



TECHNICAL PROPOSAL | UNDATED MANUSCRIPT

Constant TAO With a Data Lake for Delayed Rebalancing

Brian Mingus

OpenTensor Foundation · manuscript affiliation

A mechanism-design proposal exploring delayed rebalancing and conditional trades across Bittensor subnets, with a PyTorch simulation.

Collection cover added September 2026. Original manuscript follows unchanged.
AI-generated conceptual artwork; separate from the scientific figures.

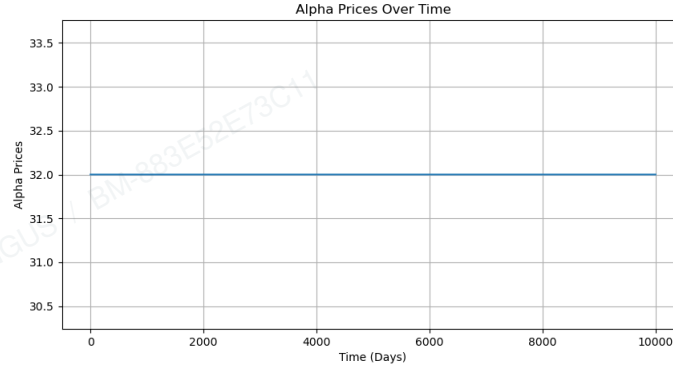
brianmingus.com

Constant TAO With a Data Lake for Delayed Rebalancing

Brian Mingus
OpenTensor Foundation

Abstract

We introduce a mechanism that enables the rebalancing of holdings of TAO across subnets without affecting the value of the TAO held - enabling zero-risk high-frequency trading (HFT). We complement this mechanism with Conditional Trades - sets of conditions applied to off-chain (in-data lake) delayed trades that determine if the trade should execute on-chain at a later date (yet to be implemented).



1 Constant TAO

```
import matplotlib.pyplot as plt
import torch
from torch import full, ones, rand, multinomial, set_default_dtype
from torch.jit import script
from torch.cuda import is_available, set_device
import time

# For plotting graphs
# PyTorch library for tensor computations
# Import specific functions from torch
# For JIT compilation of functions
# For CUDA device handling
# Import the time module for timing operations

#####
#
# Function to select the computation device (CPU or GPU)
#
#####
def select_device(cpu):
    # Selects between CPU and GPU for computations. GPU is used for its efficiency if available, otherwise CPU is used.
    if not cpu and is_available():
        set_device(0)
        return 'cuda'
    return 'cpu'

set_default_dtype(torch.float32)

CPU = True
device = select_device(CPU)

#####
#
# Function to compute alpha prices using PyTorch
#
#####
@script
def compute_alpha_prices(n: int, i_t: float, n_days: int, v: float, device: str):
    # Computes alpha prices over a number of days using a Constant TAO approach.
    # This approach maintains a fixed rate of token allocation across the simulation period.

    t_r = a_r = a_o = full((n,), i_t / n, device=device)
    k = t_r * a_r
    sp = torch.empty(n_days, device=device)

    for day in range(n_days):
        a_p = 2 * t_r / (a_r + a_o)
        r_i = a_p / a_p.sum()
        update = torch.tensor(7200.0, device=device) * r_i
        t_r += update
        a_r += update
        a_o += update
        k = t_r * a_r

    # Random swaps simulate market transactions in a constant TAO environment
    indices = multinomial(ones(n, device=device), 2)
    # Randomly select two tokens for swap
```

```

a, b = indices[0], indices[1] # Unpack indices
amt = a_o[a] * rand(1, device=device).squeeze() * v # Calculate swap amount based on rand and volatility

# Perform updates post swap, maintaining the constant allocation mechanism
nar = a_r[a] + amt # New alpha reserve for token a
ntr = k[a] / nar # New tao reserve for token a
tb = t_r[a] - ntr # Tao balance change for token a
a_o[a] -= amt # Update alpha outstanding for token a
a_r[a] = nar # Update alpha reserve for token a
t_r[a] = ntr # Update tao reserve for token a
k[a] = ntr * nar # Update k value for token a

# Similar updates for token b
ntr = t_r[b] + tb # New tao reserve for token b
nar = k[b] / ntr # New alpha reserve for token b
ab = a_r[b] - nar # Alpha balance change for token b
a_o[b] += ab # Update alpha outstanding for token b
a_r[b] = nar # Update alpha reserve for token b
t_r[b] = ntr # Update tao reserve for token b
k[b] = ntr * nar # Update k value for token b

sp[day] = a_p.sum() # Store the sum of alpha prices for the day

return sp # Return the computed prices

#####
#
# Main execution block for computing and plotting alpha prices
#
#####
n, i_t, n_days, v = 32, 6000000.0, 10000, 0.1 # Initialize parameters for the simulation

start_time = time.time() # Start the timer to measure execution time

# Run the computation
sp = compute_alpha_prices(n, i_t, n_days, v, device).cpu().numpy() # Compute alpha prices and transfer results to CPU

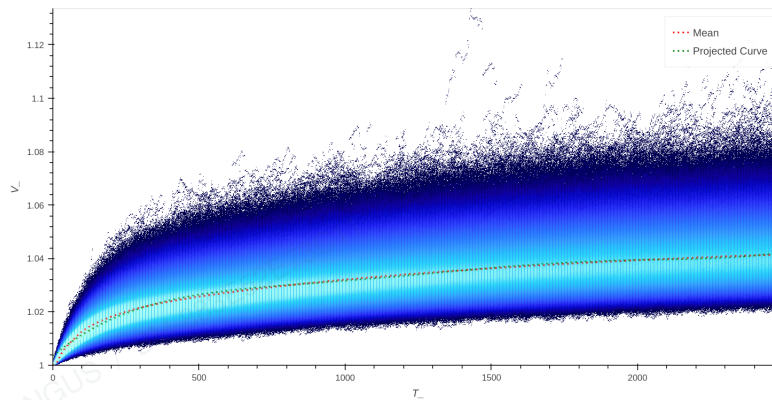
end_time = time.time() # End the timer and calculate the elapsed time
elapsed_time = end_time - start_time
print(f"Computation Time: {elapsed_time:.3f} seconds") # Print the computation time

# Plotting the results
plt.figure(figsize=(10, 5)) # Set figure size for the plot
plt.ticklabel_format(style='plain', axis='y') # Format the y-axis labels to plain style
plt.plot(sp) # Plot the alpha prices as computed
plt.title('Alpha Prices Over Time') # Set title for the plot
plt.xlabel('Time (Days)') # Label for x-axis
plt.ylabel('Alpha Prices') # Label for y-axis
plt.grid(True) # Enable grid for easier readability
plt.show() # Display the plot

# This plot visualizes the trend of alpha prices over time in a Constant TAO environment, reflecting
# the fixed rate allocation mechanism of tokens regardless of individual node contributions.

```

2 TAO



```

from torch import arange as ar, float, full, empty, randint, rand, set_default_dtype
from torch.cuda import set_device
from torch.jit import script

from holoviews.operation.datashader import datashade, dynspread
from holoviews import opts

import holoviews as hv
import pandas as pd
import numpy as np
import torch

#####
#
# Initialize PyTorch for GPU
#
#####
set_default_dtype(float) # Set the default CUDA device
set_device(0)

#####
#
# Simulation parameters
#
#####
n_s = 150000 # Number of simulations (n_s)
n_e = 32 # Number of entities (n_e)

```

```

i_v = 6000000 # Initial value (i_v)
d_d = 2500 # Number of days (d_d)
v_v = 0.1 # Volatility value (v_v)

#####
#
# Initialize arrays for batch processing
#
#####
t_r = full((n_s, n_e), i_v / n_e, dtype=float, device='cuda') # Tao reserves (t_r)
a_r = full((n_s, n_e), i_v, dtype=float, device='cuda') # Alpha reserves (a_r)
o_r = full((n_s, n_e), i_v, dtype=float, device='cuda') # Alpha outstanding (o_r)

k_v = t_r * a_r # K value (k_v)

#####
#
# Define the simulation function
#
#####
@script
def run_sim(t_r, a_r, o_r, k_v, d_d: int, n_e: int, v_v: float, n_s: int):
    a_n = ar(n_s) # range num sims (a_n)

    s_r = empty((n_s, d_d), device='cuda') # Simulation results (s_r)
    for j in range(d_d):
        p_r = t_r / a_r # Price ratio (p_r)
        s_r[:, j] = p_r.sum(dim=1) # Sum prices for each simulation

        r_p = p_r / p_r.sum(dim=1, keepdim=True) # Ratio of price (r_p)
        t_r += 7200 * r_p # Update tao reserves (t_r)
        a_r += 7200 # Update alpha reserves (a_r)
        o_r += 7200 # Update alpha outstanding (o_r)
        k_v = t_r * a_r # Update k value (k_v)

        g_i = randint(0, n_e, (2, t_r.shape[0]), device='cuda') # Random indices (g_i)
        x, y = g_i[0], g_i[1] # Indices x and y (x, y)
        f = rand((t_r.shape[0],), device='cuda') # Random factor (f)
        m = o_r[a_n, x] * f * v_v # Amount for swap (m)

        w = a_r[a_n, x] + m # New alpha reserve for x (w)
        u = k_v[a_n, x] / w # New tao reserve for x (u)
        e = t_r[a_n, x] - u # Tao change for x (e)

        o_r[a_n, x] -= m # Update alpha outstanding for x (o_r)
        a_r[a_n, x] = w # Update alpha reserve for x (a_r)
        t_r[a_n, x] = u # Update tao reserve for x (t_r)
        k_v[a_n, x] = t_r[a_n, x] * a_r[a_n, x] # Update k value for x (k_v)

        h = t_r[a_n, y] + e # New tao reserve for y (h)
        w = k_v[a_n, y] / h # New alpha reserve for y (w)
        b = a_r[a_n, y] - w # Alpha change for y (b)

        o_r[a_n, y] += b # Update alpha outstanding for y (o_r)
        a_r[a_n, y] = w # Update alpha reserve for y (a_r)
        t_r[a_n, y] = h # Update tao reserve for y (t_r)
        k_v[a_n, y] = t_r[a_n, y] * a_r[a_n, y] # Update k value for y (k_v)

    return s_r

#####
#
# Run the simulation
#
#####
s_s = run_sim(t_r, a_r, o_r, k_v, d_d, n_e, v_v, n_s) # Simulation steps (s_s)

#####
#
# Transfer results to CPU for analysis and plotting
#
#####
s_c = s_s.cpu().numpy() # Simulation results on CPU (s_c)

#####
#
# Holoviews and Datashader for plotting
#
#####
hv.extension('bokeh')

#####
#
# Calculate the mean of the simulations
#
#####
m_s = s_c.mean(axis=0) # Mean simulation (m_s)

#####
#
# Fit a polynomial to the mean simulation data
#
#####
c_f = np.polyfit(np.arange(len(m_s)), m_s, 5) # Coefficients (c_f)
p_c = np.polyval(c_f, np.arange(len(m_s))) # Projected curve (p_c)

#####
#
# Prepare DataFrame for Datashader
#
#####
df = pd.DataFrame(s_c).melt(var_name='T_', value_name='V_')
df['S_'] = df.index // d_d # Simulation index (S_)

#####
#
# Use Datashader to plot the simulations
#
#####
p_s = hv.Points(df, kdims=['T_', 'V_']) # Points (p_s)
s_d = datashade(p_s, cmap='kbc', width=1000, height=500) # Shaded plot (s_d)
s_p = dynspread(s_d, max_px=1) # Spread plot (s_p)

#####
#
# Create Holoviews objects for the mean and polynomial curves
#

```

```
#####
m_c = hv.Curve((np.arange(len(m_s)), m_s), label="Mean")          # Mean curve (m_c)
p_l = hv.Curve((np.arange(len(m_s)), p_c), label="Projected Curve") # Poly curve (p_l)
#####
#
# Overlay the curves on top of the Datashader plot
#
#####
o_l = s_p * m_c.opts(color='red', line_width=2) * p_l.opts(color='green', line_width=2) # Overlay (o_l)
o_l = o_l.opts(
    opts.Overlay(width=1000, height=500, show_legend=True),
    opts.Curve(tools=['hover'], line_dash='dotted')
)
#####
#
# Display the plot
#
#####
o_l
```