training(
        lr_schedule=[
            [0, 1e-4],
            [2e5, 1e-5],
            [5e5, 1e-6]],
        gamma=0.995, # 1.
        grad_clip=30.,
        entropy_coeff=0.03,
        kl_coeff=0,
        kl_target=0.01, # not used if kl_coeff == 0.
        num_sgd_iter=10,
        use_gae=True,
        # lambda=0.95,
        clip_param=0.2, # larger values for more policy change
        vf_clip_param=10,
        train_batch_size=8 * 6 * 168, # num_rollout_workers * num_envs_per_worker * rollo
ut_fragment_length
        shuffle_sequences=True,
        model={
            "custom_model": "SimpleTransformer",
            "custom_model_config": {
                "embed_size": 128,
                "nhead": 4,
                "nlayers": 3,
                "seq_len": 168,
                "dropout": 0.3,
            }
        }
    )
```

Results:





The model shows significant improvement in training stability, making the agent learn a policy. The explained variance of the value function rises to ~0.5 during early training (up to 2m steps), and then a sharp reduction happens followed by a gradual recovery at a reduced rate. This phenomenon might be caused by the learning rate reduction since it coincides with the scheduled learning rate reduction at 2m steps.

During evaluation, the agent converges on a “do nothing” strategy to avoid losses. While this reflects a riskless policy, it has no use because we want a profitable policy.

Run #3:

Training time: 23.94 hours

Adjustments in PPO algorithm:

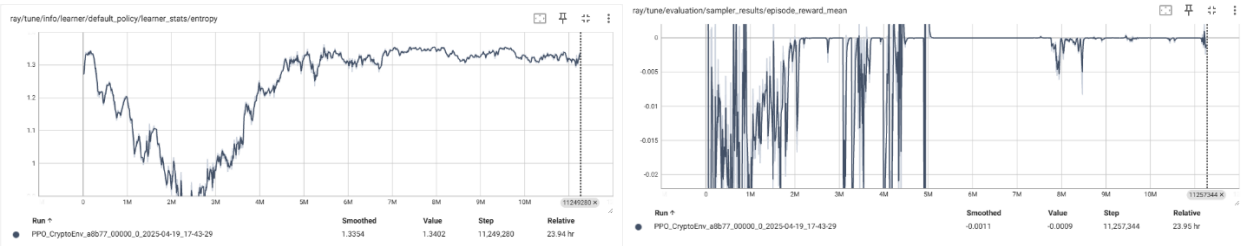
1. ***lr_schedule=[[0, 1e-4], [2e6, 5e-5], [4e6, 1e-5]]:*** A learning rate of 1e-6 might be way too low, slowing parameter updates and causing the agent to not leverage much from explorative actions. We will use a smoother and longer schedule.
2. ***entropy_coeff=0.03 → 0.05****:* The model abuses on action 0 (”do nothing”), probably due to insufficient exploration. Increasing the entropy regularization term would encourage exploration, preventing convergence to local minima (like action 0).

Adjustments in transformer model:
Adding dropout layers to the policy and value network heads to mitigate overfitting.

Config:

```
.training(
        lr_schedule=[
            [0, 1e-4],
            [3e6, 5e-5],
            [6e6, 1e-5]],
        gamma=0.995, # 1.
        grad_clip=30.,
        entropy_coeff=0.05,
        kl_coeff=0,
        kl_target=0.01, # not used if kl_coeff == 0.
        num_sgd_iter=10,
        use_gae=True,
        # lambda=0.95,
        clip_param=0.2, # larger values for more policy change
        vf_clip_param=10,
        train_batch_size=8 * 6 * 168, # num_rollout_workers * num_envs_per_worker * rollo
ut_fragment_length
        shuffle_sequences=True,
        model={
            "custom_model": "SimpleTransformer",
            "custom_model_config": {
                "embed_size": 128,
                "nhead": 4,
                "nlayers": 3,
                "seq_len": 168,
                "dropout": 0.3,
            }
        }
    )
```

Results:



The agent again, converges into the “do nothing” strategy, this time though the whole training seems to be way too unstable compared to the previous run.

Run #4:

Adjustments in PPO algorithm:

1. ***learning rate → 1e-5:*** Sudden drops in learning rate might be affecting the stabilization of the training process. We’ll remove the scheduler and keep a constant value.
2. ***entropy_coeff = 0.05 → 0.03:*** We’ll set the value back to 0.03 because this much exploration might be causing instabilities. We’ll make up for it by adding complexity in the actor-critic layers in the transformer.
3. ***dropout = 0.3 → 0.2:*** Reducing the dropout a bit because it might be causing under-fitting.

Adjustments in transformer algorithm:

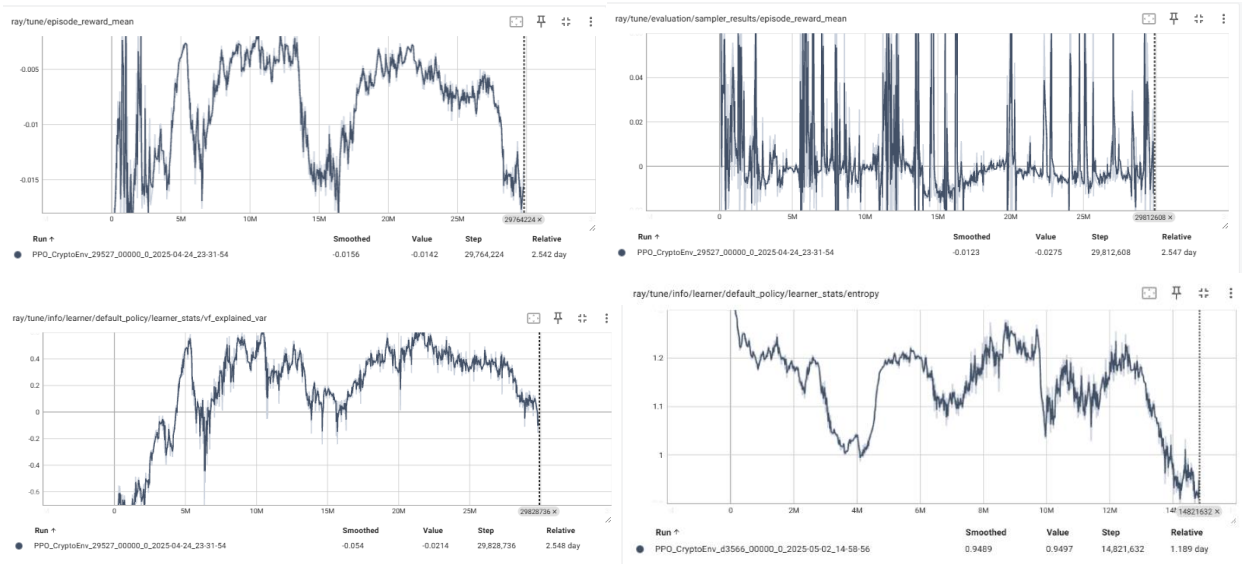
We’ll increase the complexity of the policy and value heads in the transformer by adding two additional hidden layers:

```
# Policy Head
self.policy_head = nn.Sequential(
    nn.Linear(self.embed_size + 2, 512),
    nn.LayerNorm(512),
    nn.GELU(),
    nn.Dropout(0.2),  # Dropout after activation
    nn.Linear(512, 256),
    nn.LayerNorm(256),
    nn.GELU(),
    nn.Dropout(0.1),
    nn.Linear(256, 64),
    nn.LayerNorm(64),
    nn.GELU(),
    nn.Linear(64, num_outputs)
)

# Value Head
self.value_head = nn.Sequential(
    nn.Linear(self.embed_size + 2, 512),
    nn.LayerNorm(512),
    nn.GELU(),
    nn.Dropout(0.2),
    nn.Linear(512, 256),
    nn.LayerNorm(256),
    nn.GELU(),
    nn.Dropout(0.1),
    nn.Linear(256, 64),
    nn.LayerNorm(64),
    nn.GELU(),
    nn.Linear(64, 1)
)
```

Config:

```
.training(
        lr=1e-5,
        gamma=0.995, # 1.
        grad_clip=30.,
        entropy_coeff=0.03,
        kl_coeff=0,
        kl_target=0.01, # not used if kl_coeff == 0.
        num_sgd_iter=10,
        use_gae=True,
        # lambda=0.95,
        clip_param=0.2, # larger values for more policy change
        vf_clip_param=10,
        train_batch_size=8 * 6 * 168, # num_rollout_workers * num_envs_per_worker * rollo
        shuffle_sequences=True,
        model={
            "custom_model": "SimpleTransformer",
            "custom_model_config": {
                "embed_size": 128,
                "nhead": 4,
                "nlayers": 3,
                "seq_len": 168,
                "dropout": 0.2,
            }
        }
    )
```

Results:



The training remains quite unstable.

Policy is overall losing money; learning doesn't seem to improve returns.

In evaluation, the baseline looks to be on the negative with tall spikes, which are probably caused by lucky episodes.

Value function explained variance shows that the critic initially learns but then stops tracking returns, advantage estimation become worse.

Run #5:

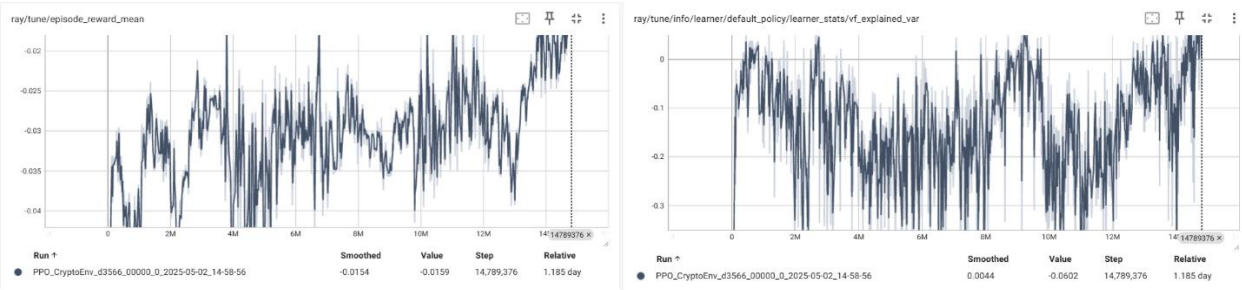
Training time: 1.185 days

Adjustment in PPO algorithm:

1. ***learning rate → 3e-4 - 1e-5***: We'll try to significantly increase it initially, with a linear decay using scheduler.
2. ***kl_coeff/kl_target = 0/0 → 0.1 / 0.01:*** **Gradient clipping** acts as a limit on large gradients and scales them down, while **KL divergence** is computed inside the loss as an extra term and adds a penalty to large updates between two consecutive states. So, my initial idea was wrong, I'll try to bring back KL divergence starting at kl_coeff = 0.2.
3. ***entropy_coeff = 0.03 → 0.05 - 0.01:*** I'll keep the exploration at high values initially and gradually decrease it to let the policy concentrate and avoid the entropy "bounce" from the last run.
4. ***vf_clip_param = 10 → 5***: I'll set stricter clipping in the value network to prevent outlandish returns due to "lucky" runs that could be dragging the explained variance down.

Adjustment in transformer algorithm:
Policy/Value Dropout = 0.2 → 0.1 only in the first layer; Reduce noise.

Results:



At this point, the goal is to let the model fully converge to its optimal policy under the current reward function, which is to be *holding or taking no action*. Then we can implement a different reward function to try and penalize this behaviour.

It seems like the scheduler interval is too large and we didn't even reach the steps limit. I'll just remove it all together, it doesn't seem to help much.

Run #6:

Training time: 20.19 hours

Due to time limitations I have to reduce the complexity of my model to train faster.

Adjustments in PPO algorithm:

1. *learning rate = scheduler → constant 5e-5*

2. *entropy_coeff= 0.05 - 0.01 → 0.03*
3. *vf_clip_param= 5 → 10*
4. *kl_coeff= 0.1 → 0.05*

Adjustments in transformer algorithm:

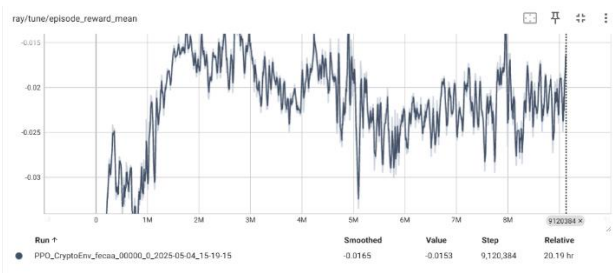
Due to time limits I have to remove the extra hidden layers from the policy and value networks to train faster.

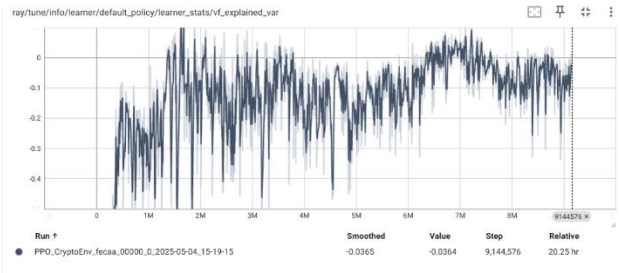
```
# ---------- Policy and value networks ------------------
self.policy_head = nn.Sequential(
    nn.Linear(self.embed_size + 2, 256),
    nn.LayerNorm(256),
    nn.GELU(),
    nn.Dropout(0.1),
    nn.Linear(256, num_outputs)
)

self.value_head = nn.Sequential(
    nn.Linear(self.embed_size + 2, 256),
    nn.LayerNorm(256),
    nn.GELU(),
    nn.Dropout(0.1),
    nn.Linear(256, 1)
)
```

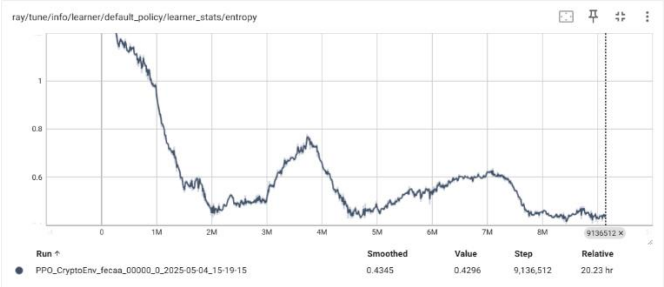
Config:

```
  .training(
        lr=5e-5,
        gamma=0.995, # 1.
        grad_clip=30,
        entropy_coeff=0.03,
        kl_coeff=0.05,
        kl_target=0.01, # not used if kl_coeff == 0.
        num_sgd_iter=10,
        use_gae=True,
        # lambda=0.95,
        clip_param=0.2, # larger values for more policy change
        vf_clip_param=10,
        train_batch_size=8 * 6 * 168, # num_rollout_workers * num_envs_per_worker * rollo
        shuffle_sequences=True,
        model={
            "custom_model": "SimpleTransformer",
            "custom_model_config": {
                "embed_size": 128,
                "nhead": 4,
                "nlayers": 3,
                "seq_len": 168,
                "dropout": 0.2
            }
        }
    )
```

Results:







Run #7:

Training time: 14.8 hours

Adjustments in PPO algorithm:

1. ***learning rate = 1e-5 → 3e-4:*** This value might be extremely low for the batch size (8k) so the agent doesn't get past random wandering.
2. ***entropy_coeff = 0.03 → 0.01:*** Reduce randomness to help the agent settle at a policy.
3. ***vf_clip_param = 10 → 5:*** Too loose, causing critic instability (vf_explained_var)

Adjustments in train enviroment:

Reward:

Current reward:

$$reward = \frac{realized\_pnl\_short + realized\_pnl\_long}{initial\_balance}$$

The agent only gets reward when it closes a position. This causes the agent to converge into policies where it never trades at all and open positions and hold forever.

We want a reward function that:

- Encourages profitable open and close decisions
- Penalizes holding through losses
- Still makes "doing nothing" neutral, not dominant

Modified reward:

$$realized = reward_{pnl_{short}} + reward\_pnl\_long$$

$$equity\_delta = next\_equity\ - current\_equity$$

$$reward = \frac{(1 - coeff) \cdot realized + coeff \cdot equity\_delta}{initial\_balance}$$

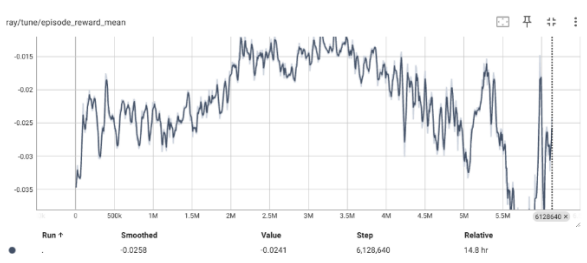
This reward function satisfies the desired conditions.

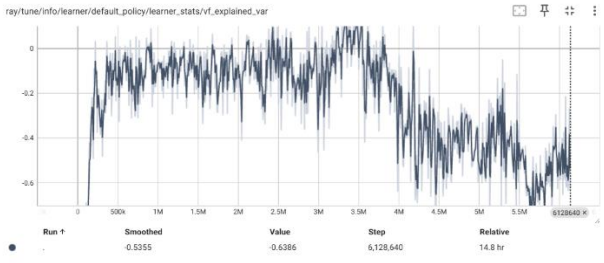
1. Partial reward while holding winning positions
2. Penalty while holding losing positions
3. Full reward when closing profitable trades
4. Full penalty when closing losing trades
5. Neutral reward when nothing happens

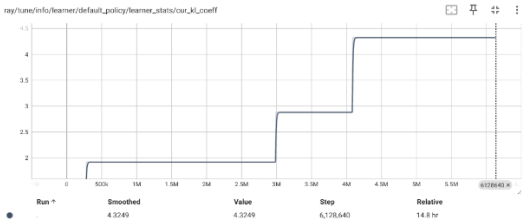
Config:

```
.training(
        lr=3e-4,
        gamma=0.995, # 1.
        grad_clip=30,
        entropy_coeff=0.01,
        kl_coeff=0.05,
        kl_target=0.01, # not used if kl_coeff == 0.
        num_sgd_iter=10,
        use_gae=True,
        # lambda=0.95,
        clip_param=0.2, # larger values for more policy change
        vf_clip_param=5,
        train_batch_size=8 * 6 * 168, # num_rollout_workers * num_envs_per_worker * rollo
ut_fragment_length * multiplier
        shuffle_sequences=True,
        model={
            "custom_model": "SimpleTransformer",
            "custom_model_config": {
                "embed_size": 128,
                "nhead": 4,
                "nlayers": 3,
                "seq_len": 168,
                "dropout": 0.1,
                "cnn_enabled": False,
                "freeze_cnn": False,
            }
        }
    )
```

Results:







Training remains unstable. The KL coefficient spikes to 4, causing significant instabilities. We should probably remove kl divergence as we did initially.

Run #8:

Training time: 17.2 hours

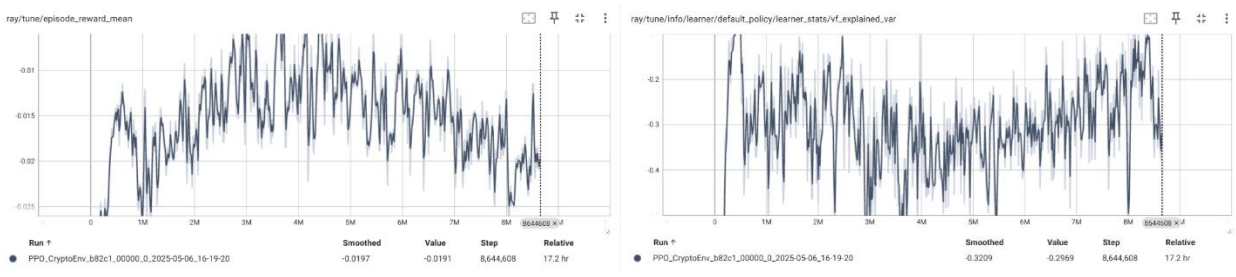
Adjustments in PPO algorithm:

1. *learning rate = 3e-4 → 1e-4*
2. *kl_coeff = 0.01 → 0*
3. *train_batch_size = 8k → 16k*
4. *entropy_coeff = 0.01 → 0.003*
5. *grad_clip = 30 → 2*

Config:

```
.training(
        lr=1e-4,
        gamma=0.995, # 1.
        grad_clip=2,
        entropy_coeff=0.003,
        kl_coeff=0,
        kl_target=0.01, # not used if kl_coeff == 0.
        num_sgd_iter=10,
        use_gae=True,
        # lambda=0.95,
        clip_param=0.2, # larger values for more policy change
        vf_clip_param=5,
        train_batch_size=8 * 6 * 168* 2, # num_rollout_workers * num_envs_per_worker * ro
llout_fragment_length * multiplier
        shuffle_sequences=True,
        model={
            "custom_model": "SimpleTransformer",
            "custom_model_config": {
                "embed_size": 128,
                "nhead": 4,
                "nlayers": 3,
                "seq_len": 168,
                "dropout": 0.1,
                "cnn_enabled": False,
                "freeze_cnn": False,
            }
        }
    )
```

Results:



Significantly worse results.

Run #9:

Adjustments in train environment:

Reward function:

We discovered a bug in the current reward function. By using equity as the reward, it counts twice the realized PnL.

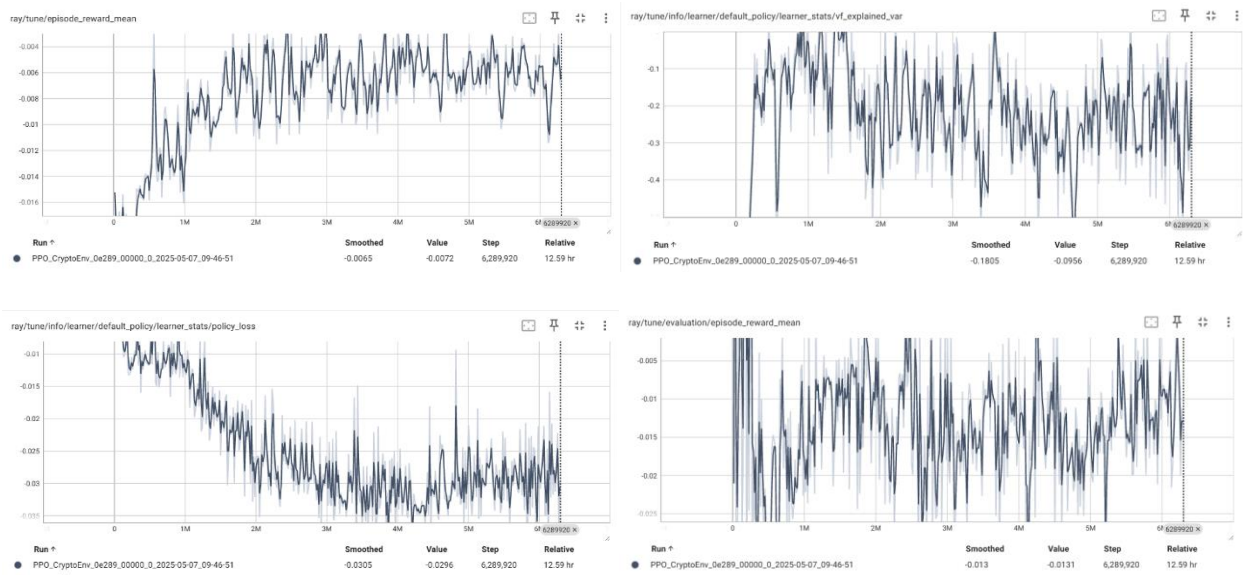
$$\Delta e_t = e_t - e_{t-1} = (B_t + u_t) - (B_{t-1} + u_{t-1}) = (B_{t-1} + r_t + u_t) - (B_{t-1} + u_{t-1})$$
$$= r_t + (u_t - u_{t-1})$$

So, when a position is closed, the *realized PnL* is counted twice. To address this, we will use the *unrealized PnL* instead.

$$reward = \frac{(1 - coeff) \cdot realized + coeff \cdot unrealized\_pnl}{initial\_balance}$$

Config:

Results:



- **vf explained variance:** Oscillates between -0.5 and -0.1 uncontrollably.
- **policy entropy:** High → Agent random walking.
- **episode mean reward:** Still losing but improved from the previous run.
- **evaluation reward:** Random moves.

Run #10:

Training time: 5.77 hours

Adjustments in PPO algorithm:

1. **entropy_coeff = 0.003 → [0.001 → 0.0001 after 2m steps]:** Reduce randomness; let entropy fall.
2. **vf_clip_param= 5 → 3:** Tighter critic updates to stabilize explained variance.

Results:

- No improvements

Run #11:

Training time: 8.88 hours

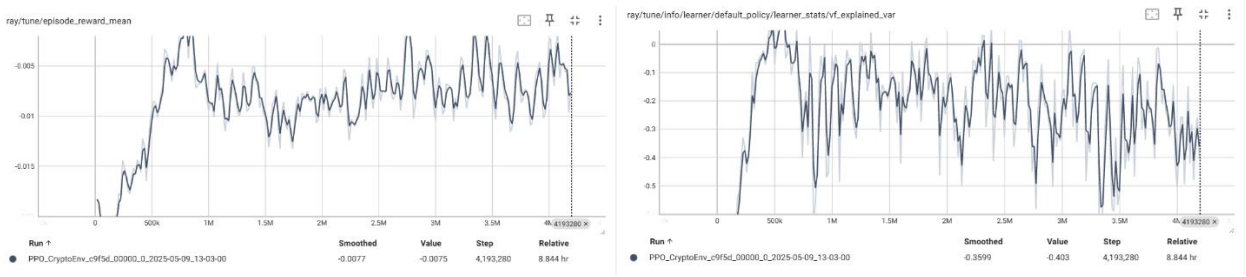
Adjustments in PPO algorithm:

1. **entropy_coeff = [0.001 → 0.0001 after 2m steps] → 0.0002**
2. **vf_clip_param = 3 → 1**

Config:

```
    .training(
        lr=1e-4,
        gamma=0.995, # 1.
        grad_clip=2,
        entropy_coeff=0.0002,
        kl_coeff=0,
        kl_target=0.01, # not used if kl_coeff == 0.
        num_sgd_iter=10,
        use_gae=True,
        # lambda=0.95,
        clip_param=0.2, # larger values for more policy change
        vf_clip_param=1,
        train_batch_size=8 * 6 * 168* 2, # num_rollout_workers * num_envs_per_worker * ro
llout_fragment_length * multiplier
        shuffle_sequences=True,
        model={
            "custom_model": "SimpleTransformer",
            "custom_model_config": {
                "embed_size": 128,
                "nhead": 4,
                "nlayers": 3,
                "seq_len": 168,
                "dropout": 0.1,
                "cnn_enabled": False,
                "freeze_cnn": False,
            }
        }
    )
```

Results:



Run #12:

Adjustments in train environment:

So far, We've been using the following action configuration:

1. do nothing
2. buy an *order_size* amount of coins
3. sell an *order_size* amount of coins
4. sell everything

This task is probably too complex for such a simple model that I'm trying to use for a project, and with the limited time I have I'll transition to a simpler action configuration that uses the whole available balance for its orders, like:

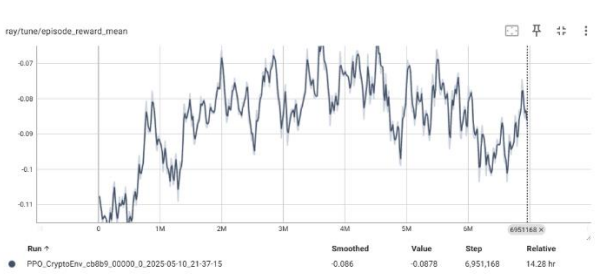
1. do nothing
2. sell & open a long position (with all available balance)
3. sell & open a short position (with all available balance)
4. sell everything

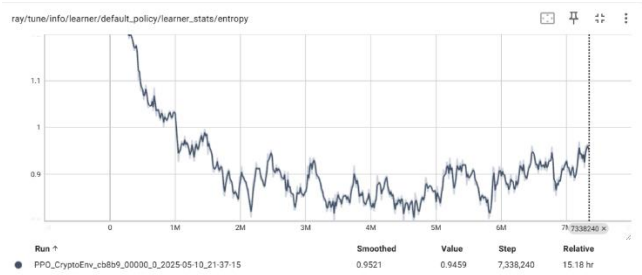
Let's give it a try.

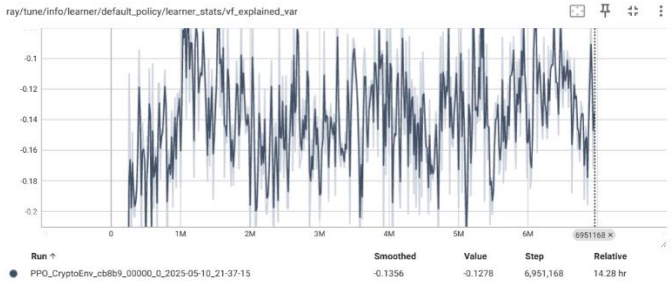
Config:

```
.training(
        lr=1e-4,
        gamma=0.995, # 1.
        grad_clip=2,
        entropy_coeff=0.0003,
        kl_coeff=0,
        kl_target=0.01, # not used if kl_coeff == 0.
        num_sgd_iter=10,
        use_gae=True,
        # lambda=0.95,
        clip_param=0.2, # larger values for more policy change
        vf_clip_param=1,
        train_batch_size=8 * 6 * 168* 2, # num_rollout_workers * num_envs_per_worker * ro
llout_fragment_length * multiplier
        shuffle_sequences=True,
        model={
            "custom_model": "SimpleTransformer",
            "custom_model_config": {
                "embed_size": 128,
                "nhead": 4,
                "nlayers": 3,
                "seq_len": 168,
                "dropout": 0.1,
                "cnn_enabled": False,
                "freeze_cnn": False,
            }
        }
    )
```

Results:







- Entropy still high ~0.95, actor is still near-uniform
- Explained variance still negative, critic still can't fit

Run #13:

Training time: 7.84

Adjustments in train environment:

*Rreward function**:***

Position after FULL-LONG is 1000$, but in a single one hour step, BTC typically moves for around 0.2% - 0.8%. Rewards are on the scale of 10^{-3} per step, and returning gradients are probably getting drowned in noise. If the critic can't move its weights, it can never fit the returns.

To address vanishing gradients due to very small reward magnitudes, the reward signal is scaled:

$$reward = reward\ x\ 100$$

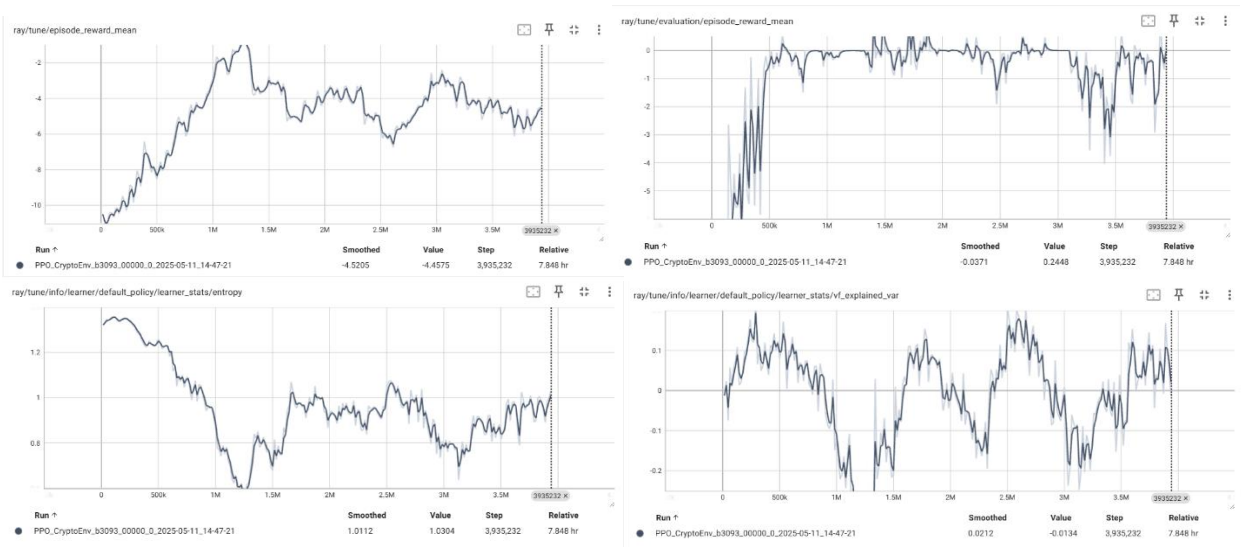
While RLlib offers built-in reward normalization, it's not fully supported on Windows. A manual scaling factor is therefore applied to amplify the reward signal and facilitate more effective learning.

```
        realized        = self.reward_realized_pnl_long + self.reward_realized_pnl_short
        equity_delta    = next_equity - self.equity
        unrealised_delta = equity_delta - realized     # remove realised part
        reward = ((1 - self.reward_coeff) * realized + self.reward_coeff * unrealised_del
ta) / self.initial_balance

        reward = reward * 100 # <--- New Line (Scaling)
```

Config:

```
.training(
        lr=1e-4,
        gamma=0.995, # 1.
        grad_clip=2,
        entropy_coeff=1e-4,
        kl_coeff=0,
        kl_target=0.01, # not used if kl_coeff == 0.
        num_sgd_iter=10,
        use_gae=True,
        # lambda=0.95,
        clip_param=0.2, # larger values for more policy change
        vf_clip_param=1,
        vf_loss_coeff=0.5,
        train_batch_size=8 * 6 * 168* 2, # num_rollout_workers * num_envs_per_worker * ro
llout_fragment_length * multiplier
        shuffle_sequences=True,
        model={
            "custom_model": "SimpleTransformer",
            "custom_model_config": {
                "embed_size": 128,
                "nhead": 4,
                "nlayers": 3,
                "seq_len": 168,
                "dropout": 0.1,
                "cnn_enabled": False,
                "freeze_cnn": False,
            }
        }
    )
```

Results:



Reward scaling led to noticeable improvements.

- ***vf_explained_variance*:** Now oscillates between -0.2 and 0.15, occasionally entering positive territory. This indicates that the critic is starting to learn, though targets remain noisy.
- ***policy_entropy*:** Initially entropy falls to 0.7 but the rises back to around 1. Actor briefly becomes more deterministic, than returns to randomness, something is shaking the policy weights.
- ***mean_episode_reward*:** Shows signs of increasing as learning progresses.

Overall, progress is evident, but persistent noise in critic updates may be destabilizing both the critic and the actor. Further work may be needed to reduce target noise and stabilize learning.

Run #14:

Training time: 8.57 hours

Adjustments in PPO algorithm:

1. ***vf_loss_coeff = 0.5 → 0.25:***

$$L_{PPO} = L_{policy} + c_v L_{value} - \beta H(\pi)$$

Where:

- L_{policy} policy loss - drives the actor
- L_{value} value loss - trains the critic
- $H(\pi)$ entropy term - encourages exploration

The transformer backbone is shared by the actor and the critic, and their gradients back-propagate through it.

Value loss coefficient scales how much the critic affects the returning gradients, meaning how much it influences the changes made in the models' weights. By lowering this coefficient, critic has less of an impact in the weight updates.

Adjustments in transformer algorithm:

1. ***Detaching the value head from the transformer backbone:***

By detaching the value head from the transformer backbone, the critic still gets features from the transformer, but its gradients stop there and do not propagate though the transformer layers.

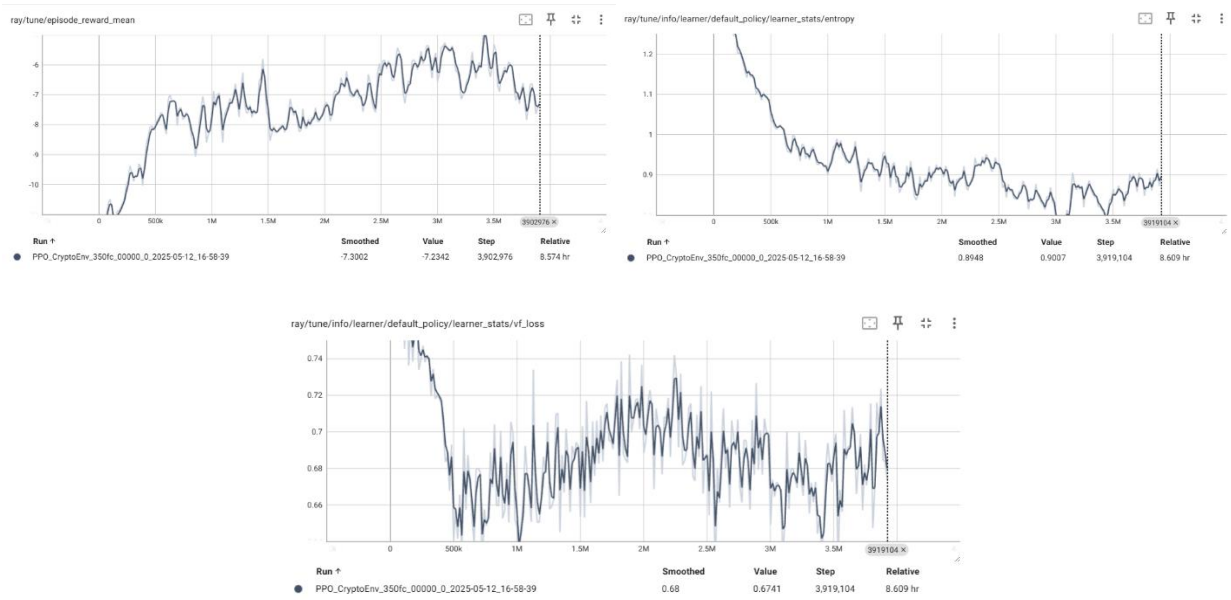
- Policy head still back-propagates thought the transformer → it can update shared weights.
- Value head learns only by adjusting its own parameters → its noisy gradients don't shake the shared backbone.

```
value_in = last_out.detach()
        self.values_out = self.value_head(torch.cat((value_in, dynamic_features.detach
()), dim=1)).squeeze(1)
```

Config:

```
.training(
        lr=1e-4,
        gamma=0.995, # 1.
        grad_clip=2,
        entropy_coeff=5e-5,
        kl_coeff=0,
        kl_target=0.01, # not used if kl_coeff == 0.
        num_sgd_iter=10,
        use_gae=True,
        # lambda=0.95,
        clip_param=0.2, # larger values for more policy change
        vf_clip_param=1,
        vf_loss_coeff=0.25, #
        train_batch_size=8 * 6 * 168* 2, # num_rollout_workers * num_envs_per_worker * ro
llout_fragment_length * multiplier
        shuffle_sequences=True,
        model={
            "custom_model": "SimpleTransformer",
            "custom_model_config": {
                "embed_size": 128,
                "nhead": 4,
                "nlayers": 3,
                "seq_len": 168,
                "dropout": 0.1,
                "cnn_enabled": False,
                "freeze_cnn": False,
            }
        }
    )
```

Results:



No particular improvement other than, the entropy seems to have stabilized.

Run #15:

Training time: 7.98 hours

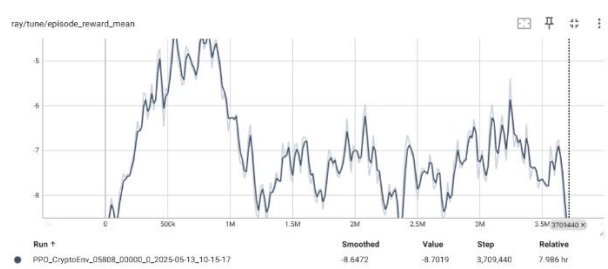
Adjustments in transformer algorithm:

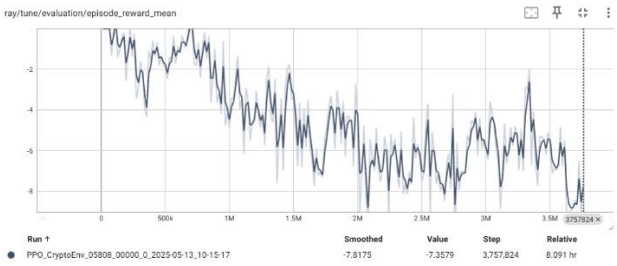
Reattaching critic but leaving *vf_loss_coeff* at 0.25:

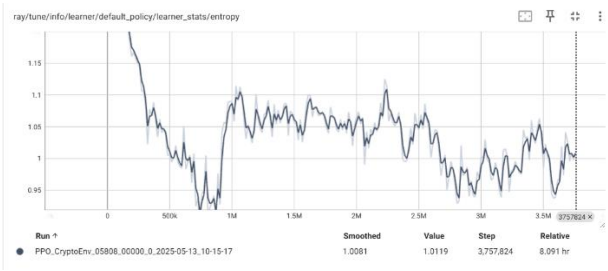
Config:

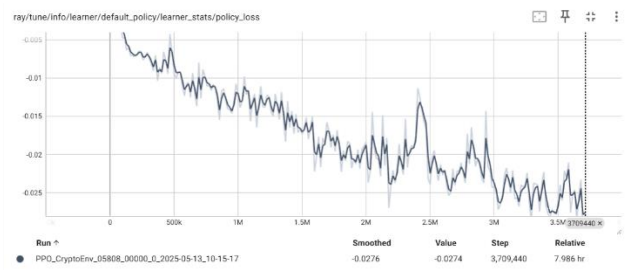
```
    .training(
        lr=1e-4,
        gamma=0.995, # 1.
        grad_clip=2,
        entropy_coeff=0.001,
        kl_coeff=0,
        kl_target=0.01, # not used if kl_coeff == 0.
        num_sgd_iter=10,
        use_gae=True,
        # lambda=0.95,
        clip_param=0.2, # larger values for more policy change
        vf_clip_param=1,
        vf_loss_coeff=0.25, #
        train_batch_size=8 * 6 * 168* 2, # num_rollout_workers * num_envs_per_worker * ro
llout_fragment_length * multiplier
        shuffle_sequences=True,
        model={
            "custom_model": "SimpleTransformer",
            "custom_model_config": {
                "embed_size": 128,
                "nhead": 4,
                "nlayers": 3,
                "seq_len": 168,
                "dropout": 0.1,
                "cnn_enabled": False,
                "freeze_cnn": False,
            }
        }
    )
```

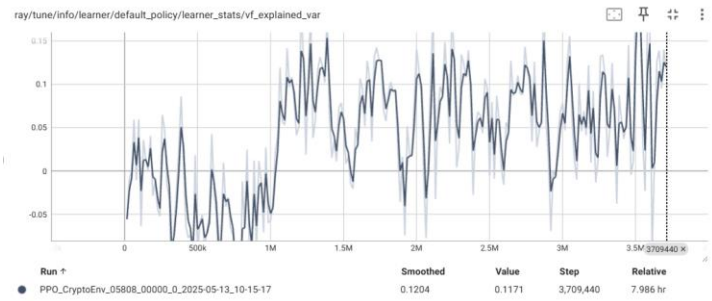
Results:











- **Policy entropy** remains high, indicating continued exploration or indecisive policy.
- **Value function explained variance** is positive but is still unstable.
- **Policy loss** is trending downward, implying the actor is optimizing its objective.
- **Mean episode reward** is decreasing, which is unexpected given the other metrics indicate that agent learns something.

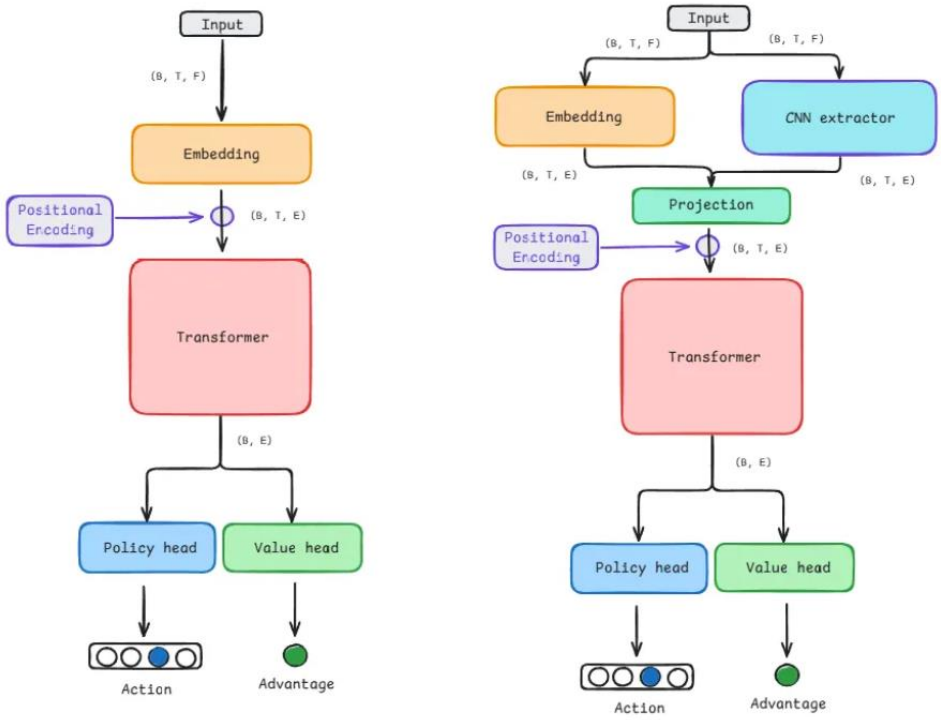
Transformer-only Conclusions

Of all the runs, Run #13 showed the most promising results. There are some points in the training that looks like the agent tries to learn a good policy but then falls off. I plan to revisit the most effective checkpoints from this run, adjust training parameters, and resume training from those points in an attempt to fine-tune the model further.

However, due to time constraints, we have to proceed to the next step and integrate the CNN front end into the architecture and proceed with experiments.

7. CNN front-end Implementation

A 1-D causal convolutional neural network is used as a feature extractor to capture local features and feed them into the transformer. The CNN feature extractor sits in front of the transformer and processes each 168-hour window of the raw data.



The CNN outputs a compact feature map of the same length as the input that's concatenated with the transformer's embeddings. Then, the new input vector is compressed back to *embed_size* so the transformer's parameter count stays uncharged.

Causal convolutions are a type of convolution used for temporal data, which ensures the model only uses current and past data and doesn't look into the future, perfect for time-series analysis.

Understanding PyTorch's Conv1D implementation:

PyTorch collapses the feature dimensions during convolution. Each filter combines information across ALL input features:

1. **Input:** PyTorch expects *(batch_size, input_size=72, sequence_length=168)*
2. **1 Filter (Kernel):**
    - A single filter has shape (*kernel_size=3, input_size=72*) → Weights: (3, 72)
    - This kernel slides over the time axis and computes a single scalar by multiplying the values of all the features x 3 → Output (1, 168)

Therefore, the output of the Conv1D layer in PyTorch returns a tensor of (*batch_size, num_kernels=64, sequence_length=168*).

7.1 CNN Pre-training

To pre-train the CNN as a feature extractor, it was essential to generate a training dataset that exactly matches the observation format of the training environment. This ensures the pre-trained CNN can be seamlessly integrated into the full pipeline, with the option to freeze or further fine-tune its parameters during RL training.

During initial experimentation, several issues arose—such as data leakage and incorrect scaling—which led to ineffective CNN training and unnecessary time spent refining the pre-training dataset. As a result, only experiments made on the corrected dataset will be shown.

The jupyter notebook that creates the pre-training data is:

cnn_pretraining_dataset.ipynb

- The ***WindowDataset()*** function converts sequential time series data into a sliding window format for classification. It takes as an input the metrics array and the price array, then computes binary labels. It extracts windows of length *seq_len* and assigns the corresponding label of the next step after the window ends. The result is a dataset where each sample consists of a sliding window of feature vectors and a list of the corresponding binary labels.

  Inputs: *(N of samples, features)* & *(N of samples, 1)* → **Outputs:** *(N of windows, sequence length, features)* & *(N of windows, 1)*

  The window range is from ***t*** to ***t + seq_len*** and the label is ***t + (seq_len + 1)***.

```
def WindowDataset(metrics, prices, seq_len=168):
    seq_len = seq_len
    X, y = [], []

    returns = np.diff(prices, axis=0)
    y_class = (returns > 0).astype(np.int8)

    for t in range(len(metrics) - seq_len):
        X.append(metrics[t:t+seq_len])
        y.append(y_class[t+seq_len-1])

    X = np.array(X) # (N, T, F)
    y = np.array(y) # (N, 1)
    return X, y
```

- The ***scale_window()*** function scales each window individually, exactly like the training environment does before sending an observation. For each window, it fits a *RobustScaler* using specified quantile ranges to make the scaling robust to outliers. The features are then scaled, clipped to a fixed range to limit extreme values, and converted to *float32*. This approach ensures that each input window is normalized independently, mirroring the environment's scaling process.

```
def scale_window(win, min_q=1, max_q=99, clip=10.0, coef=1e3):
    """
    win : (T=168, F=74)  raw values
    returns (T, F)  scaled exactly like env.Scaler
    """
    scaler = RobustScaler(quantile_range=(min_q, max_q))
    scaler.fit(win)  # fit on this *window* only
    feats = scaler.transform(win)
    feats = np.clip(feats, -clip, clip)

    return feats.astype(np.float32)
```

- The ***split_hourly_daily()*** function separates features in a time-series window into hourly and daily groups. It first removes the day and hour columns from the feature set, then identifies indices for hourly and daily features using string matching ("*_1h_*" and "*_1d_*"). It extracts the hourly features directly and reshapes the daily features. It returns the hourly and daily data, and the names of the columns in each group.

```
def split_hourly_daily(window: np.ndarray, col_names: np.ndarray):
    # Use first two columns for logic, but drop them from features
    day_of_week = window[:, :, 0]
    hour = window[:, :, 1]
    features = window[:, :, 2:]
    col_names = col_names[2:]

    # Find indices for hourly and daily features
    hourly_idx = np.array([i for i, name in enumerate(col_names) if "_1h_" in name])
    daily_idx = np.array([i for i, name in enumerate(col_names) if "_1d_" in name])

    # Extract hourly and daily data
    hourly_data = features[:, :, hourly_idx]
    daily_data_full = features[:, :, daily_idx]

    # For daily data, select rows where hour == 0 (start of each day)
    N, T, D = daily_data_full.shape
    days = T//24

    daily_data_reshaped = daily_data_full.reshape(N, days, 24, D)
    daily_data = daily_data_reshaped[:, :, 0, :]

    return hourly_data, daily_data, col_names[hourly_idx], col_names[daily_idx]
```

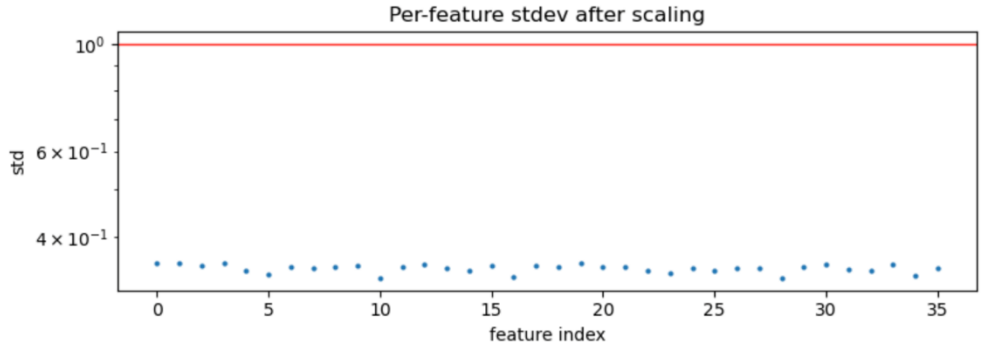
This separation is necessary because the dataset contains both hourly and daily features, but the daily features remain constant throughout each 24-hour period. If both types are processed together, the repeated daily values can confuse the CNN, especially since its receptive field (15 steps) is much smaller than a full day. As a

result, daily features would not provide meaningful information and could negatively affect learning. By extracting and separating hourly and daily features, we can train different CNNs for each timeframe.

First, the dataset is converted into sliding windows. Then, each window is individually scaled, and the scaled dataset is separated into hourly and daily groups. The resulting hourly and daily dataset, along with the target labels, are saves as *.npy* files for use in the CNN pre-training process.

```
X, y = WindowDataset(X, y)
X_scaled = np.empty_like(X)
for i in tqdm(range(X.shape[0]), desc="Scaling windows"):
    X_scaled[i] = scale_window(X[i])
X_hourly, X_daily, hourly_names, daily_names = split_hourly_daily(X_scaled, feat_names)
```

```
Scaling windows: 100%|██████████| 44257/44257 [04:34<00:00, 160.95it/s]
```




```
# Save as CNN pretraining dataset as .npy files
save_path = "./data/cnn_pretrain"
np.save(os.path.join(save_path, "hourly_outfile.npy"), X_hourly)
np.save(os.path.join(save_path, "daily_outfile.npy"), X_daily)
np.save(os.path.join(save_path, "targets_outfile.npy"), y)
```

The CNN pre-training file:

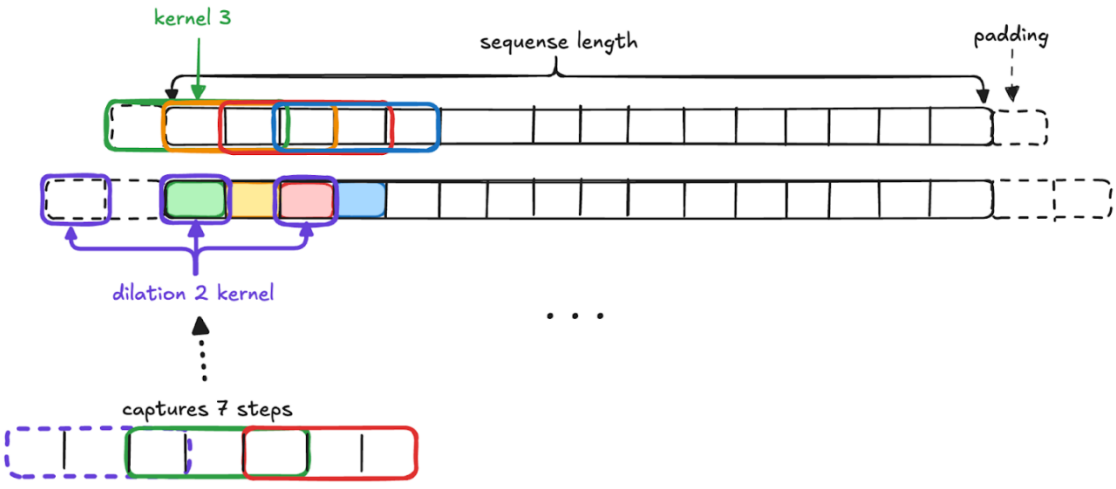
train_cnn.py

The **causal CNN** architecture used:

```python
class CausalCNN(nn.Module):
    def __init__(self, input_size, embed_size):
        super().__init__()
        self.cnn = nn.Sequential(
            nn.Conv1d(input_size, 64, kernel_size=3, padding=1),
            # nn.BatchNorm1d(64),
            nn.GELU(),
            # nn.Dropout1d(0.2),
            nn.Conv1d(64, 64, kernel_size=3, padding=2, dilation=2),
            # nn.BatchNorm1d(64),
            nn.GELU(),
            # nn.Dropout1d(0.2),
            nn.Conv1d(64, 64, kernel_size=3, padding=4, dilation=4),
            # nn.BatchNorm1d(64),
            nn.GELU(),
            # nn.Dropout1d(0.2),
            nn.Conv1d(64, embed_size, kernel_size=1)
        )

    def forward(self, x):
        return self.cnn(x)
```

It consists of four 1-D convolutional layers with 64 kernels of size 3, padding and dilation.

- **Padding** → Added to keep the sequence length unchanged.
- **Dilation** → Added to increase the filter's receptive field while keeping lower computational cost (1 → 2 → 4 = 15 steps receptive field).



The way the dilation works, every kernel always sees one overlapping step from the previous representations, as shown above. So, at the final convolution we keep an entire 15 step receptive field.

A final 1 size kernel convolution layer acts as a pooling layer, returning the final feature maps:

Input: *(batch size, features, seq_length)* → **Output**: *(batch size, embed_size, seq_length)*

The **classification head** is pretty straightforward:

```python
class ClassificationHead(nn.Module):
    def __init__(self, embed_size):
        super().__init__()
        self.head = nn.Sequential(
            nn.Linear(embed_size, 128),
            nn.ReLU(),
            # nn.Dropout(0.2),
            nn.Linear(128, 2)
        )

    def forward(self, x):
        return self.head(x);
```

A single hidden layer neural network takes the pooled extracted feature maps and makes the final prediction:

Input: *(batch size, embed_size)* → **Output**: *(batch_size, 2)*

During training:

We use Adam as the optimizer and cross entropy loss function.

```python
# model
cnn = CausalCNN(X.shape[-1], embed_size=args.embed_size).to(device)
head = ClassificationHead(args.embed_size).to(device)
optimizer = torch.optim.Adam(list(cnn.parameters()) + list(head.parameters()), lr=args.learning_rate)
criterion = nn.CrossEntropyLoss(label_smoothing=0.05)
```

The pipeline works as following:

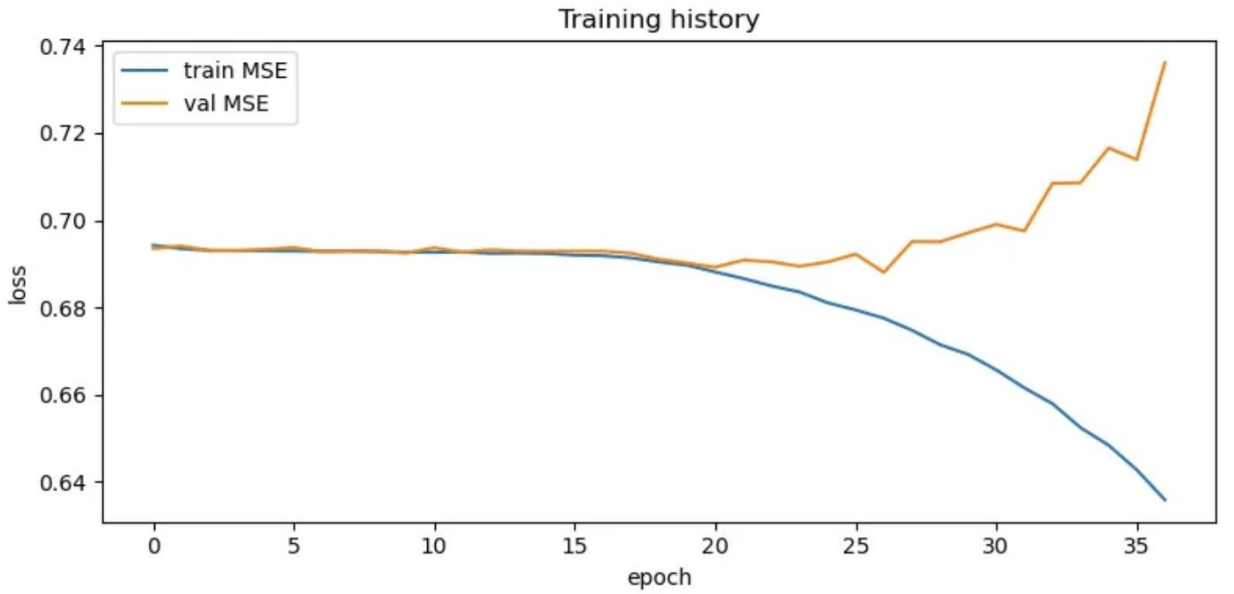
1. Samples (batch) are passed through the CNN → *(batch_size, embed_size, seq_len)*.
2. The output is max pooled → *(batch_size, embed_size)*
3. The pooled samples are the passed through the classification head that makes the final prediction → *(batch_size, 2)*

```
X, y = X.to(device), y.squeeze().long().to(device)
feat_map = cnn.forward(X.permute(0, 2, 1))  # (B, E, T)
pooled, _ = feat_map.max(dim=2) # (B, E)
logits = head(pooled)  # (B, 2)
loss = criterion(logits, y)

optimizer.zero_grad()
loss.backward()
optimizer.step()
train_loss += loss.item()
```

Runs

******* The diagrams show Average Cross Entropy Loss per epoch, not MSE *******

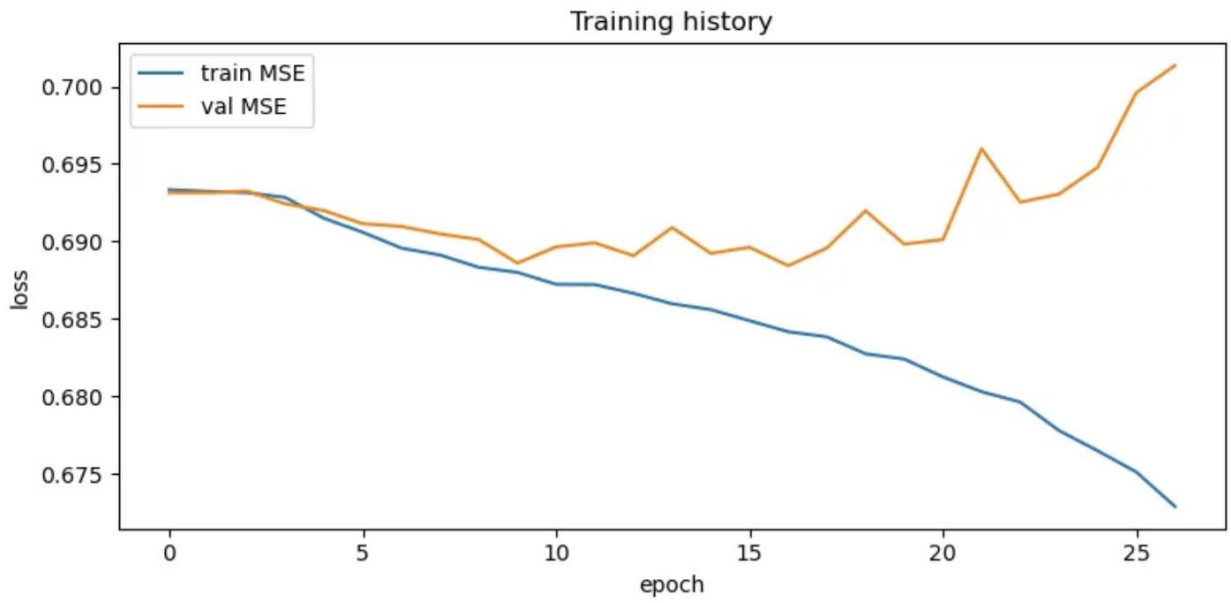
```
python train_cnn.py --epochs=200 --patience=10 --learning_rate=1e-3
```



Validation balanced accuracy: 52.84%

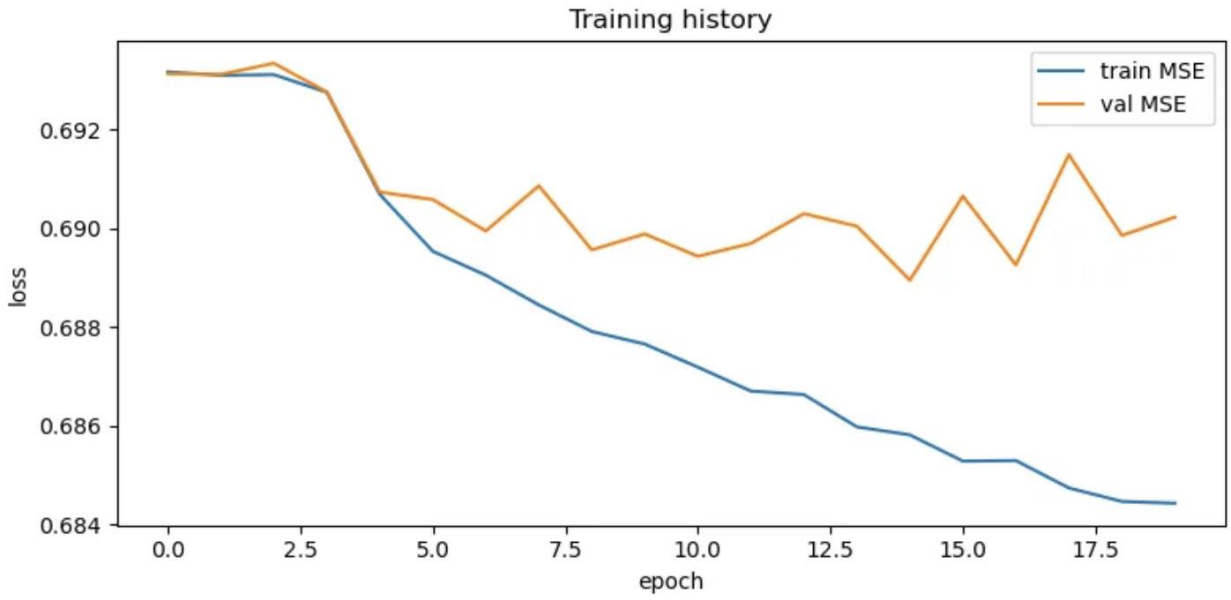
Various combinations of learning rates and batch sizes were tested, along with the inclusion of batch normalization and dropout layers, to improve the training stability and performance of the CNN.

- 1 batchnorm layer in the first conv layer, batch size 64, learning rate 6e-4:

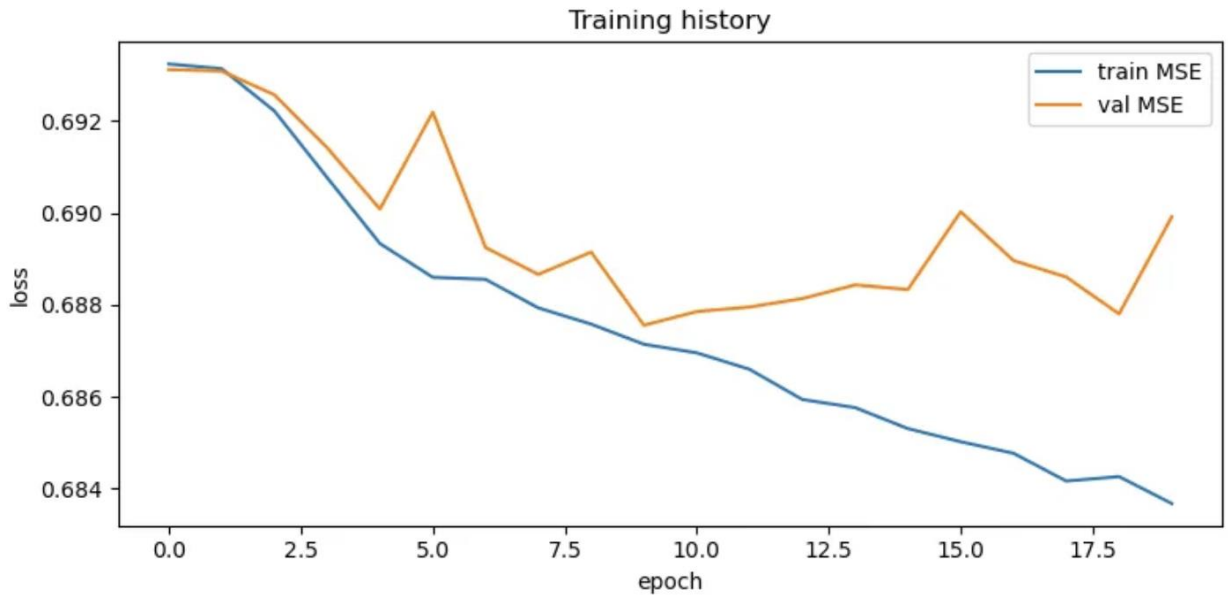


- One more hidden layer added to the classification head:

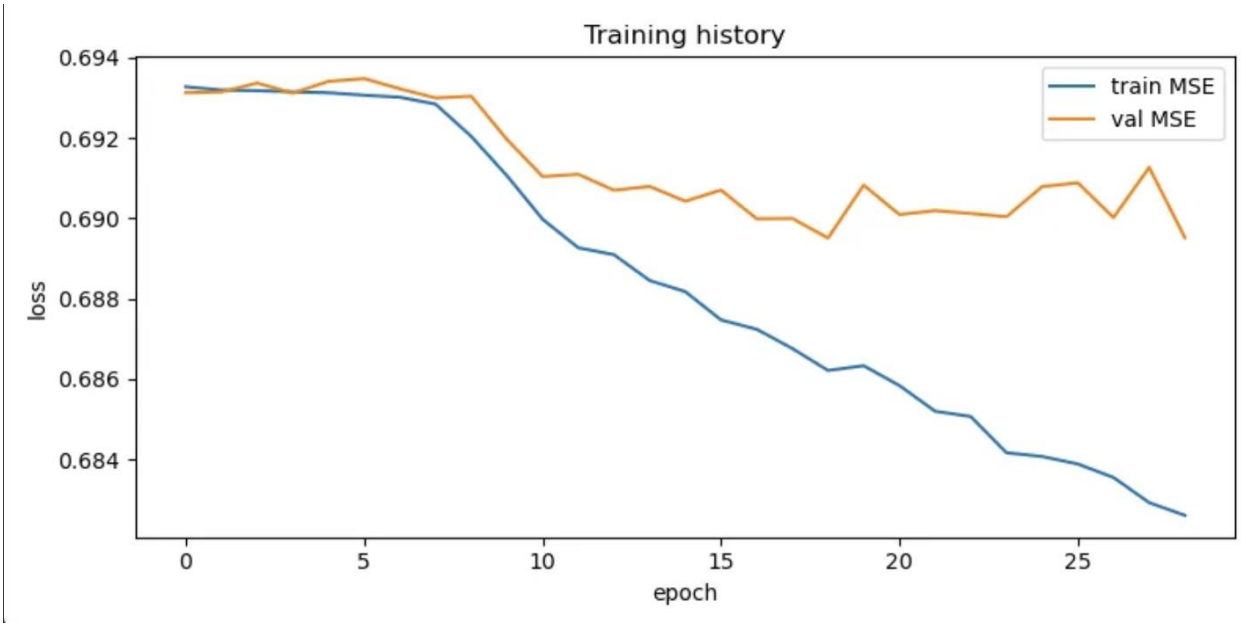
```
nn.Linear(embed_size, 128),
nn.ReLU(),
nn.Linear(128, 64),
nn.ReLU(),
nn.Linear(64, 2)
```



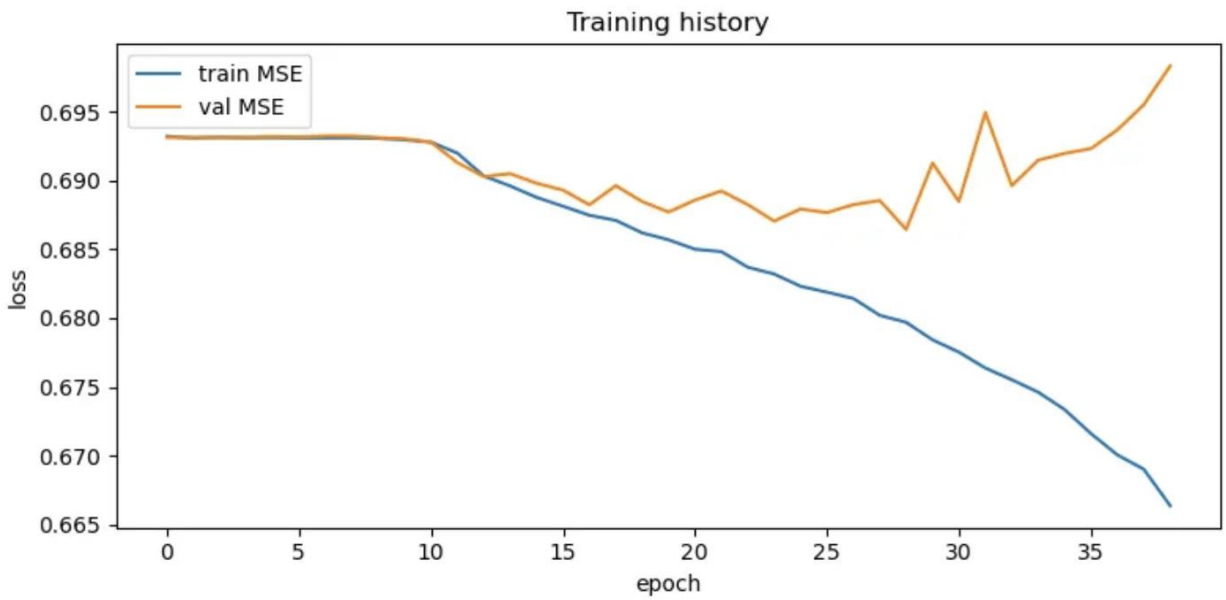
- batch_size 64, learning rate 3e-4:



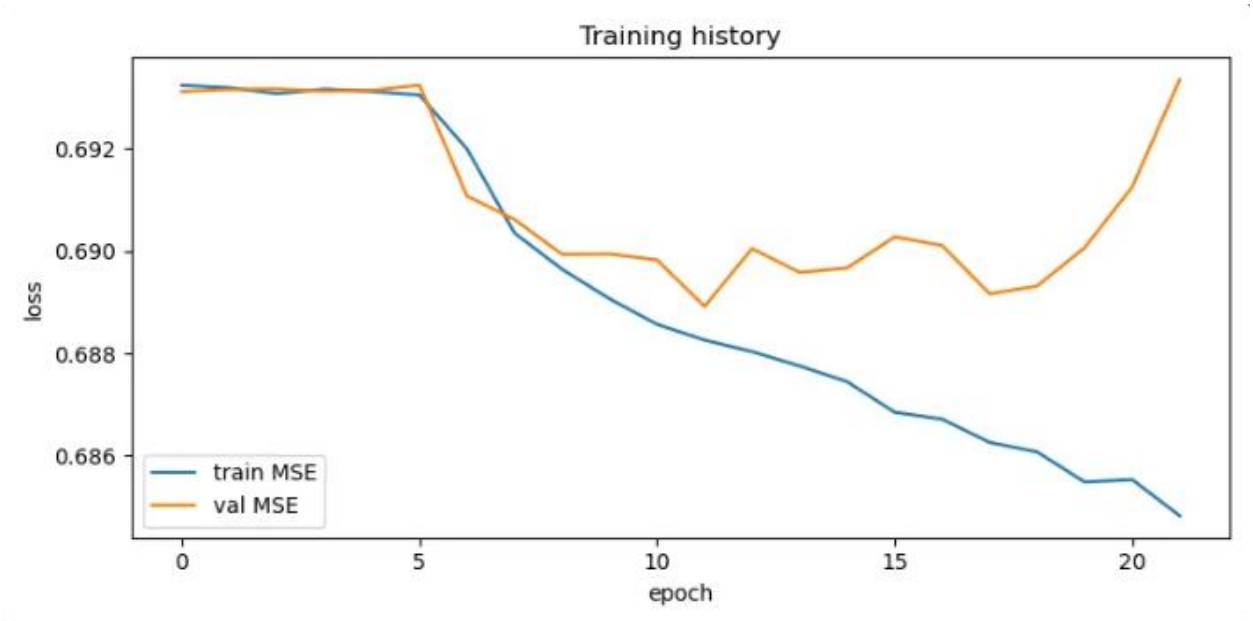
- Added dropout(0.2) to the head:



- Added extra convolution with dilation and padding 8, batch size 128, learning rate 3e-4:



- Learning rate 3e-4, batch size 64:



Despite extensive experimentation—including batch normalization, dropout, weight decay, label smoothing, and a wide range of batch size and learning rate combinations—training remained challenging.

Due to time constraints, we must move on with the integration without pretraining, unfreeze the parameters and let the agent run for two final runs.

7.2 CNN front-end integration

After pre-training, the CNN is inserted in the pipeline, and the options are *cnn_enabled = True/False* and *freeze_cnn=True/False*. If the parameters are frozen, the CNN's parameters don't change, avoiding adding extra noise and destabilize RL gradients. It just acts as a feature extractor, providing the transformer with important information such as local patterns or micro-trends, using its pre-trained weights. When PPO is stable, we can un-freeze the CNN parameters to fine tune it further.

```
        # -------------- CNN Front-end --------------------
        if self.cnn_enabled:
            self.hourly_cnn = nn.Sequential(
                nn.Conv1d(self.input_dim, 64, kernel_size=3, padding=1),
                nn.GELU(),
                nn.Conv1d(64, 64, kernel_size=3, padding=2, dilation=2),
                nn.GELU(),
                nn.Conv1d(64, 64, kernel_size=3, padding=4, dilation=4),
                nn.GELU(),
                nn.Conv1d(64, self.embed_size, kernel_size=1)
            )
         hourly_state = torch.load("hourly_cnn_pretrain.pt")

         if self.freeze_cnn:
                for p in self.hourly_cnn.parameters():
                    p.requires_grad = False
```

The final CNN architecture processes only the hourly component of the data, as integrating daily features proved complex and time constraints prevented further experimentation. This model focuses on extracting short-term temporal patterns from the hourly sequence.

```
if self.cnn_enabled:
            hourly_data = self.split_hourly_daily(x)
            hourly_feat_maps = self.hourly_cnn(hourly_data.permute(0, 2, 1))  # (N, embed
_size, seq_len)
```

The extracted features are concatenated with the embedding of the raw inputs, and then projected back to *embed_size* to be compatible with the transformer's expected input.

```
x = self.input_embed(x)
if self.cnn_enabled:
     concat = torch.cat((hourly_feat_maps.permute(0, 2, 1), x), dim=2) # (N, seq_len, 2*e
mbed_size)

      x = self.projection(concat) # projection back to embed_size
```

All other components of the model, including positional encoding and downstream processing, remain unchanged. The only adjustment is that the CNN now exclusively operates on hourly features.

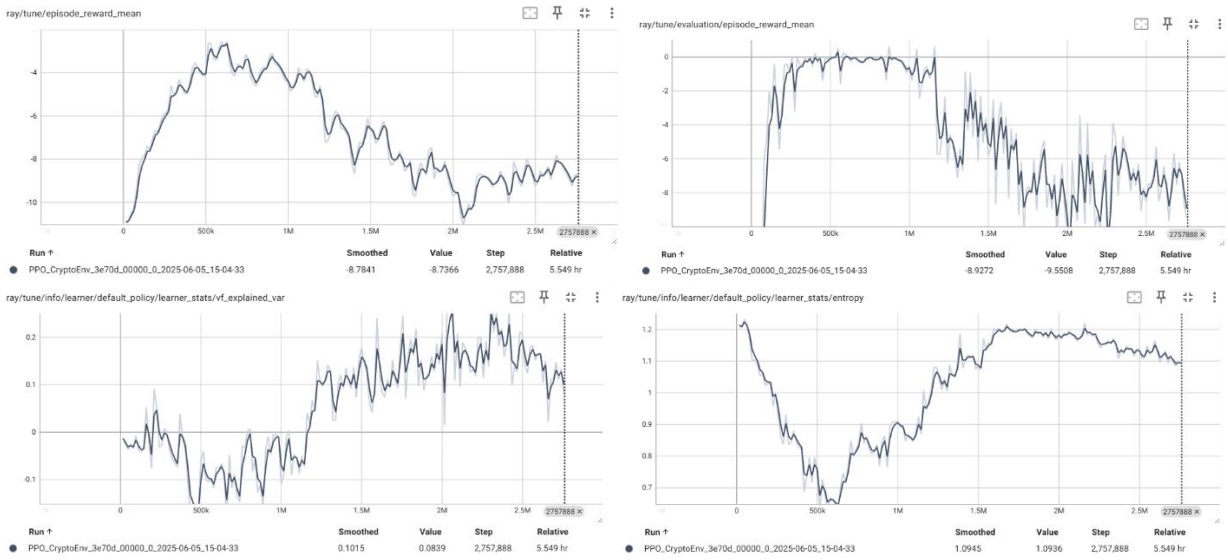
Final Runs:

Final run 1:

Config:

```
.training(
        lr=1e-4,
        gamma=0.995, # 1.
        grad_clip=2,
        entropy_coeff=0.001,
        kl_coeff=0,
        kl_target=0.01, # not used if kl_coeff == 0.
        num_sgd_iter=10,
        use_gae=True,
        # lambda=0.95,
        clip_param=0.2, # larger values for more policy change
        vf_clip_param=1,
        vf_loss_coeff=0.25, #
        train_batch_size=8 * 6 * 168* 2, # num_rollout_workers * num_envs_per_worker * ro
llout_fragment_length * multiplier
        shuffle_sequences=True,
        model={
            "custom_model": "SimpleTransformer",
            "custom_model_config": {
                "embed_size": 128,
                "nhead": 4,
                "nlayers": 3,
                "seq_len": 168,
                "dropout": 0.1,
                "cnn_enabled": True,
                "freeze_cnn": False,
            }
        }
    )
```

Results:



The integration of the CNN front end appears to have a noticeable effect on the on the agent's performance. The mean reward during training initially rises sharply, and by looking

at the validation reward we can see that the agent very quickly converges to the “do nothing” policy.

The value function explained variance shows improved stability and remains positive, which is a significant improvement compared to the previous runs. Interestingly, when the explained variance turns positive, the mean reward begins to drop. We can also see that the entropy drops significantly at the beginning as well, and then rises. All of this information indicates that initially the agents quickly converge to the “do nothing” policy that significantly increasing the reward, but then diverges in search for a new policy.

Final run 2:

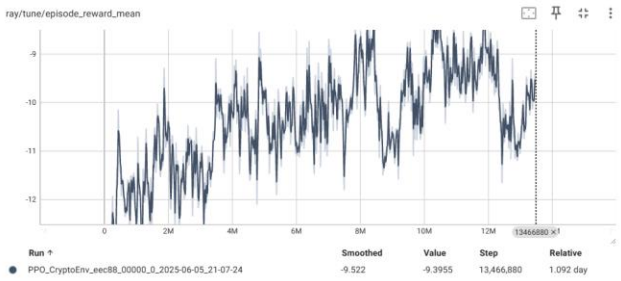
Adjustments to Environment:

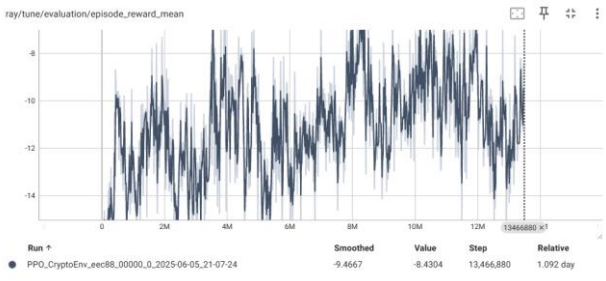
To address the agent’s tendency to converge on the “do nothing” policy, we conducted a final experiment with a modified action space limited to three actions:

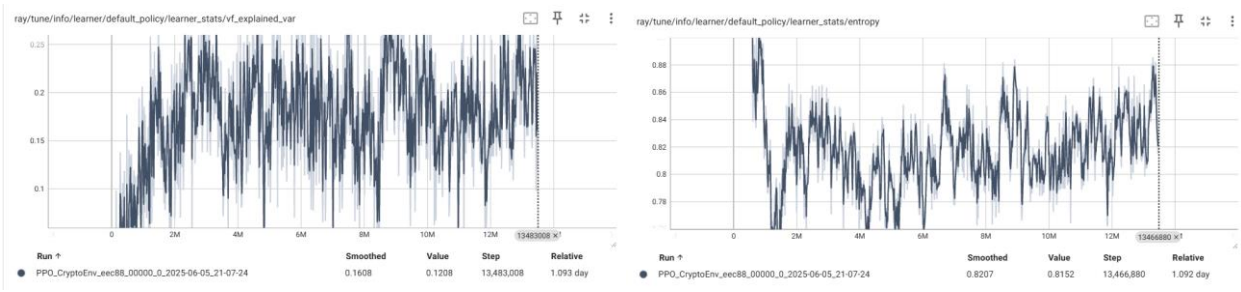
1. Do nothing
2. Close any open position and open a long
3. Close any open position and open a short

This setup removes the agent’s ability to simply stay out of the market, effectively eliminating the “do nothing” local minimum. With the revised reward function penalizing unrealized losses, persistent buy-and-hold is also discouraged. As a result, the agent is now forced to continuously seek profitable trading decisions, ideally maximizing returns at every step. More extensive experimentation of this action space should be considered.

Results:







As expected, the task became significantly more difficult, and this reflects on the level of instability introduced.

8. Conclusions

Although the hybrid CNN-Transformer RL agent did not achieve a profitable or stable trading policy within the scope of these experiments, the outcome highlights the real-world challenges of stabilizing deep reinforcement learning pipelines—especially in the volatile domain of algorithmic trading.

Despite extensive experimentation with architectures, reward functions, and hyperparameters, the agent consistently converged to trivial or suboptimal strategies (such as “do nothing” or random action selection). These results are disappointing, but they also realistically reflect the difficulty of the task.

Nevertheless, the project provided substantial hands-on experience in RL for financial markets, data preparation, and model design. Valuable lessons were learned in diagnosing training instability and environment/reward design.

There is significant room for further exploration:

- **Advanced architectures:** e.g., using multiple transformers specialized for different feature sets or timeframes
- **Longer training times** and improved reward shaping
- **More robust environment configurations** and evaluation protocols

Although the current version of the agent did not succeed, this work lays a solid foundation for future development, and I plan to continue exploring and refining these ideas. Failure is part of research, and the insights gained here are a step closer into building more capable agents going forward.