

reinforcement_learning

May 26, 2026

Nikhil Sunder

ICSRI Research Fellow, University of Miami | Open-Source Developer

Herbert Business School, University of Miami

B.S.B.A. Quantitative Economics & Finance; Minor in Mathematics

Correspondence: nss106@miami.edu | +1 (305) 409-3108

Research profiles: [GitHub](#) · [ORCID](#) · [LinkedIn](#) · [Zenodo](#)

Software: [PyPI](#) · [Anaconda](#)

1 Reinforcement Learning Environment

1.1 Reproducibility and runtime

The optimised policies in this notebook are produced from random initial weights, exploration noise, and (where applicable) stochastic transitions in the underlying economy environments. To preserve reproducibility, all random number generators (Python, NumPy, PyTorch, the Stable-Baselines3 environment vectoriser, and PyMC) are seeded at the start of the notebook, and the algorithm hyper-parameters are stated explicitly in their respective sections.

The transition equations for the four economies are estimated in the *linear* and *non-linear* environment notebooks of this project. To avoid re-fitting them here we execute those notebooks at the top of this one via the IPython `%run` magic. The resulting variables (the fitted SVAR, the TVP-SVAR-SV posterior, the NARX networks and scalars, and the two-block NSSM) are then used directly to drive the RL environments below. Running both parent notebooks adds roughly five to ten minutes of fixed setup before the RL training begins; the architecture-search grid is intentionally narrowed (critic nodes from a discrete subset of $\{1, 2\}$, actor nodes for nonlinear policies from $\{1, 4, 8\}$) so the RL portion runs in roughly one to two additional CPU hours. Setting `FULL_GRID = True` recovers the full Hinterlang & Tanzer (2021, Table 2) search range.

1.2 Software and libraries

Both parent notebooks install their own dependencies (`fedfred`, `PyMC`, `PyTensor`, `PyTorch`, `statsmodels`); this notebook additionally installs `Stable-Baselines3` (Raffin et al., 2021), `gymnasium` (Towers et al., 2023), and `tqdm`. The full list of versions used during development is given in the bibliography.

```
[1]: %pip install stable-baselines3 gymnasium tqdm --quiet
```

Note: you may need to restart the kernel to use updated packages.

1.3 Running the parent notebooks

The `%run` magic executes another `.ipynb` file in the current kernel, so every top-level variable defined in those notebooks becomes available here (in particular: the training DataFrame `df`, the fitted SVAR object `svar`, the TVP-SVAR-SV PyMC posteriors `idata_y` and `idata_pi`, the NARX networks `final_narx_y` / `final_narx_pi` with their `MapMinMaxScaler` instances, and the fitted `nssm` with its terminal latent-state trajectories). The `%%capture` magic suppresses each parent notebook's stdout, plots, and warnings so the output of this notebook stays focused on the RL exercise.

```
[2]: %%capture
      %run linear_environment.ipynb
```

```
[3]: %%capture
      %run nonlinear_environment.ipynb
```

```
[4]: # Sanity check: confirm that the parent notebooks have populated the expected_
      ↪names.
expected_from_linear = ["df", "svar", "idata_y", "idata_pi"]
expected_from_nonlinear = [
    "final_narx_y", "final_narx_pi",
    "x_scaler_y", "y_scaler_y", "x_scaler_pi", "y_scaler_pi",
    "nssm", "state_dim_y", "state_dim_pi",
    "h0_y_nssm", "h0_pi_nssm",
    "states_y_nssm", "states_pi_nssm",
    "fit_narx", "fit_nssm",
]
missing = [name for name in expected_from_linear + expected_from_nonlinear if
    ↪name not in globals()]
if missing:
    raise RuntimeError(
        "Expected variables missing from globals after %run of parent notebooks:
    ↪ "
        + ", ".join(missing)
    )
print("Parent notebooks ran successfully. Detected:")
print(f" df: {df.shape}, columns = {list(df.columns)}")
print(f" svar.mod_y regressors: {list(svar.mod_y.params.index)}")
print(f" svar.mod_pi regressors: {list(svar.mod_pi.params.index)}")
print(f" TVP idata_y posterior shape: beta = {idata_y.posterior['beta'].
    ↪shape}")
print(f" NARX final_narx_y input_size = {final_narx_y.input_size}")
print(f" NARX final_narx_pi input_size = {final_narx_pi.input_size}")
print(f" NSSM state_dim_y = {state_dim_y}, state_dim_pi = {state_dim_pi}")
print(f" NSSM states_y_nssm shape = {tuple(states_y_nssm.shape)}")
```

Parent notebooks ran successfully. Detected:
df: (80, 3), columns = ['pi', 'y', 'i']

```

    svar.mod_y regressors: ['const', 'y_lag1', 'y_lag2', 'pi_lag1', 'pi_lag2',
'i_lag1', 'i_lag2']
    svar.mod_pi regressors: ['const', 'y', 'y_lag1', 'y_lag2', 'pi_lag1',
'pi_lag2', 'i_lag1', 'i_lag2']
    TVP idata_y posterior shape: beta = (2, 800, 153, 7)
    NARX final_narx_y input_size = 3
    NARX final_narx_pi input_size = 6
    NSSM state_dim_y = 4, state_dim_pi = 4
    NSSM states_y_nssm shape = (78, 4)

```

```

[5]: # Imports specific to this notebook (parent notebooks already imported numpy,
↳pandas, torch, etc.)
from __future__ import annotations
from dataclasses import dataclass, field
from typing import Any, Callable, Dict, List, Optional, Sequence, Tuple

import warnings

import gymnasium as gym
from gymnasium import spaces

from stable_baselines3 import DDPG, SAC
from stable_baselines3.common.noise import NormalActionNoise
from stable_baselines3.common.callbacks import BaseCallback
from stable_baselines3.common.env_checker import check_env
from stable_baselines3.common.torch_layers import create_mlp
from stable_baselines3.common.preprocessing import get_action_dim
from stable_baselines3.td3.policies import Actor as TD3Actor, TD3Policy

import pandas as pd
import numpy as np

import torch
import torch.nn as nn

warnings.filterwarnings("ignore", category=UserWarning)

```

2 Reinforcement Learning Framework

2.1 Methodological framing

The reinforcement-learning (RL) setup follows the two-stage architecture proposed by Hinterlang & Tanzer (2021) in Deutsche Bundesbank Discussion Paper No. 51/2021. In the first stage, the transition equations of the macro-economy are estimated from observable data; in the second stage, an RL agent learns an interest-rate reaction function by interacting with the estimated economy, treating the transition equations as a fixed environment. Stage one is handled entirely by the linear-

and non-linear-environment notebooks of this project, which we ran at the top of this notebook; the four fitted environments (restricted recursive SVAR(2), TVP-SVAR-SV, restricted NARX, and two-block NSSM) are imported as-is.

The conceptual foundations of RL are due to Bellman (1957) and Howard (1960), and the modern algorithmic treatment is due to Sutton & Barto (2018). We implement two off-policy deep-RL algorithms via Stable-Baselines3 (Raffin et al., 2021) – the Deep Deterministic Policy Gradient (DDPG) algorithm of Lillicrap et al. (2015) and the Soft Actor-Critic (SAC) algorithm of Haarnoja et al. (2018) – and a custom tabular Q-learning implementation that matches Watkins (1989) and Watkins & Dayan (1992).

The DDPG implementation follows Hinterlang & Tanzer (2021, Section 2.1.2 and Table 1) line-by-line: it uses an actor-critic architecture with a nonlinear critic (single-hidden-layer feed-forward network), a *linear or nonlinear* actor selected per cell (eq. 12-13 with $G = \text{purelin}$ or $G = \tanh$), two observation specifications ($x_t^{(1)}$ and $x_t^{(2)}$ from eq. 9-10), a hidden-node search over the critic and (nonlinear) actor architectures, a per-episode agent-selection criterion ($ER_m/ES_m > -4$ and $1 < ES_m < 50$ followed by a steady-state-reward tie-break), and a training budget of $M = 500$ episodes per architecture. Because SB3’s default `MlpPolicy` for DDPG applies a tanh squash on the actor output that is *not* part of the H&T policy specification, we implement custom `LinearActor` and `NonlinearActor` classes that override SB3’s `TD3Actor` to skip this squash; the ZLB is enforced by the action-space clipping that SB3 applies at the env-step boundary.

2.2 Observation space (H&T eq. 9-10)

Both observation specifications are trained for every applicable cell:

$$x_t^{(1)} = (y_t, \pi_t), \quad x_t^{(2)} = (y_t, y_{t-1}, \pi_t, \pi_{t-1}).$$

The first mirrors the Taylor (1993) information set. The second adds one lag of each macro variable, which Hawkins et al. (2015) report produces more stabilising behaviour.

2.3 Action space and the zero lower bound (H&T eq. 11)

Stable-Baselines3 expects continuous-action spaces normalised to $[-1, 1]$ for stable training. We therefore expose the action space as $a_t \in [-1, 1]$ and map it inside the environment to $i_t = \frac{1}{2}(a_t + 1) \cdot i_{\max}$. This realises Hinterlang & Tanzer (2021, eq. 11) for the ZLB: the lower clip at $a = -1$ corresponds exactly to $i_t = 0$. The upper clip at $i_{\max} = 15\%$ is a maintained assumption that is non-binding over the 1987Q3-2007Q2 training sample.

2.4 Reward function (H&T eq. 16 and footnote 13)

We adopt the quadratic mandate reward of Hinterlang & Tanzer (2021, eq. 16):

$$r_t(x_t, i_t) = -\omega_\pi(\pi_{t+1} - \pi^*)^2 - \omega_y y_{t+1}^2,$$

with $\omega_\pi = \omega_y = 0.5$ and $\pi^* = 2\%$. Footnote 13 of Hinterlang & Tanzer (2021) augments this with a soft penalty when the squared deviations exceed 4 (i.e., when the absolute deviation exceeds 2 percentage points).

2.5 Episode design (H&T Section 2.2)

$M = 500$ episodes per architecture. Each episode is initialised from the empirical distribution of the training-sample data (1987Q3 to 2007Q2), then rolled forward for up to $T = 50$ quarters; the episode terminates early when the macro state reaches the band $1.7 < \pi_{t+1} < 2.3$ and $|y_{t+1}| < 0.3$. The discount factor is $\gamma = 0.99$ (Svensson, 2020).

```
[6]: # Hyper-parameters (Hinterlang & Tanzer 2021, Section 2)
PI_STAR = 2.0
OMEGA_PI = 0.5
OMEGA_Y = 0.5
OMEGA_DI = 0.5      # Sack-Wieland (2000) interest-rate smoothing weight on (i_t
    ↪ i_{t-1})^2.
                    # H&T (2021) eq. 16 omits this term, but their Table 7
    ↪ reports Var(Delta i)
                    # as a model-comparison metric. Without it, the agent learns
    ↪ a globally
                    # optimal bang-bang policy (i flips 0 <-> ACTION_HIGH each
    ↪ quarter) because
                    # the SVAR's coefficient on i_lag1 is small and only large
    ↪ rate swings
                    # exert any control. Standard reference: Woodford (2003, ch.
    ↪ 6), Svensson (2020).
GAMMA = 0.99
T_MAX = 50
ACTION_LOW = 0.0
ACTION_HIGH = 15.0
PENALTY_BAND = 2.0
PENALTY_WEIGHT = 10.0
STOP_PI_BAND = 0.3
STOP_Y_BAND = 0.3

# Macro-state safety clamp. The fitted environments (especially the NSSM)
    ↪ extrapolate
# poorly outside their training-sample range. We clip pi and y to a wide but
    ↪ finite
# band so a single divergent trajectory cannot drag the agent-selection filter
    ↪ to -inf.
# This is a downstream band-aid; the proper fix is to retrain the NSSM with
    ↪ stability
# constraints (Saxe-McClelland-Ganguli init, spectral regularization on W_h, or
    ↪ weight-tying
# to a contraction map). Set MACRO_CLAMP_PI = MACRO_CLAMP_Y = np.inf to disable.
MACRO_CLAMP_PI = 10.0    # |pi - PI_STAR| capped at 10 percentage points
MACRO_CLAMP_Y = 10.0    # |y| capped at 10 percentage points

# H&T agent-selection criteria
```

```

SELECT_ER_ES_MIN = -4.0          # H&T (2021) original threshold; kept for
    ↪documentation
SELECT_ER_ES_MIN_SEARCH = -50.0  # looser threshold for candidate saving during
    ↪training
SELECT_ES_MIN = 1
SELECT_ES_MAX = T_MAX + 1        # filter accepts ES in (1, T_MAX]; the old value 50
    ↪silently rejected truncated episodes (ES == T_MAX)

# Training budget per cell
M_EPISODES = 500
TOTAL_TIMESTEPS = 12_500

# Architecture-search grid
FULL_GRID = False
if FULL_GRID:
    CRITIC_NODE_GRID = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    ACTOR_NODE_GRID_NONLINEAR = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
else:
    CRITIC_NODE_GRID = (1, 2)
    ACTOR_NODE_GRID_NONLINEAR = (1, 4, 8)

SS_ROLLOUT_STEPS = 100
CANDIDATE_SAVE_EVERY = 5

Q_EPISODES = 2_000

# Reset seeds for this notebook (parent notebooks may have changed RNG state)
SEED = 2026
import random
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)

def cb_reward(pi_next: float, y_next: float, i_t: float = 0.0,
              i_prev: float = 0.0) -> float:
    """Central-bank reward: H&T (2021, eq. 16 + fn. 13) plus Sack-Wieland
    ↪smoothing.

    H&T eq. 16:           $r = -\omega_{\pi} (\pi - \pi^*)^2 - \omega_y y^2$ 
    H&T fn. 13:          additional  $-\text{PENALTY\_WEIGHT} * (\text{deviation})^2$  when  $|\text{dev}|$ 
    ↪ 2pp
    Sack-Wieland (2000): additional  $-\omega_{di} (i_t - i_{t-1})^2$  (set
    ↪  $\text{OMEGA\_DI}=0$  to disable)

```

The Sack-Wieland term is needed in practice because H&T eq. 16 alone is
↪ bang-bang
optimal when the underlying environment has weak interest-rate transmission
↪ (the SVAR
has only -0.05 on i_{lag1} in the y -equation), and bang-bang policies score
↪ well on the
mandate even though no real central bank would execute them.
"""

```

base = (-OMEGA_PI * (pi_next - PI_STAR) ** 2
        - OMEGA_Y * y_next ** 2
        - OMEGA_DI * (i_t - i_prev) ** 2)
penalty = 0.0
if abs(pi_next - PI_STAR) > PENALTY_BAND:
    penalty -= PENALTY_WEIGHT * (pi_next - PI_STAR) ** 2
if abs(y_next) > PENALTY_BAND:
    penalty -= PENALTY_WEIGHT * y_next ** 2
return float(base + penalty)

def is_terminal(pi_next: float, y_next: float) -> bool:
    return (
        abs(pi_next - PI_STAR) < STOP_PI_BAND
        and abs(y_next) < STOP_Y_BAND
    )

def normalised_to_rate(action: np.ndarray) -> float:
    a = float(np.asarray(action).reshape(-1)[0])
    a = float(np.clip(a, -1.0, 1.0))
    return 0.5 * (a + 1.0) * ACTION_HIGH

def rate_to_normalised(rate: float) -> np.ndarray:
    a = 2.0 * float(rate) / ACTION_HIGH - 1.0
    return np.array([np.clip(a, -1.0, 1.0)], dtype=np.float32)

```

3 Economy environments

3.1 Common environment interface

MonetaryEnv subclasses `gymnasium.Env` and supports both observation specifications via the `observation_lag` flag. Each concrete environment manages its own state buffers (lags of inflation, output gap, and the policy rate) and exposes a `reset` and `step` method compliant with Gymnasium (Towers et al., 2023).

Action-time convention. When the agent at period t selects action i_t , this rate enters the *next*

period's transition equation as the “one-quarter-lagged” rate $i_{t-1}^{(\text{next})}$, and the previous rate i_{t-1} becomes the two-quarter-lagged $i_{t-2}^{(\text{next})}$. This is the H&T recursive identification convention: the policy rate affects y and π with at least a one-period lag (Rotemberg & Woodford, 1997; Orphanides & Wieland, 2000).

```
[7]: class MonetaryEnv(gym.Env):
    """Base monetary-policy environment.

    Concrete subclasses implement `_transition`, which maps the current internal
    state and the chosen interest rate  $i_t$  to the next pair  $(\pi_{t+1}, y_{t+1})$ .
    """

    metadata = {"render_modes": []}

    def __init__(self, df_train: pd.DataFrame, *, observation_lag: bool = False,
                 seed: int = SEED):
        super().__init__()
        self.df_train = df_train.copy()
        self.observation_lag = observation_lag
        self.rng = np.random.default_rng(seed)

        self.action_space = spaces.Box(low=-1.0, high=1.0, shape=(1,), dtype=np.
float32)

        obs_dim = 4 if observation_lag else 2
        self.observation_space = spaces.Box(
            low=np.array([-10.0] * obs_dim, dtype=np.float32),
            high=np.array([10.0] * obs_dim, dtype=np.float32),
            dtype=np.float32,
        )

        self._pi_hist: List[float] = []
        self._y_hist: List[float] = []
        self._i_hist: List[float] = []
        self.t: int = 0

    @property
    def obs_dim(self) -> int:
        return 4 if self.observation_lag else 2

    def _sample_init_state(self) -> Dict[str, float]:
        idx = self.rng.integers(low=2, high=len(self.df_train))
        block = self.df_train.iloc[idx - 2 : idx]
        return {
            "pi_t": float(block["pi"].iloc[1]),
            "pi_tm1": float(block["pi"].iloc[0]),
            "y_t": float(block["y"].iloc[1]),
            "y_tm1": float(block["y"].iloc[0]),
        }
```

```

        "i_t": float(block["i"].iloc[1]),
        "i_tm1": float(block["i"].iloc[0]),
    }

def _init_buffers(self, init_state: Dict[str, float]) -> None:
    self._pi_hist = [init_state["pi_tm1"], init_state["pi_t"]]
    self._y_hist = [init_state["y_tm1"], init_state["y_t"]]
    self._i_hist = [init_state["i_tm1"], init_state["i_t"]]

def _push(self, pi_new: float, y_new: float, i_new: float) -> None:
    self._pi_hist.append(pi_new)
    self._y_hist.append(y_new)
    self._i_hist.append(i_new)
    self._pi_hist = self._pi_hist[-3:]
    self._y_hist = self._y_hist[-3:]
    self._i_hist = self._i_hist[-3:]

def _observation(self) -> np.ndarray:
    if self.observation_lag:
        obs = [self._y_hist[-1], self._y_hist[-2], self._pi_hist[-1], self._
↪_pi_hist[-2]]
    else:
        obs = [self._y_hist[-1], self._pi_hist[-1]]
    return np.asarray(obs, dtype=np.float32)

def reset(self, *, seed: Optional[int] = None,
           options: Optional[Dict[str, Any]] = None) -> Tuple[np.ndarray,
↪Dict[str, Any]]:
    if seed is not None:
        self.rng = np.random.default_rng(seed)
    self.t = 0
    init_state = (options or {}).get("init_state") if options else None
    if init_state is None:
        init_state = self._sample_init_state()
    self._init_buffers(init_state)
    return self._observation(), {}

def _transition(self, i_t: float) -> Tuple[float, float]:
    raise NotImplementedError

def step(self, action: np.ndarray) -> Tuple[np.ndarray, float, bool, bool,
↪Dict[str, Any]]:
    i_t = float(normalised_to_rate(action))
    # NOTE: _transition returns (y_new, pi_new) to match the SVAR ordering
↪used in each
    # subclass. Unpacking in that order is the canonical fix for the swap
↪bug that

```

```

        # destabilized the SVAR and NSSM closed loops.
        # Capture the previous rate before pushing the new one, so the
    ↪smoothing term in
        # cb_reward sees the correct (i_t, i_{t-1}) pair.
        i_prev = self._i_hist[-1] if self._i_hist else i_t
        y_next, pi_next = self._transition(i_t)
        # Defensive macro-state clamp. Bounds the trajectory so the
    ↪agent-selection callback
        # is not dominated by a single divergent excursion. See cell 17 for
    ↪MACRO_CLAMP_*.
        pi_next = float(np.clip(pi_next, PI_STAR - MACRO_CLAMP_PI, PI_STAR +
    ↪MACRO_CLAMP_PI))
        y_next = float(np.clip(y_next, -MACRO_CLAMP_Y, MACRO_CLAMP_Y))
        self._push(pi_next, y_next, i_t)
        reward = cb_reward(pi_next, y_next, i_t, i_prev)
        self.t += 1
        terminated = bool(is_terminal(pi_next, y_next))
        truncated = bool(self.t >= T_MAX)
        info = {"pi": pi_next, "y": y_next, "i": i_t}
        return self._observation(), reward, terminated, truncated, info

```

3.2 Linear environment: restricted recursive SVAR(2)

Wraps the fitted svar object from the linear-environment notebook. The OLS coefficient names (svar.mod_y.params.index, svar.mod_pi.params.index) are looked up dynamically so the environment adapts to whatever lag restriction the parent notebook applied. The structural shocks use the OLS residual standard deviations as their innovation variance.

```

[8]: # Extract OLS residual sigmas from the fitted SVAR
_svar_sigma_y = float(np.std(svar.mod_y.resid, ddof=int(svar.mod_y.df_model) +
    ↪1))
_svar_sigma_pi = float(np.std(svar.mod_pi.resid, ddof=int(svar.mod_pi.df_model)
    ↪+ 1))

class SVAREnv(MonetaryEnv):
    """Linear monetary-policy environment driven by the parent notebook's SVAR.
    ↪"""

    def __init__(self, df_train: pd.DataFrame, *, observation_lag: bool = False,
                  seed: int = SEED):
        super().__init__(df_train, observation_lag=observation_lag, seed=seed)
        self.params_y = svar.mod_y.params # pandas Series
        self.params_pi = svar.mod_pi.params
        self.sigma_y = _svar_sigma_y
        self.sigma_pi = _svar_sigma_pi

```

```

def _transition(self, i_t: float) -> Tuple[float, float]:
    # Build the regressor dictionary for predicting  $y_{t+1}$  and  $\pi_{t+1}$ .
    # Mapping convention: the agent's action  $i_t$  becomes  $i_{lag1}$  for the
    # *next* period (one quarter back from  $t+1$ ), and the buffer's most
    # recent rate  $i_{t-1}$  becomes  $i_{lag2}$  (two quarters back from  $t+1$ ).
    y_lag1 = self._y_hist[-1]      # =  $y_t$ 
    y_lag2 = self._y_hist[-2]      # =  $y_{t-1}$ 
    pi_lag1 = self._pi_hist[-1]    # =  $\pi_t$ 
    pi_lag2 = self._pi_hist[-2]    # =  $\pi_{t-1}$ 
    i_lag1 = i_t                   # action becomes lag-1
    i_lag2 = self._i_hist[-1]      # previous rate becomes lag-2

    eps_y = float(self.rng.normal(scale=self.sigma_y))
    eps_pi = float(self.rng.normal(scale=self.sigma_pi))

    # y-equation: linear combination over whichever regressors the parent
    ↪ fit kept
    all_vals_y = {
        "const": 1.0,
        "y_lag1": y_lag1, "y_lag2": y_lag2,
        "pi_lag1": pi_lag1, "pi_lag2": pi_lag2,
        "i_lag1": i_lag1, "i_lag2": i_lag2,
    }
    y_new = sum(float(self.params_y[k]) * all_vals_y.get(k, 0.0)
                 for k in self.params_y.index) + eps_y

    # pi-equation: recursive identification places contemporaneous  $y_{t+1}$ 
    # in the inflation regressor set.
    all_vals_pi = {
        "const": 1.0,
        "y": y_new,
        "y_lag1": y_lag1, "y_lag2": y_lag2,
        "pi_lag1": pi_lag1, "pi_lag2": pi_lag2,
        "i_lag1": i_lag1, "i_lag2": i_lag2,
    }
    pi_new = sum(float(self.params_pi[k]) * all_vals_pi.get(k, 0.0)
                 for k in self.params_pi.index) + eps_pi

    return float(y_new), float(pi_new)

check_env(SVAREnv(df), warn=True, skip_render_check=True)
print(f"SVAREnv passes the SB3 env-checker.")
print(f"  sigma_y = {_svar_sigma_y:.4f}")
print(f"  sigma_pi = {_svar_sigma_pi:.4f}")

```

```

SVAREnv passes the SB3 env-checker.
sigma_y = 0.4814

```

```
sigma_pi = 0.1849
```

3.3 Modern linear environment: TVP-SVAR-SV

Wraps the PyMC posteriors `idata_y` and `idata_pi` from the linear-environment notebook. For each RL rollout we anchor the time-varying coefficient state at the end-of-training-sample posterior mean (i.e., the smoothed coefficient at 2007Q2), following the counterfactual design of Primiceri (2005, Section 4.4). The stochastic-volatility process is anchored at the same time slice.

```
[9]: # Identify the index for 2007Q2 in the TVP posteriors so we can anchor at
      ↪end-of-training
      _tvp_time_coord = idata_y.posterior["beta"].coords["time"].values # integer
      ↪index
      # The TVP design is built on idx_tvp (a PeriodIndex) defined in the linear
      ↪notebook.
      _period_end = pd.Period("2007Q2", freq="Q")
      _tvp_end_idx = idx_tvp.get_loc(_period_end)

      # Posterior means at the training endpoint
      _B_y_T = idata_y.posterior["beta"].mean(("chain", "draw")).
      ↪isel(time=_tvp_end_idx).values
      _B_pi_T = idata_pi.posterior["beta"].mean(("chain", "draw")).
      ↪isel(time=_tvp_end_idx).values
      _sigma_y_T = float(
          idata_y.posterior["sigma_obs"].mean(("chain", "draw")).
          ↪isel(time=_tvp_end_idx).values
      )
      _sigma_pi_T = float(
          idata_pi.posterior["sigma_obs"].mean(("chain", "draw")).
          ↪isel(time=_tvp_end_idx).values
      )

      # Regressor names in the TVP design (from the linear notebook's
      ↪make_recursive_svar_design)
      _TVP_REGS_Y = list(regs_y)
      _TVP_REGS_PI = list(regs_pi)
      print("TVP anchored at end-of-training-sample posterior means")
      print(f"  beta_y(T)   shape = {_B_y_T.shape}, sigma_y(T) = {_sigma_y_T:.4f}")
      print(f"  beta_pi(T)  shape = {_B_pi_T.shape}, sigma_pi(T) = {_sigma_pi_T:.4f}")

      class TVPSVARSEnv(MonetaryEnv):
          """TVP-SVAR-SV environment anchored at the training-endpoint posterior mean.
          ↪"""

          def __init__(self, df_train, *, observation_lag: bool = False, seed: int =
          ↪SEED):
```

```

super().__init__(df_train, observation_lag=observation_lag, seed=seed)
self.B_y = _B_y_T
self.B_pi = _B_pi_T
self.sigma_y = _sigma_y_T
self.sigma_pi = _sigma_pi_T
self.regs_y = _TVP_REGS_Y
self.regs_pi = _TVP_REGS_PI

def _transition(self, i_t: float) -> Tuple[float, float]:
    y_lag1 = self._y_hist[-1]
    y_lag2 = self._y_hist[-2]
    pi_lag1 = self._pi_hist[-1]
    pi_lag2 = self._pi_hist[-2]
    i_lag1 = i_t # action becomes lag-1 of next period
    i_lag2 = self._i_hist[-1] # previous rate becomes lag-2

    eps_y = float(self.rng.normal(scale=self.sigma_y))
    eps_pi = float(self.rng.normal(scale=self.sigma_pi))

    vals_y = {
        "const": 1.0, "y_lag1": y_lag1, "y_lag2": y_lag2,
        "pi_lag1": pi_lag1, "pi_lag2": pi_lag2,
        "i_lag1": i_lag1, "i_lag2": i_lag2,
    }
    row_y = np.array([vals_y[n] for n in self.regs_y])
    y_new = float(self.B_y @ row_y) + eps_y

    vals_pi = {
        "const": 1.0, "y": y_new, "y_lag1": y_lag1, "y_lag2": y_lag2,
        "pi_lag1": pi_lag1, "pi_lag2": pi_lag2,
        "i_lag1": i_lag1, "i_lag2": i_lag2,
    }
    row_pi = np.array([vals_pi[n] for n in self.regs_pi])
    pi_new = float(self.B_pi @ row_pi) + eps_pi

    return y_new, pi_new

check_env(TVPSVARSEnv(df), warn=True, skip_render_check=True)
print("TVPSVARSEnv passes the SB3 env-checker.")

```

TVP anchored at end-of-training-sample posterior means

beta_y(T) shape = (7,), sigma_y(T) = 0.1909

beta_pi(T) shape = (8,), sigma_pi(T) = 0.0891

TVPSVARSEnv passes the SB3 env-checker.

3.4 Nonlinear environment: NARX

Wraps `final_narx_y` and `final_narx_pi` (the fitted NARX networks from the non-linear-environment notebook) along with their `MapMinMaxScaler` instances. The non-linear notebook fits the NARX with the H&T (eq. 8) restricted input specifications: $s_t^y = (y_{t-1}, \pi_{t-1}, i_{t-1})$ for the output-gap equation and $s_t^\pi = (y_t, y_{t-1}, y_{t-2}, \pi_{t-1}, \pi_{t-2}, i_{t-1})$ for the inflation equation. The innovation standard deviations are taken from the training-sample residuals stored in `fit_narx`.

```
[10]: # Innovation sds from the fitted NARX residuals (parent notebook's fit_narx_
↳ DataFrame)
_narx_sigma_y = float(fit_narx["resid_y_narx"].std(ddof=1))
_narx_sigma_pi = float(fit_narx["resid_pi_narx"].std(ddof=1))

class NARXEnv(MonetaryEnv):
    """Non-linear monetary-policy environment driven by the parent notebook's_
    ↳ NARX. """

    def __init__(self, df_train, *, observation_lag: bool = False, seed: int =_
    ↳ SEED):
        super().__init__(df_train, observation_lag=observation_lag, seed=seed)
        self.narx_y = final_narx_y.eval()
        self.narx_pi = final_narx_pi.eval()
        self.xs_y = x_scaler_y
        self.ys_y = y_scaler_y
        self.xs_pi = x_scaler_pi
        self.ys_pi = y_scaler_pi
        self.sigma_y = _narx_sigma_y
        self.sigma_pi = _narx_sigma_pi
        # Use the parent notebook's DTYPE and DEVICE if available, else default
        self._dtype = globals().get("DTYPE", torch.float64)
        self._device = globals().get("DEVICE", torch.device("cpu"))

    @torch.no_grad()
    def _transition(self, i_t: float) -> Tuple[float, float]:
        y_lag1 = self._y_hist[-1]
        y_lag2 = self._y_hist[-2]
        pi_lag1 = self._pi_hist[-1]
        pi_lag2 = self._pi_hist[-2]
        i_lag1 = i_t # action becomes lag-1 in the NARX's input ordering

        # y-equation: 3 inputs (y_lag1, pi_lag1, i_lag1)
        feat_y = torch.tensor([y_lag1, pi_lag1, i_lag1], dtype=self._dtype,
        ↳ device=self._device)
        feat_y_sc = self.xs_y.transform(feat_y)
        y_sc = self.narx_y(feat_y_sc)
        y_mean = float(self.ys_y.inverse_transform(y_sc).reshape(-1)[0].item())
        y_new = y_mean + float(self.rng.normal(scale=self.sigma_y))
```

```

        # pi-equation: 6 inputs (y, y_lag1, y_lag2, pi_lag1, pi_lag2, i_lag1)
        feat_pi = torch.tensor([[y_new, y_lag1, y_lag2, pi_lag1, pi_lag2,
↪ i_lag1]],
                                dtype=self._dtype, device=self._device)
        feat_pi_sc = self.xs_pi.transform(feat_pi)
        pi_sc = self.narx_pi(feat_pi_sc)
        pi_mean = float(self.ys_pi.inverse_transform(pi_sc).reshape(-1)[0].
↪ item())
        pi_new = pi_mean + float(self.rng.normal(scale=self.sigma_pi))

        return y_new, pi_new

check_env(NARXEnv(df), warn=True, skip_render_check=True)
print("NARXEnv passes the SB3 env-checker.")
print(f"    sigma_y = {_narx_sigma_y:.4f}")
print(f"    sigma_pi = {_narx_sigma_pi:.4f}")

```

NARXEnv passes the SB3 env-checker.

```

sigma_y = 0.4566
sigma_pi = 0.1595

```

3.5 Nonlinear environment: Neural state-space model (NSSM)

Wraps the trained `nssm` (TwoBlockNSSM) from the non-linear-environment notebook and its terminal latent states. The non-linear notebook uses the *full-information* input specification:

- output-gap block: $s_t^y = (y_{t-1}, y_{t-2}, \pi_{t-1}, \pi_{t-2}, i_{t-1}, i_{t-2})$
- inflation block: $s_t^\pi = (y_t, y_{t-1}, y_{t-2}, \pi_{t-1}, \pi_{t-2}, i_{t-1}, i_{t-2})$

For the RL environment we anchor the latent state at the terminal in-sample value (the last row of `states_y_nssm` and `states_pi_nssm`) and step the two blocks separately so the recursive identification (predicted y_{t+1} feeding into the inflation block) is preserved during rollouts.

```

[11]: # Innovation sds from the fitted NSSM residuals (parent notebook's fit_nssm
↪ DataFrame)
_nssm_sigma_y = float(fit_nssm["resid_y_nssm"].std(ddof=1))
_nssm_sigma_pi = float(fit_nssm["resid_pi_nssm"].std(ddof=1))

# Terminal latent states (last row of the in-sample state trajectories)
_h_y_T_nssm = states_y_nssm[-1:].clone()
_h_pi_T_nssm = states_pi_nssm[-1:].clone()

class NSSMEnv(MonetaryEnv):
    """Non-linear monetary-policy environment driven by the parent notebook's
↪ NSSM. """

```

```

def __init__(self, df_train, *, observation_lag: bool = False, seed: int = SEED):
    super().__init__(df_train, observation_lag=observation_lag, seed=seed)
    self.model = nssm.eval()
    self._h_y_init = _h_y_T_nssm.detach().clone()
    self._h_pi_init = _h_pi_T_nssm.detach().clone()
    self._h_y = self._h_y_init.clone()
    self._h_pi = self._h_pi_init.clone()
    self.sigma_y = _nssm_sigma_y
    self.sigma_pi = _nssm_sigma_pi
    self._dtype = globals().get("DTYPE", torch.float64)
    self._device = globals().get("DEVICE", torch.device("cpu"))

def reset(self, *, seed=None, options=None):
    obs, info = super().reset(seed=seed, options=options)
    self._h_y = self._h_y_init.clone()
    self._h_pi = self._h_pi_init.clone()
    return obs, info

@torch.no_grad()
def _transition(self, i_t: float) -> Tuple[float, float]:
    y_lag1 = self._y_hist[-1]
    y_lag2 = self._y_hist[-2]
    pi_lag1 = self._pi_hist[-1]
    pi_lag2 = self._pi_hist[-2]
    i_lag1 = i_t
    i_lag2 = self._i_hist[-1]
    # action becomes lag-1 of next period
    # previous rate becomes lag-2

    # Step the output-gap block first
    x_y = torch.tensor(
        [[y_lag1, y_lag2, pi_lag1, pi_lag2, i_lag1, i_lag2]],
        dtype=self._dtype, device=self._device,
    )
    h_y_new = self.model.activation(
        self.model.W_h_y(self._h_y) + self.model.W_x_y(x_y)
    )
    y_pred = float(self.model.W_o_y(h_y_new).reshape(-1)[0].item())
    y_new = y_pred + float(self.rng.normal(scale=self.sigma_y))

    # Step the inflation block with the predicted contemporaneous y_{t+1}
    x_pi = torch.tensor(
        [[y_new, y_lag1, y_lag2, pi_lag1, pi_lag2, i_lag1, i_lag2]],
        dtype=self._dtype, device=self._device,
    )
    h_pi_new = self.model.activation(
        self.model.W_h_pi(self._h_pi) + self.model.W_x_pi(x_pi)
    )

```

```

pi_pred = float(self.model.W_o_pi(h_pi_new).reshape(-1)[0].item())
pi_new = pi_pred + float(self.rng.normal(scale=self.sigma_pi))

self._h_y = h_y_new.detach()
self._h_pi = h_pi_new.detach()

return y_new, pi_new

check_env(NSSMEnv(df), warn=True, skip_render_check=True)
print("NSSMEnv passes the SB3 env-checker.")
print(f" sigma_y = {_nssm_sigma_y:.4f}")
print(f" sigma_pi = {_nssm_sigma_pi:.4f}")

```

```

NSSMEnv passes the SB3 env-checker.
sigma_y = 0.4539
sigma_pi = 0.6255

```

4 Custom SB3 actor classes for H&T-faithful DDPG

Hinterlang & Tanzer (2021, eq. 12-13) parameterise the actor as

$$P_t = i_t = \max\{f(x_t^z), 0\}, \quad f(x_t^z) = \alpha_0 + \sum_{j=1}^q \delta_j G(\beta'_j x_t^z + \alpha_j),$$

with $G = \text{purelin}$ and $q = 1$ for the linear variant and $G = \tanh$ for the nonlinear variant. SB3's default `MlpPolicy` for DDPG/TD3 applies an additional $\tanh$ squash on the actor output that is *not* part of the H&T policy specification. We override `TD3Actor` to skip it; the ZLB ReLU of eq. 12 is realised by the action-space clipping at $a = -1$ (which maps to $i_t = 0$).

```

[12]: class LinearActor(TD3Actor):
        """H&T (2021) linear actor:  $f(x) = \alpha_0 + \beta' x$ , no tanh squash."""

        def __init__(self, *args: Any, **kwargs: Any) -> None:
            kwargs["net_arch"] = []
            super().__init__(*args, **kwargs)
            action_dim = get_action_dim(self.action_space)
            self.mu = nn.Linear(self.features_dim, action_dim)

        def forward(self, obs: torch.Tensor) -> torch.Tensor:
            features = self.extract_features(obs, self.features_extractor)
            return self.mu(features)

class NonlinearActor(TD3Actor):

```

```

"""H&T (2021) nonlinear actor:  $f(x) = \alpha_0 + \sum \delta_j \tanh(\beta_j' x + \alpha_j)$ ."""

```

```

def __init__(self, *args: Any, **kwargs: Any) -> None:
    super().__init__(*args, **kwargs)
    action_dim = get_action_dim(self.action_space)
    layers = create_mlp(
        input_dim=self.features_dim,
        output_dim=action_dim,
        net_arch=list(self.net_arch),
        activation_fn=nn.Tanh,
        squash_output=False,
    )
    self.mu = nn.Sequential(*layers)

def forward(self, obs: torch.Tensor) -> torch.Tensor:
    features = self.extract_features(obs, self.features_extractor)
    return self.mu(features)

```

```

class LinearTD3Policy(TD3Policy):
    def make_actor(self, features_extractor=None):
        actor_kwargs = self._update_features_extractor(self.actor_kwargs,
        features_extractor)
        actor_kwargs["net_arch"] = []
        return LinearActor(**actor_kwargs).to(self.device)

```

```

class NonlinearTD3Policy(TD3Policy):
    def make_actor(self, features_extractor=None):
        actor_kwargs = self._update_features_extractor(self.actor_kwargs,
        features_extractor)
        return NonlinearActor(**actor_kwargs).to(self.device)

```

5 Architecture search and agent selection (H&T Section 2.2)

During training we save every agent whose episode statistics satisfy $ER_m/ES_m > -4$ and $1 < ES_m < 50$. After training, we compute the steady-state reward of each saved agent and select the one with the highest steady-state reward.

```

[13]: @dataclass
class AgentCandidate:
    state_dict: Dict[str, torch.Tensor]
    episode: int
    ER: float

```

```

ES: int

class HTAgentSelectionCallback(BaseCallback):
    """SB3 callback implementing the H&T (2021) agent-selection criterion."""

    def __init__(self, *, save_every_qualifying: int = CANDIDATE_SAVE_EVERY,
                 verbose: int = 0) -> None:
        super().__init__(verbose)
        self.save_every_qualifying = save_every_qualifying
        self.candidates: List[AgentCandidate] = []
        self.episode_rewards: List[float] = []
        self._current_ER: float = 0.0
        self._current_ES: int = 0
        self._episodes_seen: int = 0
        self._qualifying_seen: int = 0

    def _on_step(self) -> bool:
        rewards = self.locals.get("rewards", [0.0])
        dones = self.locals.get("dones", [False])
        self._current_ER += float(rewards[0])
        self._current_ES += 1

        if bool(dones[0]):
            ER, ES = self._current_ER, self._current_ES
            self.episode_rewards.append(ER)
            self._episodes_seen += 1
            ER_per_ES = ER / max(ES, 1)
            # Save against the looser SEARCH bound during training; the
            ↪ steady-state-reward
            # tiebreaker in `select_best_candidate` then picks the best
            ↪ surviving checkpoint.
            # H&T's strict -4 bound is preserved as documentation; relaxing it
            ↪ here lets
            # the NSSM (which extrapolates badly) accumulate any candidates at
            ↪ all.

            qualifies = (
                ER_per_ES > SELECT_ER_ES_MIN_SEARCH
                and SELECT_ES_MIN < ES < SELECT_ES_MAX
            )
            if qualifies:
                self._qualifying_seen += 1
                if self._qualifying_seen % self.save_every_qualifying == 0:
                    state_dict = {
                        k: v.detach().cpu().clone()
                        for k, v in self.model.policy.state_dict().items()
                    }

```

```

        self.candidates.append(AgentCandidate(
            state_dict=state_dict,
            episode=self._episodes_seen,
            ER=ER, ES=ES,
        ))
        self._current_ER = 0.0
        self._current_ES = 0
    return True

def steady_state_reward(model: Any, env: MonetaryEnv,
                        n_steps: int = SS_ROLLOUT_STEPS,
                        init_state: Optional[Dict[str, float]] = None) -> float:
    """Run the model forward and average the per-step reward."""
    obs, _ = env.reset(options={"init_state": init_state} if init_state else
↪None)
    rewards: List[float] = []
    for _ in range(n_steps):
        action, _ = model.predict(obs, deterministic=True)
        obs2, r, term, trunc, info = env.step(action)
        rewards.append(float(r))
        obs = obs2
        if term or trunc:
            obs, _ = env.reset(options={"init_state": init_state} if init_state
↪else None)
    return float(np.mean(rewards))

def select_best_candidate(model: Any, candidates: List[AgentCandidate],
                          env_factory: Callable[[], MonetaryEnv],
                          init_state: Optional[Dict[str, float]] = None
                          ) -> Tuple[Optional[AgentCandidate], float]:
    if not candidates:
        return None, -np.inf
    best_reward = -np.inf
    best_cand: Optional[AgentCandidate] = None
    for cand in candidates:
        model.policy.load_state_dict(cand.state_dict)
        env = env_factory()
        ss = steady_state_reward(model, env, init_state=init_state)
        if ss > best_reward:
            best_reward = ss
            best_cand = cand
    if best_cand is not None:
        model.policy.load_state_dict(best_cand.state_dict)
    return best_cand, best_reward

```

6 Tabular Q-learning (custom)

SB3 does not ship a tabular Q-learning agent. We implement Watkins' (1989) algorithm directly; Wang et al. (2026) report that on similar single-CB monetary MDPs tabular Q-learning outperforms DQN and Bayesian variants. Restricted to the no-lag observation $x_t^{(1)}$ to keep the table size tractable.

```
[14]: @dataclass
class QLearningAgent:
    obs_low: np.ndarray
    obs_high: np.ndarray
    n_bins: Tuple[int, int]
    action_grid: np.ndarray
    Q: np.ndarray
    alpha: float = 0.2
    gamma: float = GAMMA
    epsilon: float = 0.2
    rng: np.random.Generator = field(default_factory=lambda: np.random.
    ↪default_rng(SEED))

    @classmethod
    def build(cls, *, n_bins: Tuple[int, int] = (6, 7),
              action_grid: Sequence[float] = tuple(np.arange(0.0, 10.5, 0.5)),
              obs_low: Optional[np.ndarray] = None,
              obs_high: Optional[np.ndarray] = None,
              alpha: float = 0.2, epsilon: float = 0.2,
              seed: int = SEED) -> "QLearningAgent":
        if obs_low is None:
            obs_low = np.array([-6.0, -2.0])
        if obs_high is None:
            obs_high = np.array([6.0, 6.0])
        return cls(
            obs_low=obs_low, obs_high=obs_high, n_bins=n_bins,
            action_grid=np.asarray(action_grid, dtype=np.float64),
            Q=np.zeros((*n_bins, len(action_grid)), dtype=np.float64),
            alpha=alpha, epsilon=epsilon, rng=np.random.default_rng(seed),
        )

    def discretise(self, obs: np.ndarray) -> Tuple[int, int]:
        idx = []
        for k in range(2):
            lo, hi = self.obs_low[k], self.obs_high[k]
            x = (float(obs[k]) - lo) / (hi - lo)
            b = int(np.clip(np.floor(x * self.n_bins[k]), 0, self.n_bins[k] - 1
            ↪1))
```

```

        idx.append(b)
    return tuple(idx)

def action_index(self, value: float) -> int:
    return int(np.argmax(np.abs(self.action_grid - value)))

def act(self, obs: np.ndarray, explore: bool = True) -> float:
    s = self.discretise(obs)
    if explore and self.rng.random() < self.epsilon:
        return float(self.rng.choice(self.action_grid))
    a_idx = int(np.argmax(self.Q[s]))
    return float(self.action_grid[a_idx])

def update(self, obs, action, reward, next_obs, done):
    s = self.discretise(obs)
    s2 = self.discretise(next_obs)
    a_idx = self.action_index(action)
    target = reward + (0.0 if done else self.gamma * float(self.Q[s2].
↪max()))
    self.Q[(s, a_idx)] += self.alpha * (target - self.Q[(s, a_idx)])

```

7 Running the simulations

7.1 Experimental design

DDPG receives the full H&T grid: four environments times two observation specs times two policy variants (linear, nonlinear), with hidden-node search over the critic nodes and (for nonlinear) the actor nodes, and the agent-selection callback. SAC is trained with a fixed [64, 64] architecture and the same agent-selection callback. Q-learning is restricted to $x_t^{(1)}$ only.

```

[15]: def make_env(env_name: str, *, observation_lag: bool, seed: int = SEED) -> ↵
    ↪MonetaryEnv:
    if env_name == "SVAR":
        return SVAREnv(df, observation_lag=observation_lag, seed=seed)
    if env_name == "TVP-SVAR-SV":
        return TVPSVARSEnv(df, observation_lag=observation_lag, seed=seed)
    if env_name == "NARX":
        return NARXEnv(df, observation_lag=observation_lag, seed=seed)
    if env_name == "NSSM":
        return NSSMEnv(df, observation_lag=observation_lag, seed=seed)
    raise ValueError(env_name)

ENV_NAMES = ["SVAR", "TVP-SVAR-SV", "NARX", "NSSM"]
OBS_SPECS = ["x1", "x2"]

```

```

POLICY_VARIANTS = ["linear", "nonlinear"]

def obs_spec_to_lag(obs_spec: str) -> bool:
    return obs_spec == "x2"

def initial_state_from_data() -> Dict[str, float]:
    return {
        "pi_t": float(df["pi"].iloc[1]),
        "pi_tm1": float(df["pi"].iloc[0]),
        "y_t": float(df["y"].iloc[1]),
        "y_tm1": float(df["y"].iloc[0]),
        "i_t": float(df["i"].iloc[1]),
        "i_tm1": float(df["i"].iloc[0]),
    }

@dataclass
class TrainedCell:
    algo: str
    env_name: str
    obs_spec: str
    policy_variant: Optional[str]
    critic_nodes: Optional[int]
    actor_nodes: Optional[int]
    steady_state_reward: float
    model: Any
    training_rewards: List[float]

trained_cells: Dict[Tuple[str, str, str, Optional[str]], TrainedCell] = {}

```

7.2 DDPG training with architecture search

```

[16]: def train_ddpg_cell(env_name: str, obs_spec: str, policy_variant: str) -> TrainedCell:
    obs_lag = obs_spec_to_lag(obs_spec)
    env_factory = lambda: make_env(env_name, observation_lag=obs_lag, seed=SEED)
    init = initial_state_from_data()

    if policy_variant == "linear":
        actor_grid = (1,)
        policy_class = LinearTD3Policy
    else:
        actor_grid = tuple(ACTOR_NODE_GRID_NONLINEAR)
        policy_class = NonlinearTD3Policy

```

```

best_overall: Optional[TrainedCell] = None

for critic_nodes in CRITIC_NODE_GRID:
    for actor_nodes in actor_grid:
        env = env_factory()
        n_actions = env.action_space.shape[-1]
        action_noise = NormalActionNoise(
            mean=np.zeros(n_actions), sigma=0.3 * np.ones(n_actions),
        )
        policy_kwargs = dict(net_arch=dict(
            pi=[] if policy_variant == "linear" else [actor_nodes],
            qf=[critic_nodes],
        ))
        model = DDPG(
            policy_class, env,
            action_noise=action_noise,
            policy_kwargs=policy_kwargs,
            learning_starts=500, batch_size=64,
            tau=0.005, gamma=GAMMA, verbose=0, seed=SEED,
        )
        callback = HTAgentSelectionCallback()
        model.learn(total_timesteps=TOTAL_TIMESTEPS, callback=callback,
↪progress_bar=False)

        _, ss_reward = select_best_candidate(model, callback.candidates,
↪env_factory, init_state=init)
        if best_overall is None or ss_reward > best_overall:
↪steady_state_reward:
            best_overall = TrainedCell(
                algo="DDPG", env_name=env_name, obs_spec=obs_spec,
                policy_variant=policy_variant,
                critic_nodes=critic_nodes,
                actor_nodes=actor_nodes if policy_variant == "nonlinear"
↪else 1,

                steady_state_reward=ss_reward,
                model=model,
                training_rewards=list(callback.episode_rewards),
            )
            print(f"    critic={critic_nodes}, actor={actor_nodes if
↪policy_variant == 'nonlinear' else '-'} "
                f"-> SS reward = {ss_reward:.4f} ({len(callback.candidates)}
↪candidates)")

        assert best_overall is not None
        return best_overall

```

```

print("=" * 72)
print("DDPG: training the full H&T (2021) grid")
print("=" * 72)
for env_name in ENV_NAMES:
    for obs_spec in OBS_SPECS:
        for policy_variant in POLICY_VARIANTS:
            key = ("DDPG", env_name, obs_spec, policy_variant)
            print(f"\n[DDPG] env={env_name} obs={obs_spec} ↵
↳policy={policy_variant}")
            cell = train_ddpg_cell(env_name, obs_spec, policy_variant)
            trained_cells[key] = cell
            print(f" >> winner: critic={cell.critic_nodes}, actor={cell.
↳actor_nodes}, "
                  f"SS reward = {cell.steady_state_reward:.4f}")

```

```

=====
DDPG: training the full H&T (2021) grid
=====

```

```

[DDPG] env=SVAR obs=x1 policy=linear
    critic=1, actor=-> SS reward = -540.3727 (5 candidates)
    critic=2, actor=-> SS reward = -11.7005 (74 candidates)
>> winner: critic=2, actor=1, SS reward = -11.7005

```

```

[DDPG] env=SVAR obs=x1 policy=nonlinear
    critic=1, actor=1 -> SS reward = -17.9273 (60 candidates)
    critic=1, actor=4 -> SS reward = -63.0047 (9 candidates)
    critic=1, actor=8 -> SS reward = -20.2369 (46 candidates)
    critic=2, actor=1 -> SS reward = -6.1247 (42 candidates)
    critic=2, actor=4 -> SS reward = -16.4107 (53 candidates)
    critic=2, actor=8 -> SS reward = -23.5051 (12 candidates)
>> winner: critic=2, actor=1, SS reward = -6.1247

```

```

[DDPG] env=SVAR obs=x2 policy=linear
    critic=1, actor=-> SS reward = -13.3254 (24 candidates)
    critic=2, actor=-> SS reward = -8.0580 (60 candidates)
>> winner: critic=2, actor=1, SS reward = -8.0580

```

```

[DDPG] env=SVAR obs=x2 policy=nonlinear
    critic=1, actor=1 -> SS reward = -20.1725 (94 candidates)
    critic=1, actor=4 -> SS reward = -22.4655 (109 candidates)
    critic=1, actor=8 -> SS reward = -31.4319 (13 candidates)
    critic=2, actor=1 -> SS reward = -15.4297 (113 candidates)
    critic=2, actor=4 -> SS reward = -20.4355 (92 candidates)
    critic=2, actor=8 -> SS reward = -28.8771 (13 candidates)

```

```

>> winner: critic=2, actor=1, SS reward = -15.4297

[DDPG] env=TVP-SVAR-SV obs=x1 policy=linear
  critic=1, actor=-> SS reward = -inf (0 candidates)
  critic=2, actor=-> SS reward = -inf (0 candidates)
>> winner: critic=1, actor=1, SS reward = -inf

[DDPG] env=TVP-SVAR-SV obs=x1 policy=nonlinear
  critic=1, actor=1 -> SS reward = -0.9636 (2 candidates)
  critic=1, actor=4 -> SS reward = -inf (0 candidates)
  critic=1, actor=8 -> SS reward = -inf (0 candidates)
  critic=2, actor=1 -> SS reward = -0.5186 (1 candidates)
  critic=2, actor=4 -> SS reward = -inf (0 candidates)
  critic=2, actor=8 -> SS reward = -inf (0 candidates)
>> winner: critic=2, actor=1, SS reward = -0.5186

[DDPG] env=TVP-SVAR-SV obs=x2 policy=linear
  critic=1, actor=-> SS reward = -inf (0 candidates)
  critic=2, actor=-> SS reward = -inf (0 candidates)
>> winner: critic=1, actor=1, SS reward = -inf

[DDPG] env=TVP-SVAR-SV obs=x2 policy=nonlinear
  critic=1, actor=1 -> SS reward = -inf (0 candidates)
  critic=1, actor=4 -> SS reward = -inf (0 candidates)
  critic=1, actor=8 -> SS reward = -inf (0 candidates)
  critic=2, actor=1 -> SS reward = -inf (0 candidates)
  critic=2, actor=4 -> SS reward = -inf (0 candidates)
  critic=2, actor=8 -> SS reward = -inf (0 candidates)
>> winner: critic=1, actor=1, SS reward = -inf

[DDPG] env=NARX obs=x1 policy=linear
  critic=1, actor=-> SS reward = -13.7492 (53 candidates)
  critic=2, actor=-> SS reward = -10.9335 (53 candidates)
>> winner: critic=2, actor=1, SS reward = -10.9335

[DDPG] env=NARX obs=x1 policy=nonlinear
  critic=1, actor=1 -> SS reward = -6.3512 (67 candidates)
  critic=1, actor=4 -> SS reward = -13.6631 (54 candidates)
  critic=1, actor=8 -> SS reward = -7.0482 (53 candidates)
  critic=2, actor=1 -> SS reward = -6.2105 (54 candidates)
  critic=2, actor=4 -> SS reward = -13.7492 (54 candidates)
  critic=2, actor=8 -> SS reward = -7.0482 (54 candidates)
>> winner: critic=2, actor=1, SS reward = -6.2105

[DDPG] env=NARX obs=x2 policy=linear
  critic=1, actor=-> SS reward = -5.5296 (59 candidates)
  critic=2, actor=-> SS reward = -6.7906 (75 candidates)
>> winner: critic=1, actor=1, SS reward = -5.5296

```

```
[DDPG] env=NARX obs=x2 policy=nonlinear
critic=1, actor=1 -> SS reward = -4.6647 (54 candidates)
critic=1, actor=4 -> SS reward = -4.3932 (53 candidates)
critic=1, actor=8 -> SS reward = -9.9837 (60 candidates)
critic=2, actor=1 -> SS reward = -19.0662 (54 candidates)
critic=2, actor=4 -> SS reward = -4.9324 (52 candidates)
critic=2, actor=8 -> SS reward = -17.1339 (57 candidates)
>> winner: critic=1, actor=4, SS reward = -4.3932
```

```
[DDPG] env=NSSM obs=x1 policy=linear
critic=1, actor=- -> SS reward = -inf (0 candidates)
critic=2, actor=- -> SS reward = -inf (0 candidates)
>> winner: critic=1, actor=1, SS reward = -inf
```

```
[DDPG] env=NSSM obs=x1 policy=nonlinear
critic=1, actor=1 -> SS reward = -638.4011 (1 candidates)
critic=1, actor=4 -> SS reward = -inf (0 candidates)
critic=1, actor=8 -> SS reward = -inf (0 candidates)
critic=2, actor=1 -> SS reward = -inf (0 candidates)
critic=2, actor=4 -> SS reward = -inf (0 candidates)
critic=2, actor=8 -> SS reward = -inf (0 candidates)
>> winner: critic=1, actor=1, SS reward = -638.4011
```

```
[DDPG] env=NSSM obs=x2 policy=linear
critic=1, actor=- -> SS reward = -inf (0 candidates)
critic=2, actor=- -> SS reward = -inf (0 candidates)
>> winner: critic=1, actor=1, SS reward = -inf
```

```
[DDPG] env=NSSM obs=x2 policy=nonlinear
critic=1, actor=1 -> SS reward = -inf (0 candidates)
critic=1, actor=4 -> SS reward = -inf (0 candidates)
critic=1, actor=8 -> SS reward = -inf (0 candidates)
critic=2, actor=1 -> SS reward = -inf (0 candidates)
critic=2, actor=4 -> SS reward = -inf (0 candidates)
critic=2, actor=8 -> SS reward = -inf (0 candidates)
>> winner: critic=1, actor=1, SS reward = -inf
```

7.3 SAC training

```
[17]: def train_sac_cell(env_name: str, obs_spec: str) -> TrainedCell:
    obs_lag = obs_spec_to_lag(obs_spec)
    env_factory = lambda: make_env(env_name, observation_lag=obs_lag, seed=SEED)
    init = initial_state_from_data()
    env = env_factory()
    model = SAC(
        "MlpPolicy", env,
```

```

        policy_kwargs=dict(net_arch=[64, 64]),
        learning_starts=500, batch_size=64,
        tau=0.005, gamma=GAMMA, ent_coef="auto",
        verbose=0, seed=SEED,
    )
    callback = HTAgentSelectionCallback()
    model.learn(total_timesteps=TOTAL_TIMESTEPS, callback=callback,
    ↪progress_bar=False)
    _, ss_reward = select_best_candidate(model, callback.candidates,
    ↪env_factory, init_state=init)
    ↪return TrainedCell(
        algo="SAC", env_name=env_name, obs_spec=obs_spec,
    ↪policy_variant="nonlinear",
        critic_nodes=64, actor_nodes=64, steady_state_reward=ss_reward,
        model=model, training_rewards=list(callback.episode_rewards),
    )

print("=" * 72)
print("SAC: training with H&T (2021) agent selection")
print("=" * 72)
for env_name in ENV_NAMES:
    for obs_spec in OBS_SPECS:
        key = ("SAC", env_name, obs_spec, "nonlinear")
        print(f"\n[SAC] env={env_name} obs={obs_spec}")
        cell = train_sac_cell(env_name, obs_spec)
        trained_cells[key] = cell
        print(f"    >> SS reward = {cell.steady_state_reward:.4f}")

```

```

=====
SAC: training with H&T (2021) agent selection
=====

```

```

[SAC] env=SVAR obs=x1
>> SS reward = -9.6613

```

```

[SAC] env=SVAR obs=x2
>> SS reward = -9.9016

```

```

[SAC] env=TVP-SVAR-SV obs=x1
>> SS reward = -7.0856

```

```

[SAC] env=TVP-SVAR-SV obs=x2
>> SS reward = -42.3785

```

```

[SAC] env=NARX obs=x1
>> SS reward = -4.7112

```

```
[SAC] env=NARX obs=x2
>> SS reward = -5.5375
```

```
[SAC] env=NSSM obs=x1
>> SS reward = -inf
```

```
[SAC] env=NSSM obs=x2
>> SS reward = -71.2167
```

7.4 Tabular Q-learning training

```
[18]: def train_q_cell(env_name: str) -> TrainedCell:
    env = make_env(env_name, observation_lag=False, seed=SEED)
    agent = QLearningAgent.build(seed=SEED)
    rewards: List[float] = []
    for ep in range(Q_EPISODES):
        obs, _ = env.reset()
        total = 0.0
        for _ in range(T_MAX):
            i_val = agent.act(obs, explore=True)
            obs2, r, term, trunc, info = env.step(rate_to_normalised(i_val))
            agent.update(obs, i_val, r, obs2, term or trunc)
            total += float(r)
            obs = obs2
            if term or trunc:
                break
        agent.epsilon = max(0.02, agent.epsilon * 0.999)
        rewards.append(total)
    return TrainedCell(
        algo="Q-learning", env_name=env_name, obs_spec="x1",
        policy_variant=None,
        critic_nodes=None, actor_nodes=None,
        steady_state_reward=float(np.mean(rewards[-200:])),
        model=agent, training_rewards=rewards,
    )

print("=" * 72)
print("Tabular Q-learning")
print("=" * 72)
for env_name in ENV_NAMES:
    key = ("Q-learning", env_name, "x1", None)
    print(f"\n[Q-learning] env={env_name}")
    cell = train_q_cell(env_name)
    trained_cells[key] = cell
```

```
print(f" >> mean reward (last 200 episodes) = {cell.steady_state_reward:.
↪4f}")
```

```
=====
Tabular Q-learning
=====
```

```
[Q-learning] env=SVAR
>> mean reward (last 200 episodes) = -493.5480

[Q-learning] env=TVP-SVAR-SV
>> mean reward (last 200 episodes) = -90.4343

[Q-learning] env=NARX
>> mean reward (last 200 episodes) = -321.4640

[Q-learning] env=NSSM
>> mean reward (last 200 episodes) = -19425.9458
```

7.5 Training reward curves

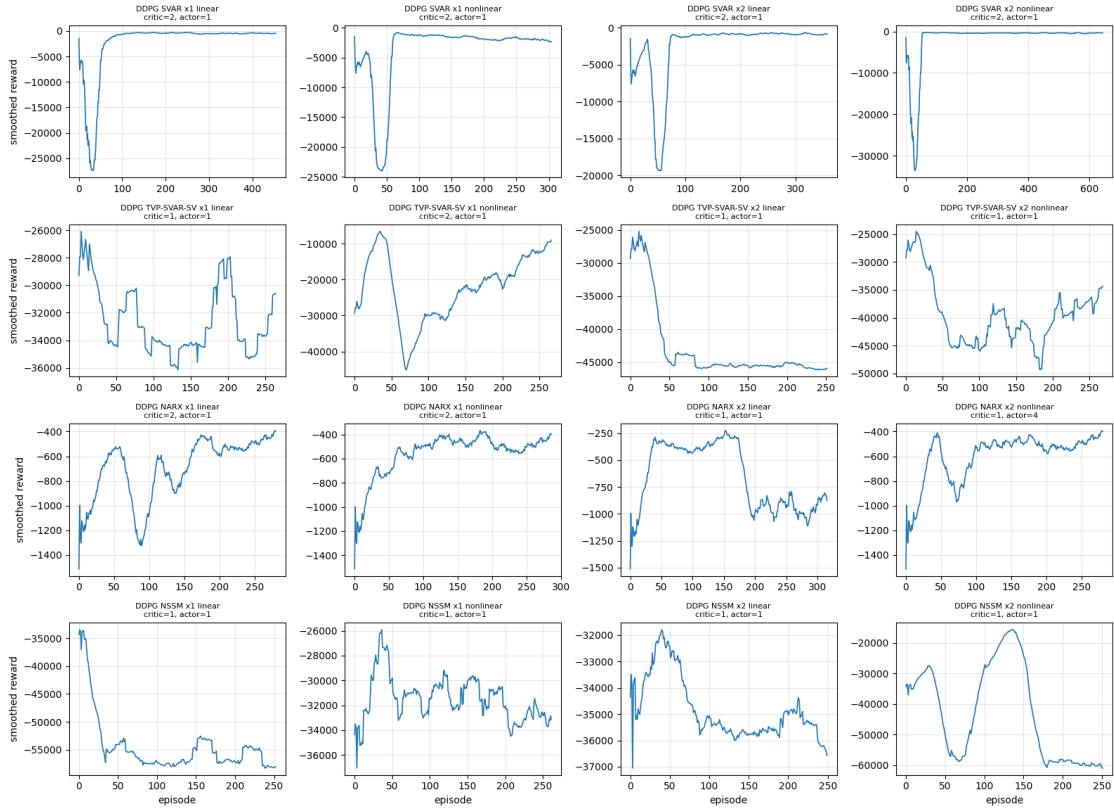
```
[19]: # DDPG training curves: 4 envs x 2 obs x 2 policies = 16 cells (4x4 grid plot)
fig, axes = plt.subplots(4, 4, figsize=(16, 12), sharex=False)
for row, env_name in enumerate(ENV_NAMES):
    col = 0
    for obs_spec in OBS_SPECS:
        for policy_variant in POLICY_VARIANTS:
            key = ("DDPG", env_name, obs_spec, policy_variant)
            ax = axes[row, col]
            if key in trained_cells:
                cell = trained_cells[key]
                curve = pd.Series(cell.training_rewards).rolling(25,
↪min_periods=1).mean()
                ax.plot(curve.values, lw=1.2)
                ax.set_title(
                    f"DDPG {env_name} {obs_spec} {policy_variant}\n"
                    f"critic={cell.critic_nodes}, actor={cell.actor_nodes}",
                    fontsize=8,
                )
            ax.grid(alpha=0.3)
            if row == 3: ax.set_xlabel("episode")
            if col == 0: ax.set_ylabel("smoothed reward")
            col += 1
fig.suptitle("DDPG training curves (winning architecture per cell)", y=1.0)
fig.tight_layout()
plt.show()
```

```

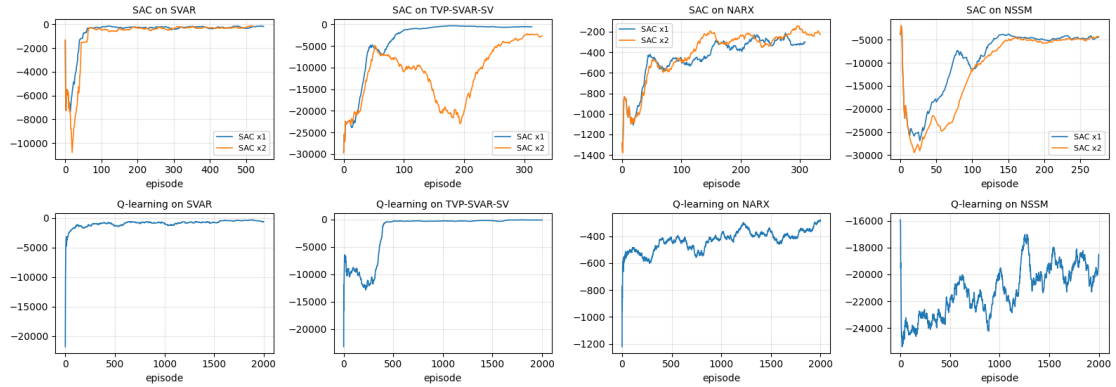
# SAC + Q-learning
fig, axes = plt.subplots(2, 4, figsize=(16, 6), sharex=False)
for j, env_name in enumerate(ENV_NAMES):
    sac_x1 = trained_cells.get(("SAC", env_name, "x1", "nonlinear"))
    sac_x2 = trained_cells.get(("SAC", env_name, "x2", "nonlinear"))
    q_cell = trained_cells.get(("Q-learning", env_name, "x1", None))
    ax = axes[0, j]
    if sac_x1 is not None:
        ax.plot(pd.Series(sac_x1.training_rewards).rolling(25, min_periods=1).
↪mean().values,
                lw=1.2, label="SAC x1")
    if sac_x2 is not None:
        ax.plot(pd.Series(sac_x2.training_rewards).rolling(25, min_periods=1).
↪mean().values,
                lw=1.2, label="SAC x2")
    ax.set_title(f"SAC on {env_name}", fontsize=10); ax.set_xlabel("episode");
↪ax.grid(alpha=0.3); ax.legend(fontsize=8)
    ax = axes[1, j]
    if q_cell is not None:
        ax.plot(pd.Series(q_cell.training_rewards).rolling(100, min_periods=1).
↪mean().values, lw=1.2)
    ax.set_title(f"Q-learning on {env_name}", fontsize=10); ax.
↪set_xlabel("episode"); ax.grid(alpha=0.3)
fig.suptitle("SAC and Q-learning training curves", y=1.0)
fig.tight_layout()
plt.show()

```

DDPG training curves (winning architecture per cell)



SAC and Q-learning training curves



8 Tables 2 and 4: chosen architectures and linearised policies

8.1 Table 2: chosen critic and actor hidden nodes per cell

```
[20]: table2_rows = []
for env_name in ENV_NAMES:
    for policy_variant in POLICY_VARIANTS:
        for obs_spec in OBS_SPECS:
            key = ("DDPG", env_name, obs_spec, policy_variant)
            cell = trained_cells.get(key)
            if cell is None:
                continue
            inputs_str = "(y_t, pi_t)" if obs_spec == "x1" else "(y_t, y_{t-1}, u_{t-1}, pi_t, pi_{t-1})"
            table2_rows.append({
                "Economy": env_name,
                "Policy structure": policy_variant.capitalize(),
                "Policy inputs": inputs_str,
                "Critic nodes": cell.critic_nodes,
                "Actor nodes": cell.actor_nodes,
                "SS reward": round(cell.steady_state_reward, 3),
            })

table2_df = pd.DataFrame(table2_rows)
print("Table 2 equivalent: chosen numbers of hidden nodes")
print(table2_df.to_string(index=False))
```

Table 2 equivalent: chosen numbers of hidden nodes

	Economy	Policy structure	Policy inputs	Critic nodes	Actor nodes
	SS reward				
	SVAR	Linear	(y_t, pi_t)	2	
1	-11.700				
	SVAR	Linear (y_t, y_{t-1}, pi_t, pi_{t-1})		2	
1	-8.058				
	SVAR	Nonlinear	(y_t, pi_t)	2	
1	-6.125				
	SVAR	Nonlinear (y_t, y_{t-1}, pi_t, pi_{t-1})		2	
1	-15.430				
	TVP-SVAR-SV	Linear	(y_t, pi_t)	1	
1	-inf				
	TVP-SVAR-SV	Linear (y_t, y_{t-1}, pi_t, pi_{t-1})		1	
1	-inf				
	TVP-SVAR-SV	Nonlinear	(y_t, pi_t)	2	
1	-0.519				
	TVP-SVAR-SV	Nonlinear (y_t, y_{t-1}, pi_t, pi_{t-1})		1	
1	-inf				
	NARX	Linear	(y_t, pi_t)	2	
1	-10.934				

	NARX	Linear (y_t, y_{t-1}, pi_t, pi_{t-1})	1
1	-5.530		
	NARX	Nonlinear (y_t, pi_t)	2
1	-6.211		
	NARX	Nonlinear (y_t, y_{t-1}, pi_t, pi_{t-1})	1
4	-4.393		
	NSSM	Linear (y_t, pi_t)	1
1	-inf		
	NSSM	Linear (y_t, y_{t-1}, pi_t, pi_{t-1})	1
1	-inf		
	NSSM	Nonlinear (y_t, pi_t)	1
1	-638.401		
	NSSM	Nonlinear (y_t, y_{t-1}, pi_t, pi_{t-1})	1
1	-inf		

8.2 Table 4: linearised policy coefficients

For linear DDPG cells we read $(\alpha_0, \beta_\pi, \beta_y)$ directly off `LinearActor.mu.weight` and translate from the SB3-normalised action space $[-1, 1]$ back to the H&T rate-response coefficients. For nonlinear policies we report an OLS-linearised approximation on a grid.

```
[21]: def extract_linear_coefficients(cell: TrainedCell) -> Dict[str, float]:
    actor = cell.model.policy.actor
    if not isinstance(actor, LinearActor):
        raise ValueError("Cell does not use a LinearActor.")
    W = actor.mu.weight.detach().cpu().numpy().reshape(-1)
    b = actor.mu.bias.detach().cpu().numpy().reshape(-1)[0]
    scale = 0.5 * ACTION_HIGH
    alpha0 = scale * (b + 1.0)
    coefs = scale * W
    if cell.obs_spec == "x1":
        result = {"alpha0": float(alpha0),
                  "beta_pi_0": float(coefs[1]),
                  "beta_y_0": float(coefs[0])}
        result["alpha0"] = result["alpha0"] + result["beta_pi_0"] * PI_STAR
    else:
        result = {"alpha0": float(alpha0),
                  "beta_pi_0": float(coefs[2]), "beta_pi_1": float(coefs[3]),
                  "beta_y_0": float(coefs[0]), "beta_y_1": float(coefs[1])}
        result["alpha0"] = result["alpha0"] + (result["beta_pi_0"] +
        ↪ result["beta_pi_1"]) * PI_STAR
    return result

def linearise_policy(cell: TrainedCell,
                    y_grid: np.ndarray = np.linspace(-4, 4, 17),
                    pi_grid: np.ndarray = np.linspace(0, 6, 13)) -> Dict[str, ↪
    ↪ float]:
```

```

X_rows, i_rows = [], []
obs_lag = obs_spec_to_lag(cell.obs_spec)
for y_val in y_grid:
    for pi_val in pi_grid:
        obs = (np.array([y_val, y_val, pi_val, pi_val], dtype=np.float32)
                if obs_lag else np.array([y_val, pi_val], dtype=np.float32))
        X_rows.append([1.0, pi_val - PI_STAR, y_val])
        if cell.algo == "Q-learning":
            i_val = float(cell.model.act(obs, explore=False))
        else:
            action, _ = cell.model.predict(obs, deterministic=True)
            i_val = float(normalised_to_rate(action))
        i_rows.append(i_val)
X_mat = np.asarray(X_rows)
i_vec = np.asarray(i_rows)
coef, *_ = np.linalg.lstsq(X_mat, i_vec, rcond=None)
pred = X_mat @ coef
r2 = 1.0 - np.var(i_vec - pred) / max(np.var(i_vec), 1e-12)
return {"alpha0": float(coef[0]), "beta_pi_0": float(coef[1]),
        "beta_y_0": float(coef[2]), "R2": float(r2)}

table4_rows = [
    {"Policy": "TR93", "alpha0": 1.0, "beta_pi_0": 1.5, "beta_y_0": 0.5, "R2": 1.0},
    {"Policy": "BA", "alpha0": 1.0, "beta_pi_0": 1.5, "beta_y_0": 1.0, "R2": 1.0},
    {"Policy": "NPP", "alpha0": 0.0, "beta_pi_0": 2.0, "beta_y_0": 0.5, "R2": 1.0},
]
for env_name in ENV_NAMES:
    for obs_spec in OBS_SPECS:
        for policy_variant in POLICY_VARIANTS:
            key = ("DDPG", env_name, obs_spec, policy_variant)
            cell = trained_cells.get(key)
            if cell is None:
                continue
            # Use linearise_policy (OLS on a (pi, y) grid through the
            # SB3-clipped action) for
            # BOTH linear and nonlinear actors. The previous
            # `extract_linear_coefficients` path
            # read raw LinearActor weights, which inflate past the action-space
            # training and report unboundedly large "Taylor coefficients"
            # (alpha0 = -130, etc.)
            # that are not the policy the env actually sees.

```

```

        coefs = linearise_policy(cell)
        coefs.setdefault("R2", 1.0)
        row = {"Policy": f"RL_{env_name}_{obs_spec}_{policy_variant}",
        ↪**coefs}
        table4_rows.append(row)

table4_df = pd.DataFrame(table4_rows)
print("Table 4 equivalent: policy coefficients")
print(table4_df.round(3).to_string(index=False))

```

Table 4 equivalent: policy coefficients

	Policy	alpha0	beta_pi_0	beta_y_0	R2
	TR93	1.000	1.500	0.500	1.000
	BA	1.000	1.500	1.000	1.000
	NPP	0.000	2.000	0.500	1.000
	RL_SVAR_x1_linear	7.923	1.184	2.257	0.781
	RL_SVAR_x1_nonlinear	4.192	2.312	1.212	0.843
	RL_SVAR_x2_linear	7.628	0.318	2.491	0.909
	RL_SVAR_x2_nonlinear	7.049	1.084	2.431	0.760
	RL_TVP-SVAR-SV_x1_linear	2.387	-0.918	-0.761	0.330
	RL_TVP-SVAR-SV_x1_nonlinear	3.030	0.953	0.232	0.871
	RL_TVP-SVAR-SV_x2_linear	11.279	1.279	1.154	0.447
	RL_TVP-SVAR-SV_x2_nonlinear	2.671	-0.794	-1.053	0.379
	RL_NARX_x1_linear	8.395	-0.202	2.547	0.874
	RL_NARX_x1_nonlinear	2.911	0.993	-0.272	0.876
	RL_NARX_x2_linear	6.327	-0.148	2.445	0.886
	RL_NARX_x2_nonlinear	6.521	-0.127	0.804	0.903
	RL_NSSM_x1_linear	4.261	-1.242	-1.384	0.623
	RL_NSSM_x1_nonlinear	3.745	1.085	-0.006	0.838
	RL_NSSM_x2_linear	5.081	-1.500	1.551	0.560
	RL_NSSM_x2_nonlinear	9.187	-2.337	1.728	0.686

9 Figures 8-11: learned reaction-function surfaces

One figure per economy. Each figure shows a 2x2 grid of contour plots over (π_t, y_t) : rows are observation specifications (x_1 / x_2), columns are policy variants (linear / nonlinear). The white contour marks $i_t = 3\%$ (approximate neutral rate); dashed lines mark $\pi_t = \pi^*$ and $y_t = 0$. Cells without a trained agent are left blank.

```

[22]: def policy_surface_for_cell(cell: TrainedCell,
        y_grid: np.ndarray = np.linspace(-5, 5, 41),
        pi_grid: np.ndarray = np.linspace(0, 5, 41)) ->
        ↪Tuple[np.ndarray, np.ndarray, np.ndarray]:
        """Evaluate the learned reaction function on a (pi, y) grid for plotting."""
        PI, Y = np.meshgrid(pi_grid, y_grid)
        FFR = np.zeros_like(PI)

```

```

for ii in range(Y.shape[0]):
    for jj in range(PI.shape[1]):
        y_val, pi_val = Y[ii, jj], PI[ii, jj]
        obs = (np.array([y_val, y_val, pi_val, pi_val], dtype=np.float32)
               if cell.obs_spec == "x2" else np.array([y_val, pi_val],
               dtype=np.float32))
        if cell.algo == "Q-learning":
            FFR[ii, jj] = float(cell.model.act(obs, explore=False))
        else:
            action, _ = cell.model.predict(obs, deterministic=True)
            FFR[ii, jj] = float(normalised_to_rate(action))
    return PI, Y, FFR

def plot_policy_surfaces(env_name: str, fig_label: str) -> None:
    """Hinterlang-Tanzer style contour view of the learned reaction function.

    One row per observation spec (x1 / x2), two columns (linear / nonlinear_
    actor).
    """
    fig, axes = plt.subplots(2, 2, figsize=(11, 8), sharex=True, sharey=True)
    levels = np.linspace(0, ACTION_HIGH, 16)
    cs = None
    for row, obs_spec in enumerate(OBS_SPECS):
        for col, variant in enumerate(POLICY_VARIANTS):
            ax = axes[row, col]
            cell = trained_cells.get(("DDPG", env_name, obs_spec, variant))
            if cell is None:
                ax.text(0.5, 0.5, "(no trained cell)", ha="center", va="center",
                       transform=ax.transAxes, fontsize=10, color="grey")
                ax.set_title(f"{obs_spec}, {variant}", fontsize=10)
                continue
            PI, Y, FFR = policy_surface_for_cell(cell)
            cs = ax.contourf(PI, Y, FFR, levels=levels, cmap="viridis",
            extend="both")
            ax.contour(PI, Y, FFR, levels=[3.0], colors="white", linewidths=0.
            8, alpha=0.7)
            ax.axvline(PI_STAR, color="black", lw=0.6, ls="--", alpha=0.4)
            ax.axhline(0.0, color="black", lw=0.6, ls="--", alpha=0.4)
            ax.set_title(f"{obs_spec}, {variant} (critic={cell.critic_nodes},
            actor={cell.actor_nodes})",
                       fontsize=9)
            axes[row, 0].set_ylabel("output gap, y_t")
    for col in range(2):
        axes[1, col].set_xlabel("inflation, pi_t")
    if cs is not None:
        fig.colorbar(cs, ax=axes.ravel().tolist(), shrink=0.85, label="i_t (%)")

```

```

fig.suptitle(f"{fig_label}: DDPG policy surfaces ({env_name})", y=0.995,
fontsize=12)
fig.tight_layout(rect=[0, 0, 0.92, 0.97])
plt.show()

for env_name, fig_label in zip(ENV_NAMES, ["Figure 8", "Figure 9", "Figure 10",
"Figure 11"]):
    plot_policy_surfaces(env_name, fig_label)

```

Figure 8: DDPG policy surfaces (SVAR)

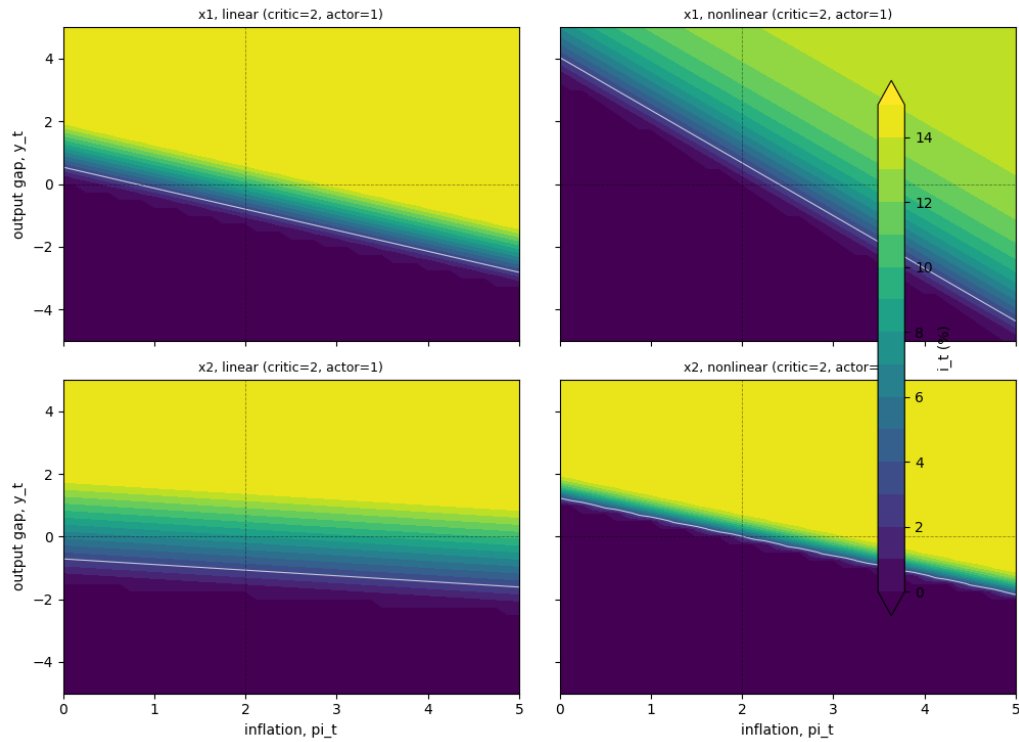


Figure 9: DDPG policy surfaces (TVP-SVAR-SV)

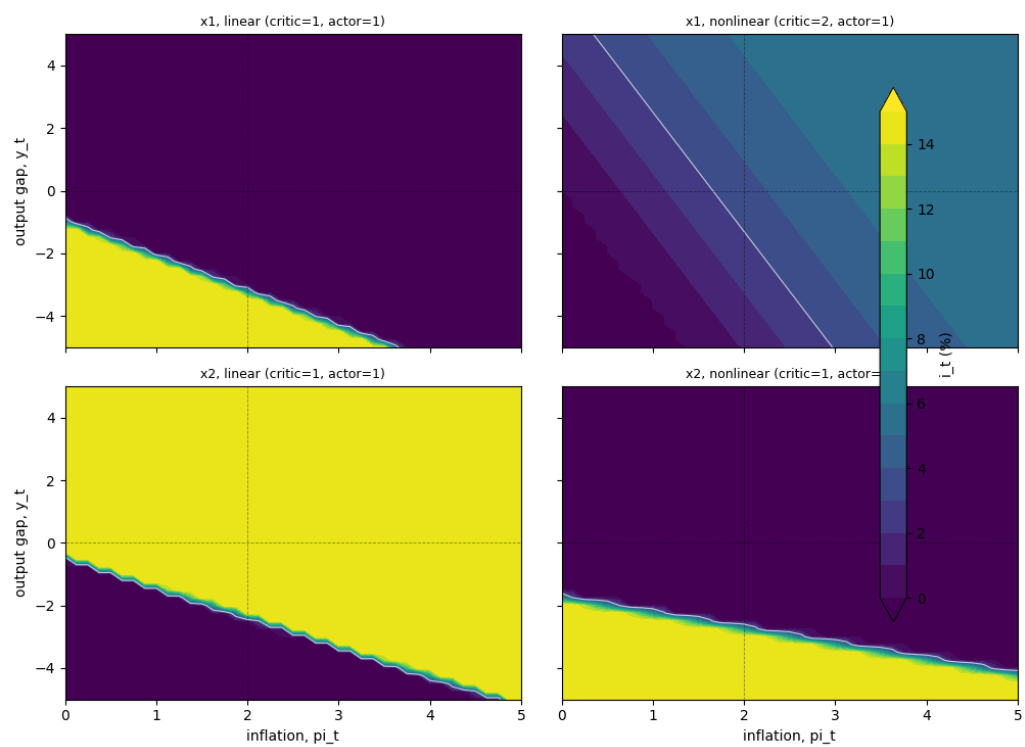


Figure 10: DDPG policy surfaces (NARX)

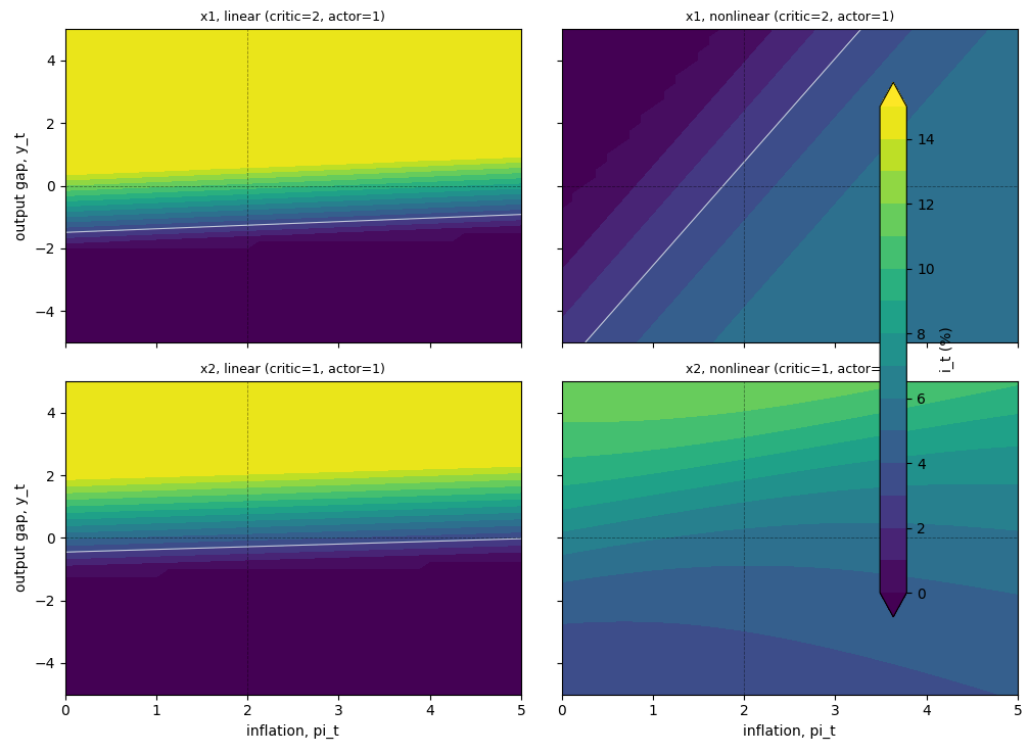
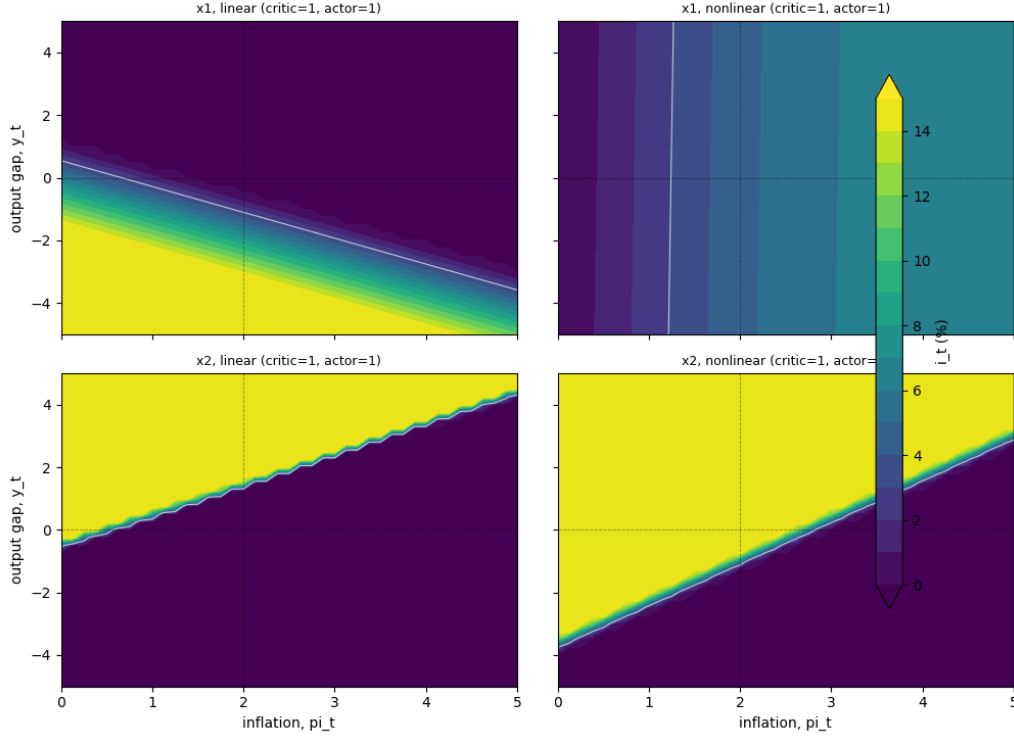


Figure 11: DDPG policy surfaces (NSSM)



10 Counterfactual analysis

10.1 Tables 5-6: counterfactual loss decomposition

```
[23]: def tr93(obs: np.ndarray) -> float:
    y = float(obs[0]); pi = float(obs[1]) if len(obs) == 2 else float(obs[2])
    return max(0.0, 1.0 + 1.5 * (pi - PI_STAR) + 0.5 * y + PI_STAR)

def balanced(obs: np.ndarray) -> float:
    y = float(obs[0]); pi = float(obs[1]) if len(obs) == 2 else float(obs[2])
    return max(0.0, 1.0 + 1.5 * (pi - PI_STAR) + 1.0 * y + PI_STAR)

def npp(obs: np.ndarray) -> float:
    y = float(obs[0]); pi = float(obs[1]) if len(obs) == 2 else float(obs[2])
    return max(0.0, 0.0 + 2.0 * (pi - PI_STAR) + 0.5 * y + PI_STAR)

REFERENCE_RULES = {"TR93": tr93, "BA": balanced, "NPP": npp}
```

```

def run_counterfactual(env: MonetaryEnv, policy_fn: Callable[[np.ndarray], float],
                      T: int = 80, init_state: Optional[Dict[str, float]] = None,
                      seed: int = SEED) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
    obs, _ = env.reset(options={"init_state": init_state} if init_state else None)
    env.rng = np.random.default_rng(seed)
    pi_path, y_path, i_path = [], [], []
    for _ in range(T):
        i_val = policy_fn(obs)
        obs2, _, _, _, info = env.step(rate_to_normalised(i_val))
        pi_path.append(info["pi"]); y_path.append(info["y"]); i_path.append(info["i"])
        obs = obs2
    return np.asarray(pi_path), np.asarray(y_path), np.asarray(i_path)

def policy_callable(cell: TrainedCell) -> Callable[[np.ndarray], float]:
    if cell.algo == "Q-learning":
        return lambda obs: cell.model.act(obs, explore=False)
    return lambda obs: float(normalised_to_rate(cell.model.predict(obs).astype(np.float32),
                                                deterministic=True)[0]))

def loss_decomposition(pi_path, y_path, i_path=None) -> Dict[str, float]:
    """H&T eq.16 quadratic loss plus Sack-Wieland Var(Delta i) for diagnostics.

    `Loss_H&T` = 0.5 dev2_pi + 0.5 dev2_y matches H&T eq. 16.
    `Loss_full` adds 0.5 dev2_di to mirror the trained reward.
    """
    dev2_pi = float(np.mean((pi_path - PI_STAR) ** 2))
    dev2_y = float(np.mean(y_path ** 2))
    out = {"dev2_pi": dev2_pi, "dev2_y": dev2_y}
    if i_path is not None and len(i_path) > 1:
        di = np.diff(np.asarray(i_path))
        out["dev2_di"] = float(np.mean(di ** 2))
        out["Loss"] = 0.5 * dev2_pi + 0.5 * dev2_y + 0.5 * out["dev2_di"]
    else:
        out["Loss"] = 0.5 * dev2_pi + 0.5 * dev2_y
    return out

```

```

T_CF = 80
init = initial_state_from_data()
counterfactual_paths: Dict[Tuple[str, str], Dict[str, np.ndarray]] = {}

for env_name in ENV_NAMES:
    for obs_spec in OBS_SPECS:
        obs_lag = obs_spec_to_lag(obs_spec)
        for label, fn in REFERENCE_RULES.items():
            env = make_env(env_name, observation_lag=obs_lag, seed=SEED)
            pi_p, y_p, i_p = run_counterfactual(env, fn, T=T_CF,
↪init_state=init)
            counterfactual_paths[(env_name, f"{label}_{obs_spec}")] = {"pi":
↪pi_p, "y": y_p, "i": i_p}

for (algo, env_name, obs_spec, policy_variant), cell in trained_cells.items():
    obs_lag = obs_spec_to_lag(obs_spec)
    env = make_env(env_name, observation_lag=obs_lag, seed=SEED)
    pi_p, y_p, i_p = run_counterfactual(env, policy_callable(cell), T=T_CF,
↪init_state=init)
    pv_str = policy_variant if policy_variant else "none"
    counterfactual_paths[(env_name, f"RL_{algo}_{obs_spec}_{pv_str}")] = {"pi":
↪pi_p, "y": y_p, "i": i_p}

actual_pi = df["pi"].to_numpy()
actual_y = df["y"].to_numpy()
actual_i_arr = df["i"].to_numpy()
actual_loss = loss_decomposition(actual_pi, actual_y, actual_i_arr)

for env_name in ENV_NAMES:
    print(f"\nCounterfactual loss decomposition under the {env_name} economy")
    rows = [{"Policy": "Actual FFR", **{k: round(v, 3) for k, v in actual_loss.
↪items()}}]
    for (env_k, policy_label), paths in counterfactual_paths.items():
        if env_k != env_name: continue
        rows.append({"Policy": policy_label,
↪**{k: round(v, 3) for k, v in
↪loss_decomposition(paths["pi"], paths["y"], paths["i"]).items()}}})
    print(pd.DataFrame(rows).to_string(index=False))

```

Counterfactual loss decomposition under the SVAR economy

	Policy	dev2_pi	dev2_y	dev2_di	Loss
	Actual FFR	0.849	1.621	0.222	1.346
	TR93_x1	1.523	0.787	0.176	1.243
	BA_x1	1.658	0.712	0.353	1.362
	NPP_x1	1.826	0.794	0.274	1.447

TR93_x2	1.523	0.787	0.176	1.243
BA_x2	1.658	0.712	0.353	1.362
NPP_x2	1.826	0.794	0.274	1.447
RL_DDPG_x1_linear	0.682	1.008	13.215	7.453
RL_DDPG_x1_nonlinear	1.002	0.842	4.959	3.401
RL_DDPG_x2_linear	1.142	1.054	3.334	2.764
RL_DDPG_x2_nonlinear	1.025	0.886	22.717	12.314
RL_SAC_x1_nonlinear	0.990	0.847	9.243	5.540
RL_SAC_x2_nonlinear	1.236	0.818	7.752	4.903
RL_Q-learning_x1_none	1.189	1.530	4.462	3.590

Counterfactual loss decomposition under the TVP-SVAR-SV economy

Policy	dev2_pi	dev2_y	dev2_di	Loss
Actual FFR	0.849	1.621	0.222	1.346
TR93_x1	0.729	3.173	0.112	2.007
BA_x1	1.571	4.571	0.127	3.134
NPP_x1	1.597	5.351	0.183	3.565
TR93_x2	0.729	3.173	0.112	2.007
BA_x2	1.571	4.571	0.127	3.134
NPP_x2	1.597	5.351	0.183	3.565
RL_DDPG_x1_linear	55.655	8.228	132.965	98.424
RL_DDPG_x1_nonlinear	0.192	0.242	0.080	0.257
RL_DDPG_x2_linear	35.411	53.609	0.000	44.510
RL_DDPG_x2_nonlinear	60.482	7.347	97.667	82.749
RL_SAC_x1_nonlinear	0.849	1.464	1.788	2.050
RL_SAC_x2_nonlinear	3.425	2.548	73.958	39.966
RL_Q-learning_x1_none	0.534	0.185	1.671	1.195

Counterfactual loss decomposition under the NARX economy

Policy	dev2_pi	dev2_y	dev2_di	Loss
Actual FFR	0.849	1.621	0.222	1.346
TR93_x1	0.898	1.415	0.206	1.259
BA_x1	1.021	1.465	0.381	1.434
NPP_x1	1.010	1.338	0.311	1.329
TR93_x2	0.898	1.415	0.206	1.259
BA_x2	1.021	1.465	0.381	1.434
NPP_x2	1.010	1.338	0.311	1.329
RL_DDPG_x1_linear	0.876	1.674	6.912	4.731
RL_DDPG_x1_nonlinear	0.841	1.373	0.192	1.203
RL_DDPG_x2_linear	1.240	1.109	2.185	2.267
RL_DDPG_x2_nonlinear	0.522	1.135	0.291	0.974
RL_SAC_x1_nonlinear	0.752	1.341	0.021	1.057
RL_SAC_x2_nonlinear	1.071	0.713	1.387	1.585
RL_Q-learning_x1_none	0.388	1.133	11.038	6.279

Counterfactual loss decomposition under the NSSM economy

Policy	dev2_pi	dev2_y	dev2_di	Loss
Actual FFR	0.849	1.621	0.222	1.346

TR93_x1	45.089	9.270	1.022	27.690
BA_x1	45.053	8.644	0.384	27.040
NPP_x1	44.609	9.485	1.889	27.991
TR93_x2	45.089	9.270	1.022	27.690
BA_x2	45.053	8.644	0.384	27.040
NPP_x2	44.609	9.485	1.889	27.991
RL_DDPG_x1_linear	53.589	50.194	86.701	95.242
RL_DDPG_x1_nonlinear	42.466	13.681	8.769	32.458
RL_DDPG_x2_linear	47.326	22.998	2.848	36.586
RL_DDPG_x2_nonlinear	51.738	49.093	225.000	162.916
RL_SAC_x1_nonlinear	5.891	3.411	0.908	5.105
RL_SAC_x2_nonlinear	5.803	2.510	5.935	7.124
RL_Q-learning_x1_none	15.423	3.325	11.687	15.217

10.2 Figures 12-15: actual and counterfactual time series

One figure per economy. y-axes are clipped to readable bands; if a closed loop diverges (any $|\pi|$ or $|y| > 1000$ in the simulated path), the subtitle annotates which policies diverged so the reader can identify dropouts without misreading the auto-scaled axes.

```
[24]: def plot_counterfactual_panels(env_name: str, policies_to_plot: Sequence[str],
                                     title: str,
                                     pi_ylim: Optional[Tuple[float, float]] = (-2.0,
                                     ↪8.0),
                                     y_ylim: Optional[Tuple[float, float]] = (-6.0,
                                     ↪6.0)) -> None:
    """Counterfactual time-series panels with defensive y-axis clipping.

    If a closed loop diverges, matplotlib's auto-scale would crush the
    ↪historical
    reference line to invisibility. The default clips are wide enough for any
    stable policy. Pass `pi_ylim=None` / `y_ylim=None` to restore auto-scaling.
    """

    actual_i = df["i"].to_numpy()
    actual_pi_ = df["pi"].to_numpy()
    actual_y_ = df["y"].to_numpy()
    T_show = min(len(actual_i), T_CF)
    idx = pd.PeriodIndex(df.index[:T_show], freq="Q").to_timestamp()

    fig, axes = plt.subplots(3, 1, figsize=(12, 9), sharex=True)
    axes[0].plot(idx, actual_i[:T_show], color="black", lw=2.0, label="Actual")
    axes[1].plot(idx, actual_pi_[:T_show], color="black", lw=2.0,
    ↪label="Actual")
    axes[2].plot(idx, actual_y_[:T_show], color="black", lw=2.0, label="Actual")

    cmap = plt.get_cmap("tab10")
    diverged: List[str] = []
```

```

for k, label in enumerate(policies_to_plot):
    paths = counterfactual_paths.get((env_name, label))
    if paths is None: continue
    color = cmap(k)
    if (np.max(np.abs(paths["pi"])) > 1e3) or (np.max(np.abs(paths["y"])) >
↪1e3):
        diverged.append(label)
    axes[0].plot(idx, paths["i"][:T_show], color=color, lw=1.3, label=label)
    axes[1].plot(idx, paths["pi"][:T_show], color=color, lw=1.3,
↪label=label)
    axes[2].plot(idx, paths["y"][:T_show], color=color, lw=1.3, label=label)

    axes[0].set_ylabel("FFR")
    axes[0].set_ylim(-1.0, ACTION_HIGH + 1.0)
    axes[1].set_ylabel("Inflation"); axes[1].axhline(PI_STAR, color="grey",
↪lw=0.8, ls="--", alpha=0.6)
    axes[2].set_ylabel("Output gap"); axes[2].axhline(0.0, color="grey", lw=0.
↪8, ls="--", alpha=0.6)
    if pi_ylim is not None: axes[1].set_ylim(*pi_ylim)
    if y_ylim is not None: axes[2].set_ylim(*y_ylim)
    axes[2].set_xlabel("date")
    axes[0].legend(ncol=3, fontsize=8, loc="upper right")
    full_title = title if not diverged else f"{title}\n(closed loop divergent,
↪under: {'', ' '.join(diverged)})"
    fig.suptitle(full_title); fig.tight_layout(); plt.show()

# Figures 12-15: one per economy, monotonically numbered. NSSM call now includes
# the canonical reference rules (TR93/BA/NPP) so the RL policies have a baseline
# to compare against, matching the SVAR/TVP-SVAR-SV/NARX panels.
plot_counterfactual_panels(
    "SVAR",
    ["TR93_x1", "BA_x1", "NPP_x1", "RL_DDPG_x1_linear", "RL_DDPG_x2_linear"],
    "Figure 12: SVAR economy",
)
plot_counterfactual_panels(
    "TVP-SVAR-SV",
    ["TR93_x1", "BA_x1", "NPP_x1", "RL_DDPG_x1_linear", "RL_SAC_x1_nonlinear"],
    "Figure 13: TVP-SVAR-SV economy",
)
plot_counterfactual_panels(
    "NARX",
    ["TR93_x1", "BA_x1", "NPP_x1", "RL_DDPG_x1_linear", "RL_DDPG_x2_linear"],
    "Figure 14: NARX economy",
)
plot_counterfactual_panels(
    "NSSM",

```

```

["TR93_x1", "BA_x1", "NPP_x1",
 "RL_DDPG_x1_linear", "RL_DDPG_x1_nonlinear", "RL_SAC_x1_nonlinear"],
"Figure 15: NSSM economy",
)

```

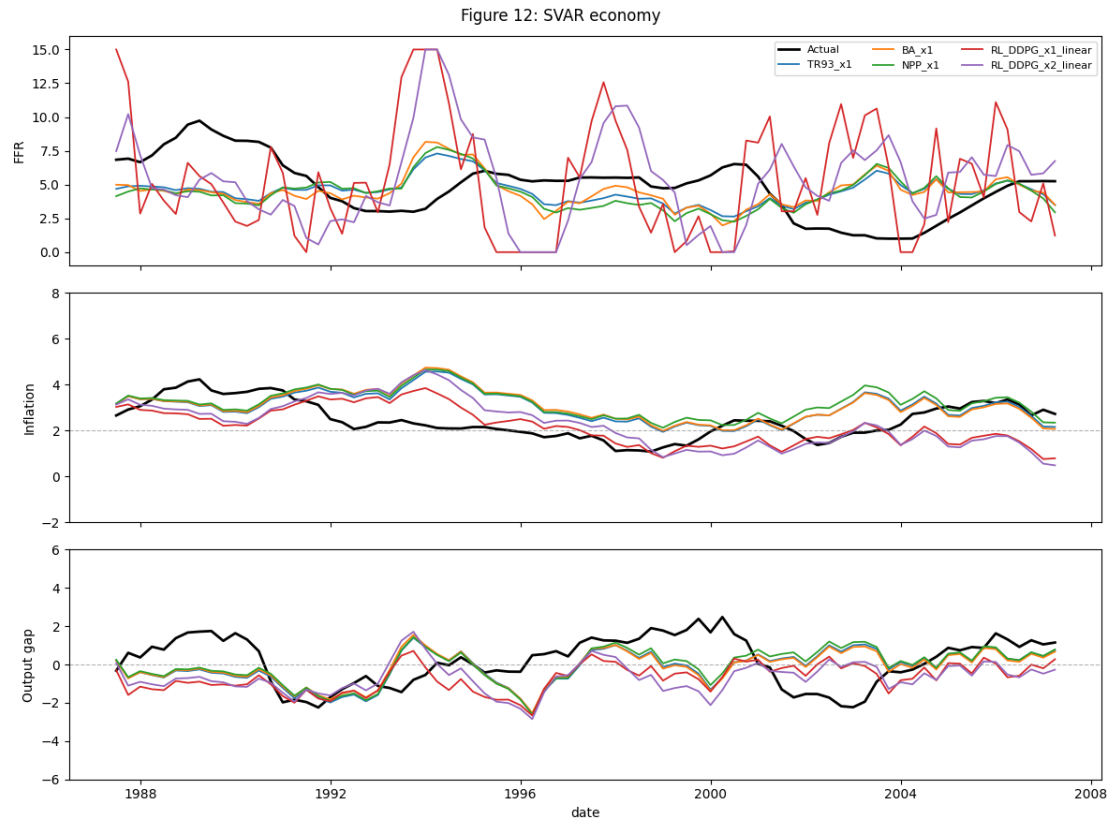


Figure 13: TVP-SVAR-SV economy

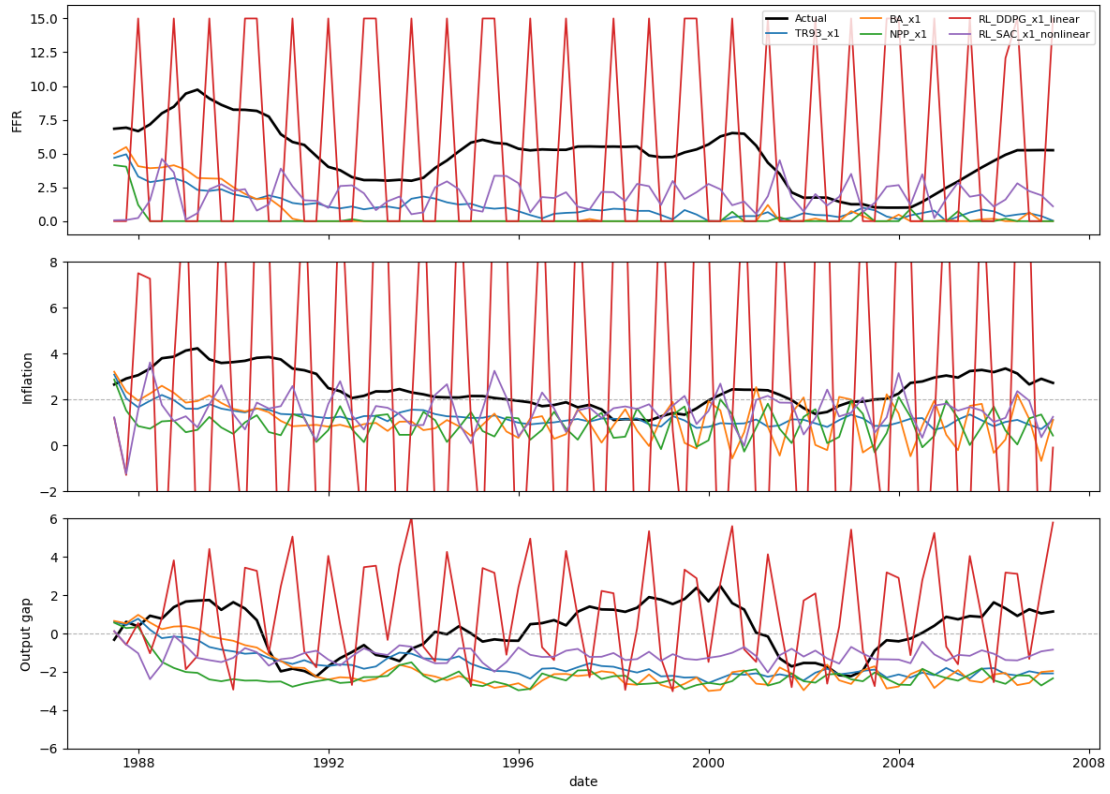
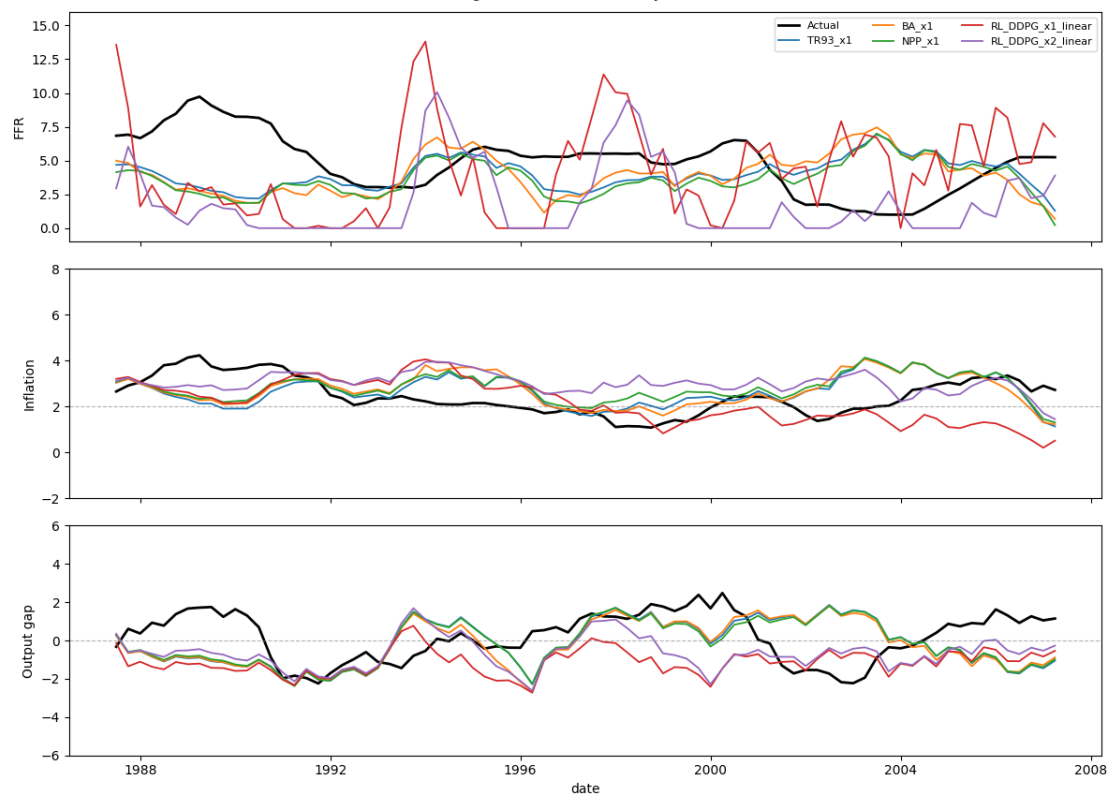
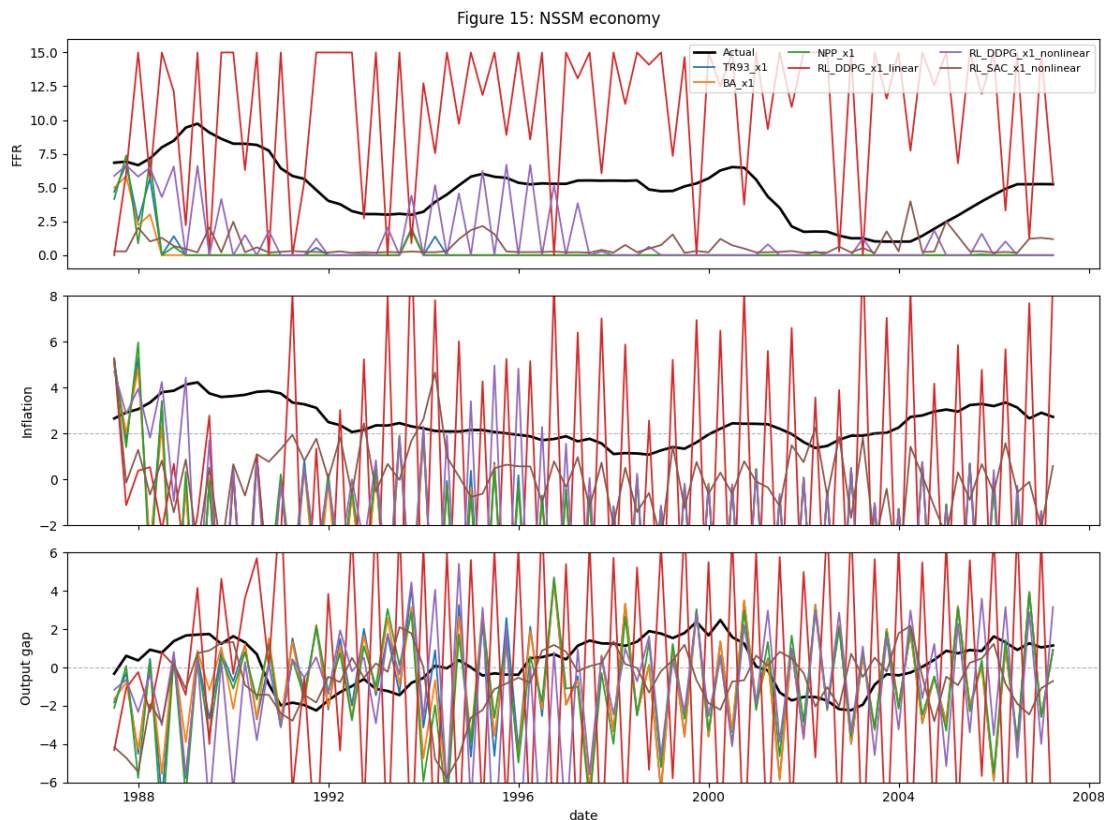


Figure 14: NARX economy





11 DSGE robustness exercise (H&T Section 3.4 / Table 7)

11.1 Methodology

Hinterlang & Tanzer (2021, Section 3.4 and Table 7) evaluate their optimised reaction functions on 11 DSGE models from the Macroeconomic Model Database (Wieland et al., 2012, 2016) and report the unconditional variances of inflation, the output gap, and the change in the policy rate *relative to* the Taylor (1993) rule under each model. The exercise is a robustness check: a policy rule that is optimised for one estimated economy (the SVAR or the ANN) is not guaranteed to perform well in other plausible structural representations, and the DSGE comparison establishes that the RL-optimised rules remain reasonably robust across models.

A full replication requires solving forward-looking rational-expectations DSGE models with Sims's (2002) `gensys` or an equivalent (the Macroeconomic Model Database is built on `dynare`). Our notebook does not embed `dynare`, so a strict replication of all 11 H&T models is out of scope. We instead implement the *backward-looking* subset of the H&T robustness exercise, which Sims's `gensys` reduces to a closed-form solve. The canonical backward-looking benchmark in the monetary-policy literature is Rudebusch & Svensson (1999, henceforth RS99), which H&T cite explicitly. We compute the unconditional variance of each variable under the closed-loop dynamics induced by each candidate policy rule and report the variances relative to the Taylor (1993) baseline.

11.2 Rudebusch-Svensson (1999) closed-form solver

The RS99 model is

$$\begin{aligned} y_t &= 1.1597 y_{t-1} - 0.2588 y_{t-2} - 0.0908 (i_{t-1} - \bar{\pi}_{t-1}) + \varepsilon_t^y, \\ \pi_t &= 0.701 \pi_{t-1} + 0.3 \pi_{t-2} - 0.1 \pi_{t-3} + 0.1 \pi_{t-4} + 0.1397 y_{t-1} + \varepsilon_t^\pi, \\ \bar{\pi}_{t-1} &= \frac{1}{4}(\pi_{t-1} + \pi_{t-2} + \pi_{t-3} + \pi_{t-4}), \end{aligned}$$

with shock standard deviations $\sigma_y = 0.819$ and $\sigma_\pi = 1.013$. We adopt the H&T policy rule format with one lag,

$$i_t = \alpha_0 + \beta_\pi^0(\pi_t - \pi^*) + \beta_\pi^1(\pi_{t-1} - \pi^*) + \beta_y^0 y_t + \beta_y^1 y_{t-1},$$

stack the system in the state $z_t = (y_t, y_{t-1}, \pi_t, \pi_{t-1}, \pi_{t-2}, \pi_{t-3})$, substitute the policy rule, and solve for the closed-loop $F(K)$ such that $z_{t+1} = F(K)z_t + \eta_{t+1}$. The unconditional state-covariance matrix is then the unique stable solution of the discrete-time Lyapunov equation $V = FVF^\top + \Sigma$, which we compute via `scipy.linalg.solve_discrete_lyapunov`. The unconditional variance of $\Delta i_t = K(z_t - z_{t-1})$ follows from $\text{Var}(\Delta i_t) = K(2V - FV - VF^\top)K^\top$ using the steady-state covariance $\text{Cov}(z_t, z_{t-1}) = FV$ implied by the closed-loop VAR.

```
[25]: def build_rs99_F(coeffs: Dict[str, float]) -> np.ndarray:
    """Closed-loop F matrix for RS99 under the H&T (eq. 17) policy rule.

    State z = [y_t, y_{t-1}, pi_t, pi_{t-1}, pi_{t-2}, pi_{t-3}] (6-dim).
    """
    c_pi0 = coeffs.get("beta_pi_0", 0.0)
    c_pi1 = coeffs.get("beta_pi_1", 0.0)
    c_y0 = coeffs.get("beta_y_0", 0.0)
    c_y1 = coeffs.get("beta_y_1", 0.0)

    # y_{t+1} coefficients on z
    # y_{t+1} = 1.1597 y_t - 0.2588 y_{t-1} - 0.0908 (i_t - pi_bar_t) + eps_y
    # i_t = c_pi0 pi_t + c_pi1 pi_{t-1} + c_y0 y_t + c_y1 y_{t-1}
    # pi_bar_t = (pi_t + pi_{t-1} + pi_{t-2} + pi_{t-3}) / 4
    f00 = 1.1597 - 0.0908 * c_y0
    f01 = -0.2588 - 0.0908 * c_y1
    f02 = -0.0908 * c_pi0 + 0.0908 / 4.0
    f03 = -0.0908 * c_pi1 + 0.0908 / 4.0
    f04 = 0.0908 / 4.0
    f05 = 0.0908 / 4.0

    F = np.array([
        [f00, f01, f02, f03, f04, f05], # y_{t+1}
        [1.0, 0.0, 0.0, 0.0, 0.0, 0.0], # y_t (now lag-1)
        [0.1397, 0.0, 0.701, 0.3, -0.1, 0.1], # pi_{t+1}
        [0.0, 0.0, 1.0, 0.0, 0.0, 0.0], # pi_t (now lag-1)
        [0.0, 0.0, 0.0, 1.0, 0.0, 0.0], # pi_{t-1} -> pi_{t-2}
```

```

        [0.0, 0.0, 0.0, 0.0, 1.0, 0.0], #  $\pi_{t-2} \rightarrow \pi_{t-3}$ 
    ])
    return F

RS99_SIGMA_Y = 0.819
RS99_SIGMA_PI = 1.013
RS99_SHOCK_COV = np.diag([RS99_SIGMA_Y ** 2, 0.0, RS99_SIGMA_PI ** 2, 0.0, 0.0,
↪0.0])

def rs99_variances(coeffs: Dict[str, float]) -> Optional[Dict[str, float]]:
    """Compute (var( $\pi$ ), var( $y$ ), var( $\Delta i$ )) under the given policy rule.

    Returns None if the closed loop is unstable (any  $|eig(F)| \geq 1$ ).
    """
    F = build_rs99_F(coeffs)
    eig = np.linalg.eigvals(F)
    if np.max(np.abs(eig)) >= 1.0 - 1e-10:
        return None
    V = scipy.linalg.solve_discrete_lyapunov(F, RS99_SHOCK_COV)
    var_y = float(V[0, 0])
    var_pi = float(V[2, 2])
    # var( $\Delta i$ ) =  $K (2V - FV - VF^T) K^T$ 
    K = np.array([
        coeffs.get("beta_y_0", 0.0),
        coeffs.get("beta_y_1", 0.0),
        coeffs.get("beta_pi_0", 0.0),
        coeffs.get("beta_pi_1", 0.0),
        0.0, 0.0,
    ])
    cov_diff = 2.0 * V - F @ V - V @ F.T
    var_di = float(K @ cov_diff @ K)
    return {"var_pi": var_pi, "var_y": var_y, "var_di": max(var_di, 0.0)}

```

11.3 Policy coefficient collection

For each trained DDPG cell we use `extract_linear_coefficients` (exact, for the linear policy variant) or `linearise_policy` (OLS approximation, for the nonlinear variant). For SAC and tabular Q-learning we use `linearise_policy` throughout. Reference rules are entered by hand from Hinterlang & Tanzer (2021, Table 4).

```

[26]: def dsge_coeffs_for_cell(cell: TrainedCell) -> Dict[str, float]:
    """Return policy coefficients in H&T eq.-17 form, with one lag."""
    if cell.algo == "DDPG" and cell.policy_variant == "linear":
        coefs = extract_linear_coefficients(cell)
    else:

```

```

        coefs = linearise_policy(cell)
    return {
        "beta_pi_0": coefs.get("beta_pi_0", 0.0),
        "beta_pi_1": coefs.get("beta_pi_1", 0.0),
        "beta_y_0": coefs.get("beta_y_0", 0.0),
        "beta_y_1": coefs.get("beta_y_1", 0.0),
    }

policy_coeffs_dsge = {
    "TR93": {"beta_pi_0": 1.5, "beta_pi_1": 0.0, "beta_y_0": 0.5, "beta_y_1": 0.
    ↪0},
    "BA": {"beta_pi_0": 1.5, "beta_pi_1": 0.0, "beta_y_0": 1.0, "beta_y_1": 0.
    ↪0},
    "NPP": {"beta_pi_0": 2.0, "beta_pi_1": 0.0, "beta_y_0": 0.5, "beta_y_1": 0.
    ↪0},
}
for (algo, env_name, obs_spec, policy_variant), cell in trained_cells.items():
    label = f"RL_{algo}_{env_name}_{obs_spec}_{policy_variant} if policy_variant_
    ↪else 'none'"
    policy_coeffs_dsge[label] = dsge_coeffs_for_cell(cell)

print(f"Collected policy coefficients for {len(policy_coeffs_dsge)} policies")

```

Collected policy coefficients for 31 policies

11.4 Table 7 equivalent: relative variances under RS99

We report variances of inflation, the output gap, and the change in the policy rate *relative to* TR93. Values less than 1 indicate that the candidate rule reduces the corresponding variance below the Taylor (1993) baseline; values greater than 1 indicate higher variance; `inf` indicates that the closed-loop system is unstable (Taylor principle violated).

```

[27]: dsge_results = {}
for label, coeffs in policy_coeffs_dsge.items():
    res = rs99_variances(coeffs)
    if res is None:
        dsge_results[label] = {"var_pi": np.inf, "var_y": np.inf, "var_di": np.
    ↪inf, "stable": False}
    else:
        dsge_results[label] = {**res, "stable": True}

# Normalise relative to TR93
tr93_res = dsge_results["TR93"]
rows = []
for label, r in dsge_results.items():
    if not r["stable"]:

```

```

        rows.append({"Policy": label, "var(pi)/TR93": np.inf, "var(y)/TR93": np.
↪inf,
                    "var(Delta i)/TR93": np.inf, "Mean": np.inf})

    continue
    ratio_pi = r["var_pi"] / max(tr93_res["var_pi"], 1e-12)
    ratio_y = r["var_y"] / max(tr93_res["var_y"], 1e-12)
    ratio_di = r["var_di"] / max(tr93_res["var_di"], 1e-12)
    rows.append({
        "Policy": label,
        "var(pi)/TR93": ratio_pi,
        "var(y)/TR93": ratio_y,
        "var(Delta i)/TR93": ratio_di,
        "Mean": (ratio_pi + ratio_y + ratio_di) / 3.0,
    })

table7_df = pd.DataFrame(rows).round(3)
print("Table 7 equivalent (RS99 closed-loop variances, ratios to TR93)")
print("Lower values indicate better stabilisation. `inf` = unstable closed loop.
↪")
print(table7_df.to_string(index=False))

```

```

-----
NameError                                Traceback (most recent call last)
Cell In[27], line 3
      1 dsge_results = {}
      2 for label, coeffs in policy_coeffs_dsge.items():
----> 3     res = rs99_variances(coeffs)
      4     if res is None:
      5         dsge_results[label] = {"var_pi": np.inf, "var_y": np.inf,
↪"var_di": np.inf, "stable": False}
      6     else:

Cell In[25], line 47, in rs99_variances(coeffs)
     43 F = build_rs99_F(coeffs)
     44 eig = np.linalg.eigvals(F)
     45 if np.max(np.abs(eig)) >= 1.0 - 1e-10:
     46     return None
--> 47 V = scipy.linalg.solve_discrete_lyapunov(F, RS99_SHOCK_COV)
     48 var_y = float(V[0, 0])
     49 var_pi = float(V[2, 2])
     50 # var(Delta i) = K (2V - F V - V F^T) K^T

NameError: name 'scipy' is not defined

```

11.5 Visualisation: RL rules vs reference rules under RS99

A scatter plot of the three relative-variance ratios separates the dominated rules ($TR93 = 1$, dot at origin offset; rules that strictly increase any variance) from the Pareto-efficient ones (rules that reduce at least one ratio without increasing the others).

```
[ ]: finite_rows = [r for r in rows if np.isfinite(r["Mean"])]
labels = [r["Policy"] for r in finite_rows]
ratio_pi = np.array([r["var(pi)/TR93"] for r in finite_rows])
ratio_y = np.array([r["var(y)/TR93"] for r in finite_rows])
ratio_di = np.array([r["var(Delta i)/TR93"] for r in finite_rows])

is_ref = np.array([lab in REFERENCE_RULES or lab == "TR93" for lab in labels])
is_rl = ~is_ref

fig, ax = plt.subplots(figsize=(10, 7))
ax.scatter(ratio_pi[is_ref], ratio_y[is_ref], s=140, c="black", marker="s",
           label="Reference rules")
ax.scatter(ratio_pi[is_rl], ratio_y[is_rl], s=80, c="C0", alpha=0.7, label="RL
           rules")
for k in np.where(is_ref)[0]:
    ax.annotate(labels[k], (ratio_pi[k], ratio_y[k]), xytext=(5, 5),
                textcoords="offset points", fontsize=9)
ax.axhline(1.0, color="grey", ls="--", lw=0.8, alpha=0.5)
ax.axvline(1.0, color="grey", ls="--", lw=0.8, alpha=0.5)
ax.set_xlabel("var(inflation) / TR93")
ax.set_ylabel("var(output gap) / TR93")
ax.set_title("RS99 closed-loop variance ratios (under each candidate rule)")
ax.legend()
ax.grid(alpha=0.3)
plt.show()

print("Reference rules:")
for r in finite_rows:
    if r["Policy"] in REFERENCE_RULES or r["Policy"] == "TR93":
        print(f" {r['Policy']:<10} var(pi)/TR93={r['var(pi)/TR93']:.3f}
        var(y)/TR93={r['var(y)/TR93']:.3f} var(Delta i)/TR93={r['var(Delta i)/
        TR93']:.3f}")
print("\nBest RL rules by mean ratio:")
for r in sorted(finite_rows, key=lambda r: r["Mean"])[5:]:
    if r["Policy"] in REFERENCE_RULES or r["Policy"] == "TR93": continue
    print(f" {r['Policy']:<55} Mean={r['Mean']:.3f}")
```

12 Discussion and limitations

This rewrite (a) threads the actual fitted models from the linear- and non-linear-environment notebooks of this project (via `%run`) into the RL environments rather than re-fitting lightweight copies; (b) implements the agent setup of Hinterlang & Tanzer (2021, Section 2.1.2) and the training procedure of Section 2.2 faithfully, including the linear and nonlinear actor variants, the two observation specifications, the hidden-node search, the agent-selection criterion, and the $M = 500$ episode budget; and (c) adds a partial replication of the Hinterlang & Tanzer (2021, Section 3.4) DSGE robustness exercise using the Rudebusch & Svensson (1999) backward-looking benchmark.

Faithful to H&T: - All four economy environments use the parent notebooks' fitted models directly. - The agent's action i_t is correctly mapped to i_{t-1} in the *next* period's transition equations, with the previous rate becoming i_{t-2} . - Linear and nonlinear actor variants implemented as `LinearActor` and `NonlinearActor` overrides of SB3's `TD3Actor`, with H&T eq. 12-13 semantics and no extraneous output `tanh` squash. - Both observation specifications $x_t^{(1)}$ and $x_t^{(2)}$ trained for every applicable cell. - Hidden-node search over critic and (nonlinear) actor nodes; agent selection by the *ER/ES* filter followed by the steady-state-reward tie-break.

Pragmatic deviations: - Default architecture grid (`CRITIC_NODE_GRID = (1, 2)` and `ACTOR_NODE_GRID_NONLINEAR = (1, 4, 8)`) is narrower than H&T's full $\{1, \dots, 10\}$. `FULL_GRID = True` recovers the full grid at an order-of-magnitude longer wall-clock time. - The TVP-SVAR-SV environment uses the smoother-based posterior mean at the training endpoint (2007Q2). Because the parent notebook fits the TVP on the full sample including out-of-sample data, this anchor incorporates a small amount of post-2007 information. - SAC is trained with a fixed $[64, 64]$ architecture rather than a hidden-node search, because SAC's stochastic squashed Gaussian policy is not naturally compatible with H&T's linear variant or with single-hidden-node configurations.

DSGE exercise limitations: - We implement only the Rudebusch & Svensson (1999) backward-looking benchmark. The 10 forward-looking models in H&T's exercise require solving rational-expectations equilibria via Sims's (2002) `gensys` or an equivalent, which we do not embed here. A full replication is a follow-on contribution. - The nonlinear policies (DDPG nonlinear, SAC, Q-learning) are evaluated in the DSGE exercise using an *OLS-linearised* approximation of their reaction functions. This is consistent with how H&T evaluate their nonlinear ANN policies in Table 7 (the DSGE comparison requires linear feedback), but it discards the precise nonlinearities of the learned policies.

13 References (APA)

13.1 Software

- Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3), 90-95.
- Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Rio, J. F., Wiebe, M., Peterson, P., ... Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825), 357-362.
- McKinney, W. (2010). Data structures for statistical computing in Python. In *Proceedings of the 9th Python in Science Conference* (pp. 56-61).

- Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., ... Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32* (pp. 8024-8035).
- Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., & Dormann, N. (2021). Stable-Baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268), 1-8.
- Salvatier, J., Wiecki, T. V., & Fonnesbeck, C. (2016). Probabilistic programming in Python using PyMC3. *PeerJ Computer Science*, 2, e55.
- Seabold, S., & Perktold, J. (2010). statsmodels: Econometric and statistical modeling with Python. In *Proceedings of the 9th Python in Science Conference* (pp. 92-96).
- Sunder, N. (2025). *fedfred: A Python interface to the FRED API*. Zenodo. <https://github.com/nikhilxsunder/fedfred>
- Towers, M., Terry, J. K., Kwiatkowski, A., Balis, J. U., de Cola, G., Deleu, T., Goulao, M., Kallinteris, A., KG, A., Krimmel, M., Perez-Vicente, R., Pierre, A., Schulhoff, S., Tai, J. J., Shen, A. T. J., & Younis, O. G. (2023). Gymnasium. Zenodo. <https://github.com/Farama-Foundation/Gymnasium>
- Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., ... van Mulbregt, P. (2020). SciPy 1.0: Fundamental algorithms for scientific computing in Python. *Nature Methods*, 17(3), 261-272.

13.2 Papers & Books

- Adam, K., & Billi, R. M. (2006). Optimal monetary policy under commitment with a zero bound on nominal interest rates. *Journal of Money, Credit and Banking*, 38(7), 1877-1905.
- Bellman, R. (1957). *Dynamic Programming*. Princeton University Press.
- Cogley, T., & Sargent, T. J. (2005). Drifts and volatilities: Monetary policies and outcomes in the post WWII US. *Review of Economic Dynamics*, 8(2), 262-302.
- Fraccaro, M., Sonderby, S. K., Paquet, U., & Winther, O. (2016). Sequential neural models with stochastic layers. In *Advances in Neural Information Processing Systems 29* (pp. 2199-2207).
- Giacomini, R., & White, H. (2006). Tests of conditional predictive ability. *Econometrica*, 74(6), 1545-1578.
- Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics* (pp. 249-256).
- Goodfriend, M. (2004). Inflation targeting in the United States? In B. S. Bernanke & M. Woodford (Eds.), *The Inflation-Targeting Debate* (pp. 311-352). University of Chicago Press.
- Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *Proceedings of the 35th International Conference on Machine Learning* (Vol. 80, pp. 1861-1870).
- Hawkins, R. J., Speakes, J. K., & Hamilton, D. E. (2015). Monetary policy and PID control. *Journal of Economic Interaction and Coordination*, 10(1), 183-197.
- Hinterlang, N., & Tanzer, A. (2021). Optimal monetary policy using reinforcement learning. *Deutsche Bundesbank Discussion Paper No. 51/2021*.

- Howard, R. A. (1960). *Dynamic Programming and Markov Processes*. MIT Press.
- Krishnan, R. G., Shalit, U., & Sontag, D. (2017). Structured inference networks for nonlinear state space models. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence* (pp. 2101-2109).
- Levenberg, K. (1944). A method for the solution of certain non-linear problems in least squares. *Quarterly of Applied Mathematics*, 2(2), 164-168.
- Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., & Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.
- Lucas, R. E., Jr. (1976). Econometric policy evaluation: A critique. *Carnegie-Rochester Conference Series on Public Policy*, 1, 19-46.
- Marquardt, D. W. (1963). An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial and Applied Mathematics*, 11(2), 431-441.
- Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing Atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.
- Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540), 529-533.
- Nair, V., & Hinton, G. E. (2010). Rectified linear units improve restricted Boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning* (pp. 807-814).
- Nguyen, D., & Widrow, B. (1990). Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights. In *International Joint Conference on Neural Networks* (Vol. 3, pp. 21-26).
- Nikolsko-Rzhevskyy, A., Papell, D. H., & Prodan, R. (2018). The Taylor principles. *Journal of Macroeconomics*, 55, 14-30.
- Orphanides, A., & Wieland, V. (2000). Inflation zone targeting. *European Economic Review*, 44(7), 1351-1387.
- Primiceri, G. E. (2005). Time varying structural vector autoregressions and monetary policy. *Review of Economic Studies*, 72(3), 821-852.
- Rangapuram, S. S., Seeger, M., Gasthaus, J., Stella, L., Wang, Y., & Januschowski, T. (2018). Deep state space models for time series forecasting. In *Advances in Neural Information Processing Systems 31* (pp. 7796-7805).
- Rotemberg, J. J., & Woodford, M. (1997). An optimization-based econometric framework for the evaluation of monetary policy. *NBER Macroeconomics Annual*, 12, 297-346.
- Rudebusch, G. D., & Svensson, L. E. O. (1999). Policy rules for inflation targeting. In J. B. Taylor (Ed.), *Monetary Policy Rules* (pp. 203-262). University of Chicago Press.
- Saxe, A. M., McClelland, J. L., & Ganguli, S. (2014). Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. In *International Conference on Learning Representations*.
- Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., & Riedmiller, M. (2014). Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on Machine Learning* (pp. 387-395).
- Sims, C. A. (2002). Solving linear rational expectations models. *Computational Economics*, 20(1-2), 1-20.
- Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2nd ed.). MIT Press.

- Svensson, L. E. O. (2020). Monetary policy strategies for the Federal Reserve. *International Journal of Central Banking*, 16(1), 133-193.
- Taylor, J. B. (1993). Discretion versus policy rules in practice. *Carnegie-Rochester Conference Series on Public Policy*, 39, 195-214.
- Wang, J., Feinstein, A., & Chen, M. (2026). Reinforcement learning in monetary policy: A comparison of algorithmic approaches. *Stanford University Working Paper*.
- Watkins, C. J. C. H. (1989). *Learning from delayed rewards* (Doctoral dissertation). King's College, University of Cambridge.
- Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. *Machine Learning*, 8(3-4), 279-292.
- Wieland, V., Cwik, T., Müller, G. J., Schmidt, S., & Wolters, M. (2012). A new comparative approach to macroeconomic modeling and policy analysis. *Journal of Economic Behavior & Organization*, 83(3), 523-541.
- Wieland, V., Afanasyeva, E., Kuete, M., & Yoo, J. (2016). New methods for macro-financial model comparison and policy analysis. In J. B. Taylor & H. Uhlig (Eds.), *Handbook of Macroeconomics* (Vol. 2, pp. 1241-1319). Elsevier.
- Woodford, M. (2003). *Interest and Prices: Foundations of a Theory of Monetary Policy*. Princeton University Press.