机器学习方法求解 Krusell-Smith 模型

李博

2025 年 11 月 4 日

Krusell-Smith 模型问题描述

- **目标**：求解一个动态经济模型（Krusell-Smith），其主要特征包括：
  - 异质性代理人（在资产 a 和生产率 z 上存在差异）。
  - 总体不确定性（aggregate uncertainty，例如 TFP 冲击）。
  - 资产分布作为状态变量。
- **挑战**：高维状态空间（个体状态 + 分布）。传统的基于网格的方法面临维度灾难问题。
- **方法**：采用神经网络 (NN) 近似策略函数和价值函数。基于贝尔曼方程原理的迭代算法。
- **实现**：使用 Python 和 PyTorch 实现神经网络模型。模块化设计，便于扩展和维护。

Krusell-Smith 模型原问题

The role of the aggregate state is to allow the consumer to predict future prices. His optimization problem can therefore be expressed as

$$v(\epsilon, k; z, \Gamma) = \max_{k'} u(c) + \beta E v[(\epsilon', k'; z', \Gamma')|\epsilon, z] \tag{1}$$

subject to

$$c + k' = r(\bar{k}, \bar{l}, z)k + w(\bar{k}, \bar{l}, z)\epsilon\bar{l} + (1 - \delta)k \tag{2}$$

$$\Gamma' = H(\Gamma, z, z') \tag{3}$$

$$k' \geq 0 \tag{4}$$

Krusell-Smith 模型近似问题

The agent then solves the following problem:

$$v(\epsilon, k; z, \bar{k}) = \max_{c,k'} u(c) + \beta E v[(\epsilon', k'; z', \bar{k}')|\epsilon, z] \tag{5}$$

subject to

$$c + k' = r(\bar{k}, \bar{l}, z)k + w(\bar{k}, \bar{l}, z)\epsilon\widetilde{l} + (1 - \delta)k \tag{6}$$

$$log(\bar{k}') = a_0(z) + a_1(z)\log(\bar{k}) \tag{7}$$

$$log(\bar{k}') = b_0(z) + b_1(z)\log(\bar{k}) \tag{8}$$

$$k' \geq 0$$

整体算法逻辑

1. **初始化阶段**：
   - 加载配置参数。
   - 初始化神经网络，包括策略网络 f_p 和价值网络 f_v。
   - 可选步骤：加载预训练模型。
   - 定义资产分布矩的初始估计值。
2. **迭代循环（外循环）**：重复执行 `num_iter` 次迭代，具体步骤如下：
   (a) **策略函数训练 (`policy_bellman_training`)**：
      - 在给定当前价值网络 f_v 和分布矩的前提下进行。
      - 采样状态（包括个体状态和分布状态）。
      - 使用当前策略网络 f_p 进行前向模拟，以估计期望的未来价值（基于 f_v）。
      - 更新策略网络 f_p，以最大化模拟的贝尔曼方程右侧（RHS）。

   (b) **价值函数训练 (`value_training`)**：
      - 在给定更新后的策略网络 f_p 和分布矩的前提下进行。
      - 采样状态。
      - 使用更新后的 f_p 进行前向模拟，以生成目标价值（贝尔曼方程 RHS）。
      - 通过监督学习更新价值网络 f_v，以拟合生成的目标价值。

   (c) **分布更新**：模拟经济动态以更新资产（状态）分布（通过 `sim_path` 或隐式地通过 `calculate_G_batch` 实现）。
3. **最终模拟与分析**：使用最终训练完成的模型进行动态分析，并绘制相关结果。

代码结构概览

项目由几个 Python 文件和一个配置文件组成：

- `config_v1.json`:
  配置文件，包含所有参数（超参数、模型参数、网格、边界）。
- `dashboard_v1.py`:
  主执行脚本。协调加载、训练和模拟过程。
- `module_basic_v1.py`:
  定义基础组件：
  - 配置加载 (‘Config‘)。
  - 神经网络架构 (‘MyModel‘)。
  - 状态空间采样 (‘DomainSampling‘)。
  - 辅助函数（归一化、绘图）。
- `module_obj_bellman_v1.py`:
  定义基于贝尔曼方程的目标函数和模拟逻辑 (‘define_objective‘)。对于计算价值和状态转移至关重要。
- `module_training_bellman_v1.py`:
  实现策略和价值函数的训练循环 (‘EqumTrainer‘)。

配置文件: config_v1.json

目标文件: config_v1.json
该 JSON 文件集中管理模型及训练过程的所有参数。

- **模型扩展与保存:** 包括用于加载与保存模型的参数 (model_number_input, model_number_output, i_pretrainted, i_save)。
- **训练超参数:** 包括迭代次数 (num_iter)，训练轮数 (num_epochs_v, num_epochs_p)，样本量 (num_samples_policy, num_samples_value)，批大小，学习率 (lr_value, lr_policy)，以及目标函数内部模拟步数 (n_v_sim, n_p_sim)。
- **神经网络架构:** 包括输入与输出维度 (n_input, n_p_output, n_v_output)，隐藏层大小 (n1_p, n2_p, n1_v, n2_v)，分布维度 (k_dist)，以及旧模型参数 (_old)。
- **经济参数:** 包括偏好参数 (beta, sigma, alpha, psi_l, theta_l)，技术参数 (delta, TFP 过程参数 rho_tfp, eps_tfp, tfp_grid, tfp_transition)，以及生产率冲击参数 (rho_z, mu_z, sigma_z)。

配置文件: config_v1.json

目标文件: config_v1.json

- **状态空间边界:** 定义状态变量 z 和 a 的最小值与最大值 (bounds)，以及相关的惩罚参数 (lambda_penalty, a_pdf_penalty)。
- **分布网格与概率密度函数:** 包括用于表示或采样分布的离散点 (dist_a_mid, dist_z_mid) 及其对应的概率或密度 (dist_a_pdf, dist_z_pdf)，以及采样带宽 (dist_a_band)。
- **模拟参数:** 包括蒙特卡洛样本数量 (eu_samples)、路径模拟的长度 (n_sim_path, n_sim_step)，以及预烧期长度 (n_burn)。

module_basic_v1.py: 基础构建块

该模块提供基本的类和函数。

Config 类

目标文件: module_basic_v1.py - 类 Config

- 读取 config_v1.json 文件。
- 使参数可以通过属性访问（例如 config.beta）。

MyModel 类

目标文件: module_basic_v1.py - 类 MyModel

- 使用 torch.nn.Module 定义神经网络架构。
- 包含两个主要的序列网络:

  policy_func: 将状态映射到策略动作（例如储蓄率）。使用 ReLU 激活函数和最终的 Sigmoid（将输出限制在 0 和 1 之间）。
  value_func: 将状态映射到价值。ReLU 激活函数和最终的线性层。
- 提供 f_policy(x) 和 f_value(x) 方法来分别调用相应的网络。

module_basic_v1.py: 采样与工具函数

DomainSampling 类

目标文件: module_basic_v1.py - 类 DomainSampling

- 负责从状态空间生成样本。
- 状态包括：个体生产率 (z), 个体资产 (a), 以及资产分布表示 (dist_a_pdf)。
- 方法 generate_samples(num_samples, num_k):

  从有界的对数正态分布中采样 z。
  在其边界内均匀采样 a。
  在配置的均值 (config.dist_a_pdf) 周围的一个带宽 (config.dist_a_band) 内采样 dist_a_pdf 的分量。
  归一化最后 num_k 列（分布）使其和为 1。
- 其他方法: generate_samples_a_pdf, dist_enforce_boundaries。

module_basic_v1.py: 采样与工具函数

工具函数

目标文件: module_basic_v1.py - 函数

- normalize_inputs(inputs, bounds): 根据配置的边界对状态变量进行归一化处理（通常归一化至 [0, 1] 区间）。此步骤为神经网络输入的必要预处理环节。
- plot_equm_funcs: 用于生成策略函数与价值函数的诊断图的类。
- enforce_boundary, bounded_log_normal_samples: 辅助函数，用于处理边界约束及生成有界对数正态分布样本。

伪代码: generate_samples I

目标: module_basic_v1.py - 类 DomainSampling, 方法 generate_samples

procedure 生成样本 (N, k) ▷ N: 样本数量, k: 分布维度
    初始化空张量 'samples', 其大小为 (N, n_{states})
    从配置文件中获取 z 和 a 的边界
    从配置文件中获取基础分布 'dist_a_pdf' 和带宽 'dist_a_band'
    定义范围: ['z_bounds', 'a_bounds', ('pdf_i' ± band) for $i = 1..k$]
    基于 'mu_z' 和 'sigma_z' 定义 z 的对数正态分布
    for $i = 1$ to N **do** ▷ 生成 z 样本
        repeat
            从对数正态分布中采样 'z_candidate'
        until 'z_candidate' 落在 'z_bounds' 范围内
        将 'z_candidate' 赋值给 'samples[i, 0]'
    end for
    for $j = 1$ to $n_{states} - 1$ **do** ▷ 生成其他状态变量
        从 'ranges' 中获取状态 j 的 'lower' 和 'upper' 边界
        将均匀分布随机数 ('lower', 'upper', size=N) 赋值给 'samples[:, j]'
    end for

伪代码: generate_samples II

提取 ‘samples‘ 的最后 k 列，记为 ‘last_k_cols‘
计算 ‘last_k_cols‘ 的按行求和，记为 ‘sums‘
将 ‘last_k_cols‘ 按行归一化，即 ‘samples[:, -k:]‘ = ‘last_k_cols‘ / ‘sums‘
return ‘samples‘
end procedure

module_obj_bellman_v1.py: 目标定义

该模块负责定义评估贝尔曼方程和模拟模型动态的核心逻辑。

define_objective 类

目标文件: module_obj_bellman_v1.py - 类 define_objective

- 保存当前的神经网络模型 (self.model) 和设备信息。
- 提供用于计算基于贝尔曼方程的目标值的方法。
- 核心方法包括:

  obj_sim_value: 用于向前模拟以计算期望折现效用。
  calculate_aggregates: 用于计算价格（工资、利率）和总量。
  calculate_G_batch: 用于计算资产分布的转移矩阵。
  expected_value_V: 用于计算期望延续价值。
  sim_path: 用于模拟长路径以进行分析。

关键方法详解 (define_objective 类) I

目标文件: `module_obj_bellman_v1.py` - 类 `define_objective`
该类包含了模型模拟与评估的核心逻辑。以下是几个关键方法的说明:

关键方法详解 (define_objective 类) II

obj_sim_value(X_batch, N_sim, ...)

- **目的:** 针对一批给定的初始状态 X_batch，计算在当前策略网络 (PolicyNet) 下进行 N_sim 步模拟所能获得的期望总折现效用，并生成用于训练价值网络 (ValueNet) 的目标数据。
- **核心逻辑:**
  1. 从初始状态开始，迭代执行 N_sim 次模拟。
  2. 在每一步 t: 利用 PolicyNet 确定下一期资产 a_{t+1}，计算当前的总量（工资 w_t, 利率 r_t），消费 c_t 和劳动 l_t，并计算当期效用 $u(c_t, l_t)$。
  3. 模拟下一期的个体生产率 z_{t+1} 和总体 TFP tfp_{t+1}。
  4. （可选）通过 calculate_G_batch 更新资产分布 $dist_{t+1}$。
  5. 累加折现后的当期效用 $\beta^t u_t$，同时考虑并累加可能的惩罚项（如消费过低或分布越界）。
  6. 在 N_{sim} 步结束后，利用 expected_value_V 计算期望延续价值 V_{cont}。
- **输出:**

  用于策略优化的目标值（模拟的总折现效用，通常为其负均值）。
  用于价值函数拟合的 (状态, 目标价值) 数据对。
- **本质:** 实现了贝尔曼方程右侧的计算，既可用于评估当前策略（生成价值目标），也可作为优化策略本身的目标函数。

关键方法详解 (define_objective 类) (续) I

calculate_G_batch(dist_a_mid, ..., dist_0, tfp_0)

- **目标:** 计算资产分布的内生转移矩阵 G。矩阵元素 G_{ij} 表示，在给定当前资产分布 $dist_0$ 和 TFP tfp_0 的条件下，处于资产区间 i 的代理人在下一期转移至资产区间 j 的概率。
- **核心逻辑:**
  1. 依赖于当前的策略函数 `PolicyNet`。
  2. 针对每个目标资产区间 j 的边界，反向求解需要多高的个体生产率 z，才能使得遵循策略函数所得到的下一期资产 a' 恰好落在该边界上（利用 `find_x_z0_linear` 的线性近似）。
  3. 利用个体生产率 z 的已知分布（对数正态 CDF），计算出落入步骤 2 中求解出的 z 值范围内的概率，即为 G_{ij}。
  4. 对每个起始区间 i 的转移概率进行归一化，以确保 $\sum_j G_{ij} = 1$。
- **输出:** 状态依赖的资产分布转移矩阵 G。
- **本质:** 该方法揭示了异质性代理人模型中，个体决策（由策略函数决定）如何加总并影响宏观分布动态的关键机制。

关键方法详解 (define_objective 类) (续) II

`expected_value_V(z1, a1, dist1, tfp1)`

- **目的:** 计算期望延续价值 $\mathbb{E}[V(z', a_1, dist_1, tfp_1)]$。即，在已知下一期状态中的资产 a_1、分布 $dist_1$ 和 TFP tfp_1 的情况下，对下一期未知的个体生产率冲击 z' 取期望，得到价值函数的值。
- **核心逻辑:**
  1. 使用配置文件中定义的离散生产率网格 Z_{grid} 及其对应的概率 Z_{pdf}。
  2. 对于每个可能的 $z' \in Z_{grid}$，构建完整的状态向量 $(z', a_1, dist_1, tfp_1)$。
  3. 将该状态向量输入当前的价值网络 `ValueNet`，得到预测值 $V(z', a_1, dist_1, tfp_1)$。
  4. 使用 Z_{pdf} 对所有可能的 z' 的预测值进行加权平均。
- **输出:** 单个期望价值数值。
- **本质:** 在计算贝尔曼方程右侧时，处理未来个体不确定性的标准方法。

关键方法详解 (define_objective 类) (续) III

`sim_path(X_batch, N_sim, ...)`

- **目的:** 从一个初始状态 `X_batch` 开始，使用训练好的策略函数 `PolicyNet` 和价值函数 `ValueNet` (虽然价值函数在此处不直接用于决策)，模拟经济系统随时间演变的路径，用于分析和可视化。
- **核心逻辑:** 与 `obj_sim_value` 类似，但不以计算总效用为目标，而是记录每一步的状态变量（$z_t, a_t, dist_t, tfp_t$）以及计算出的总量（w_t, r_t, Y_t 等）。
- **输出:** 各个状态变量和总量的时间序列数据，并直接调用绘图函数生成路径图。
- **本质:** 模型验证和结果展示的重要工具，用于观察模型是否能产生合理的宏观和微观动态。

伪代码: obj_sim_value I

目标: `module_obj_bellman_v1.py` - 类 `define_objective`, 方法 `obj_sim_value`, 目的向前模拟以计算期望折现效用

procedure ObjSimValue(X_{batch}, N_{sim}, $dist\_a\_mid$, $dist\_a\_mesh$)
    从 X_{batch} 提取初始状态 $z_0, a_0, dist_0, tfp_0$
    初始化 $V_{sim\_accum} = 0$, $V_{fit\_accum} = 0$
    for $t = 0$ to $N_{sim} - 1$ **do**
        对当前状态 $(tfp_t, z_t, a_t, dist_t)$ 进行归一化处理，得到 X_{norm}
        预测策略: $a_{t+1\_norm} = \text{PolicyNet}(X_{norm})$
        对策略进行反归一化: $a_{t+1} = a_{t+1\_norm} \times (a_{max} - a_{min})$
        计算总量: $w_t, l_t, r_t = \text{CalculateAggregates}(tfp_t, z_t, dist_t, dist\_a\_mid)$
        计算消费: $c_t = (1 + r_t)a_t + w_t l_t z_t - a_{t+1}$
        若 $c_t < \epsilon$，则施加惩罚: $c_{punish} = \max(0, \epsilon - c_t)/\epsilon$
        $c_t = \max(c_t, \epsilon)$
        计算当期效用: $u_t = \text{Utility}(c_t, l_t)$
        模拟下一期生产率: $z_{t+1} \sim$ 有界对数正态分布(μ_z, σ_z)
        模拟下一期 TFP: $tfp_{t+1} \sim$ 转移概率(tfp_t)
        if 需要更新分布 (`config.i_dist == 1`) **then**

伪代码: obj_sim_value II

计算转移矩阵:
$G_t = \text{Calculate\_G\_batch}(dist\_a\_mid, dist\_a\_mesh, dist_t, tfp_t)$
更新分布: $dist_{t+1} = dist_t^T \times G_t$
else
采样下一期分布: $dist_{t+1} \sim \text{SamplePDF}(k)$
end if
强制分布边界并获取惩罚: $dist_{t+1}, dist_{punish} = \text{EnforceDistBounds}(dist_{t+1})$
总惩罚: $P_t = c_{punish} + dist_{punish}$
累积价值:
$V_{sim\_accum} = V_{sim\_accum} + \beta^t(u_t - P_t)$
$V_{fit\_accum} = V_{fit\_accum} + \beta^t u_t$
更新下一期状态: $(z_t, a_t, dist_t, tfp_t) \leftarrow (z_{t+1}, a_{t+1}, dist_{t+1}, tfp_{t+1})$
end for
计算期望延续价值: $V_{cont} = \text{ExpectedValueV}(z_{N_{sim}}, a_{N_{sim}}, dist_{N_{sim}}, tfp_{N_{sim}})$
最终模拟价值: $V_{sim\_final} = V_{sim\_accum} + \beta^{N_{sim}} V_{cont}$
最终拟合价值目标: $V_{fit\_final} = V_{fit\_accum} + \beta^{N_{sim}} V_{cont}$
创建价值数据: $ValueData = \text{concat}(X_{batch}, V_{fit\_final})$

伪代码: obj_sim_value III

return $-\text{mean}(V_{sim\_final})$, $ValueData$ ▷ 返回负均值以便进行最大化

end procedure

伪代码: calculate_aggregates

目标: `module_obj_bellman_v1.py` - 类 `define_objective`, 方法 `calculate_aggregates`, 目的 计算价格（工资、利率）和总量

procedure CalculateAggregates($tfp_0, z_0, dist_0, dist\_a\_mid$)
    计算总资本: $K_0 = \sum(dist_0 \times dist\_a\_mid)$
    计算积分项: $IntZ = \mathbb{E}[z^{1+1/\theta_l}]$ （若 z 的分布固定，可预先计算）
    计算中间工资项: $w_{0\_1} = (1-\alpha)(K_0/IntZ)^{\alpha}$
    计算工资: $w_0 = tfp_0 \cdot \psi_l^{\alpha/\theta_l} \cdot w_{0\_1}^{\theta_l/(\alpha+\theta_l)}$
    计算劳动供给: $l_0 = (w_0 z_0/\psi_l)^{1/\theta_l}$
    计算利率: $r_0 = tfp_0 \cdot \alpha(w_0/(1-\alpha))^{(\alpha-1)/\alpha} - \delta$
    计算产出（近似值）: $Y_0 = tfp_0 \cdot K_0^{\alpha} \cdot (\mathbb{E}[l_0 z_0])^{1-\alpha}$ （简化形式，代码可能略有不同）
    return w_0, l_0, r_0, Y_0
end procedure

伪代码: calculate_G_batch I

目标: module_obj_bellman_v1.py - 类 define_objective, 方法 calculate_G_batch, 目的 计算资产分布的转移矩阵

procedure CALCULATE_G_BATCH($dist\_a\_mid, dist\_a\_mesh, dist_0, tfp_0$) ▷ 计算转移矩阵 G[i, j] = Prob(a' 在区间 j | a 在区间 i)
    $N_{bins} = \text{length}(dist\_a\_mid)$
    初始化 G 矩阵，大小为 $(N_{batch}, N_{bins}, N_{bins})$
    基于 $dist\_a\_mid$ 和 $dist\_a\_mesh$ 定义区间边界
    for 每个起始区间 $i = 1$ 到 N_{bins} **do**
        for 每个结束区间 $j = 1$ 到 N_{bins} **do**
            确定 z_{upper}，使得 PolicyNet($z_{upper}, a_i, dist_0, tfp_0$) 映射到区间 j 的上边界
            确定 z_{lower}，使得 PolicyNet($z_{lower}, a_i, dist_0, tfp_0$) 映射到区间 j 的下边界
            计算概率质量: $P_{ij} = \text{CDF}_{LogNormal}(z_{upper}) - \text{CDF}_{LogNormal}(z_{lower})$
            $G[:, i, j] = \max(0, P_{ij})$
        end for
    end for
    对 G 的行进行归一化: $G[:, i, :] = G[:, i, :] / \sum_j G[:, i, j]$ ▷ 处理除零问题
    return G

伪代码: calculate_G_batch II

15: **end procedure**

注意: *find_x_z0_linear* 方法假设策略在 z 上局部线性，以确定所需的 z 值。

伪代码: expected_value_V

目标: `module_obj_bellman_v1.py` - 类 `define_objective`, 方法 `expected_value_V`, 目的 计算期望延续价值

procedure ExpectedValueV($z_1, a_1, dist_1, tfp_1$) ▷ 计算 $\mathbb{E}[V(z', a_1, dist_1, tfp_1)]$

从配置文件中获取离散生产率网格 Z_{grid} 和其对应的概率 Z_{pdf}

初始化期望值 $E\_V = 0$

$N_{z\_pts} = \text{length}(Z_{grid})$

for $k = 1$ to N_{z_pts} **do**

构造状态向量 $X_k = (Z_{grid}[k], a_1, dist_1)$

对状态进行归一化处理: $X_{k\_norm} = \text{NormalizeInputs}(X_k)$

添加 TFP 信息: $X_{k\_norm\_tfp} = \text{concat}(tfp_1, X_{k\_norm})$

预测价值: $V_k = \text{ValueNet}(X_{k\_norm\_tfp})$

更新期望值: $E\_V = E\_V + Z_{pdf}[k] \times V_k$

end for

return E_V

end procedure

注意: 此方法用于计算在给定其他状态变量 $a_1, dist_1, tfp_1$ 的条件下, 对下一期个体生产率冲击 z' 的期望价值。

module_training_bellman_v1.py: 训练逻辑

本模块包含用于训练神经网络的 EqumTrainer 类。

EqumTrainer 类

目标文件: module_training_bellman_v1.py - 类 EqumTrainer

- 通过训练超参数（包括训练轮数、学习率、批大小）、模型以及预训练权重的路径进行初始化。
- 负责加载预训练的策略网络和价值网络 (load_pretrained, get_pretrained_model)。
- 提供以下主要训练方法：

  policy_bellman_training: 用于训练策略网络。
  value_training: 用于训练价值网络。
- 包含用于绘制损失曲线 (update_graph) 和保存模型的辅助功能。

训练流程概述 (EqumTrainer)

目标文件: `module_training_bellman_v1.py` - 类 `EqumTrainer`
`EqumTrainer` 类通过两个核心方法协调策略和价值网络的训练:

策略函数训练 (policy_bellman_training)

- **目标:** 确定最优的策略函数 π (由 `PolicyNet` 表示)，以使得在给定状态下，选择该策略能够最大化期望的未来总折现效用。
- **方法:** 基于策略迭代的思想。
- **流程:**
  1. **固定价值网络:** 假设当前的价值网络 `ValueNet` (f_v) 是准确的。
  2. **采样状态:** 从状态空间中抽取一批状态 X_{batch}。
  3. **模拟与评估:** 对于每个状态，利用当前的策略网络 `PolicyNet` (f_p) 向前模拟 N_{sim} 步，并结合固定的 `ValueNet` 计算得到的期望总折现效用（通过调用 `obj_sim_value`，该函数内部计算贝尔曼方程右侧的值)。
  4. **优化策略网络:** 调整 `PolicyNet` 的参数，以最大化上述步骤中计算出的期望总折现效用（即最大化贝尔曼方程右侧)。这一过程通常通过梯度上升（或最小化其负值）实现。
- **本质:** 使策略网络能够学习选择带来更高长期价值（由当前价值网络评估）的行动。

训练流程概述 (EqumTrainer)

价值函数训练 (value_training)

- **目标:** 训练价值网络 `ValueNet` (f_v)，以使其能够准确预测在给定状态下，遵循当前策略 π (由 `PolicyNet` 表示) 所能带来的期望总折现效用。
- **方法:** 基于拟合价值迭代 (Fitted Value Iteration) 的思想进行训练。
- **流程:**
  1. **固定策略网络:** 假定当前的策略网络 `PolicyNet` (f_p) 是已知的。
  2. **采样状态与生成目标:** 从状态空间中抽取一批状态 X_{data_gen}。对于这些状态，利用固定的 `PolicyNet` 进行 N_{sim} 步的前向模拟，并计算实际获得的期望总折现效用（通过调用 `obj_sim_value`，重点关注其返回的 'ValueData'，即 (X, V_{target}) 对）。
  3. **监督学习:** 将步骤 2 生成的 (X, V_{target}) 对作为训练数据，训练 `ValueNet`，使其输出 $f_v(X)$ 尽可能接近目标值 V_{target}。这一过程通常通过最小化均方误差 (MSE) 实现。
- **本质:** 通过训练价值网络，使其能够准确评估当前策略的优劣程度。

伪代码: policy_bellman_training I

目标: `module_training_bellman_v1.py` - 类 `EqumTrainer`, 方法 `policy_bellman_training`

procedure PolicyBellmanTraining(N_{sim}, $dist\_a\_mid$, $dist\_a\_mesh$)
    加载预训练策略权重（如适用）
    初始化 PolicyNet 参数的 Adam 优化器
    初始化学习率调度器（例如 ReduceLROnPlateau）
    获取 Domain Sampler 和 Objective Updater 实例
    for epoch = 1 to N_{epochs_p} **do**
        生成训练样本 X_{train} = DomainSampler.generate_samples($N_{samples\_p}, k$)
        为 X_{train} 创建 DataLoader
        初始化 epoch loss = 0
        for 每个批次 X_{batch} in DataLoader **do**
                                                ▷ 目标函数返回模拟价值的负均值
            $Loss_{batch}, \_ =$ ObjectiveUpdater.obj_sim_value($X_{batch}, N_{sim}, dist\_a\_mid, dist\_a\_mesh$)
            梯度清零: Optimizer.zero_grad()
            反向传播: $Loss_{batch}$.backward()

伪代码: policy_bellman_training II

15: 更新权重: Optimizer.step()
16: 累积损失（可选，用于监控）
17: **end for**
18: 基于 epoch loss 更新学习率调度器
19: 绘制或记录损失
20: **end for**
21: 保存训练完成的 PolicyNet 权重（如适用）
22: **return** 更新后的模型
23: **end procedure**

注意: 此方法通过最大化从 *obj_sim_value* 获得的模拟价值来训练策略网络，有效地利用贝尔曼方程执行策略迭代。

伪代码: value_training I

目标: `module_training_bellman_v1.py` - 类 `EqumTrainer`, 方法 `value_training`

procedure ValueTraining(N_{sim}, *dist_a_mid*, *dist_a_mesh*)
    加载预训练价值权重（如适用）
    初始化 ValueNet 参数的 Adam 优化器
    初始化学习率调度器
    定义损失函数（例如 MSELoss）
    获取 Domain Sampler 和 Objective Updater 实例
    for epoch = 1 to N_{epochs_v} **do**
        if epoch % N_{epochs_draw} == 0 **then** ▷ 周期性生成新的目标数据
            生成样本 X_{data_gen} = DomainSampler.generate_samples($N_{samples\_v} \times N_{epochs\_draw}, k$)
            冻结 PolicyNet 梯度
            计算目标价值: _, $ValueData_{all}$ = ObjectiveUpdater.obj_sim_value(X_{data_gen}, N_{sim}, *dist_a_mid*, *dist_a_mesh*)
            解冻 PolicyNet 梯度（如适用）
        end if
        从 $ValueData_{all}$ 中为当前 epoch 选择子集 $ValueData_{epoch}$

伪代码: value_training II

从 $ValueData_{epoch}$ 提取输入 X_{epoch} 和目标 Y_{epoch}
为 (X_{epoch}, Y_{epoch}) 创建 DataLoader
初始化 epoch 损失为 0
for 每个批次 (X_{batch}, Y_{batch}) in DataLoader **do**
    预测价值: $Y_{pred} = \text{ValueNet}(X_{batch})$
    计算损失: $Loss_{batch} = \text{MSELoss}(Y_{pred}, Y_{batch})$
    梯度清零: Optimizer.zero_grad()
    反向传播: $Loss_{batch}$.backward()
    更新权重: Optimizer.step()
    累积损失
end for
基于 epoch 损失更新学习率调度器
绘制或记录损失
end for
保存训练完成的 ValueNet 权重（如适用）
return 更新后的模型
end procedure

伪代码: value_training III

注意: 该方法是一种拟合价值迭代（*Fitted Value Iteration*）。通过当前策略（由 *PolicyNet* 提供）生成状态-价值对（通过 `obj_sim_value` 实现），然后使用监督学习训练 *ValueNet* 以拟合这些对。

dashboard_v1.py: 协调流程

目标文件: `dashboard_v1.py` - 主执行模块

该脚本负责整合所有模块并协调其运行:

- 导入必要的模块以及自定义代码。
- 使用 `Config` 类加载配置文件。
- 设置计算设备（CPU 或 GPU）。
- 实例化 `MyModel` 类，并在多 GPU 环境下进行适配（如适用）。
- 定义辅助函数 `instantiate_trainer`，用于创建 `EqumTrainer` 实例，并根据配置文件管理预训练模型的加载。
- 定义分布网格点 (`dist_a_mid`)。
- ** 主迭代循环:** 执行核心算法，循环次数为 `config.num_iter`。
- 在循环内部:
  - 调整分布采样带宽 (`dist_a_band`)。
  - 针对首次迭代可能进行特殊处理（例如，执行初始模拟）。
  - 实例化或重新实例化 `EqumTrainer`。
  - 调用 `trainer.policy_bellman_training(...)` 方法以训练策略网络。
  - 调用 `trainer.value_training(...)` 方法以训练价值网络。
  - 调用 `plot_equm_funcs` 方法以可视化当前的函数。
- 在循环结束后，使用 `equm_updater.sim_path(...)` 方法执行最终模拟，以生成并绘制时间路径。

伪代码: dashboard_v1.py 中的主循环 I

目标: dashboard_v1.py - 主执行模块 (简化)

加载 config
设置 device (CPU/GPU)
初始化 model = MyModel(...)
将 model 移动到 device
加载 dist_a_mid 张量
for iter = 0 to config.num_iter - 1 **do**
    输出当前迭代次数: "iter:", iter
    调整 config.dist_a_band (如有必要)
    if iter == 0 **then**
        设置标志: i_save = 1, load_pretrained = config.i_pretrainted
        trainer = InstantiateTrainer(model, config.model_number_input, load_pretrained)
        model = trainer.get_pretrained_model() ▷ 加载初始权重
        updater = DefineObjective(model, device)
        sampler = trainer.get_domain_sampler()
        initial_data = sampler.generate_samples(...)

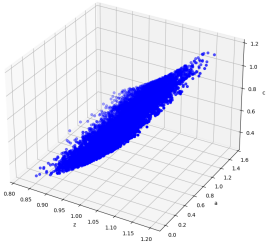
伪代码: dashboard_v1.py 中的主循环 II

```
        updater.sim_path(initial_data, ...)                    ▷ 可选的初始模拟
    else
        设置标志: i_save = 1, load_pretrained = 0 ▷ 后续迭代不再加载预训练模型
    end if
    trainer = InstantiateTrainer(model, config.model_number_input,
load_pretrained)                                               ▷ 基于当前模型实例化
                                                               ▷ 执行策略训练步骤
    model = trainer.policy_bellman_training(config.n_p_sim, dist_a_mid,
...)
                                                               ▷ 执行价值训练步骤
    model = trainer.value_training(config.n_v_sim, dist_a_mid, ...)
                                                               ▷ 生成绘图
    plotter = PlotEqumFuncs(model, ...)
    plotter.create_plot()
end for
                                                               ▷ 执行最终模拟
updater = DefineObjective(model, device)
```

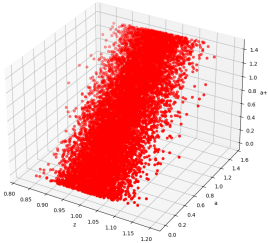
伪代码: dashboard_v1.py 中的主循环 III

```
sampler = trainer.get_domain_sampler()        ▷ 或从最后一个 trainer 实例获取
initial_data = sampler.generate_samples(...)
updater.sim_path(initial_data, config.n_sim_path, ...)
记录执行时间和内存使用情况
```

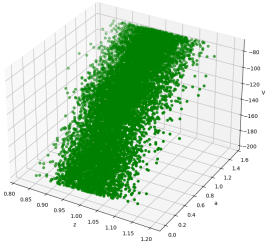
训练结果可视化: 策略与价值函数



消费策略 $c(z,a)$



下一期资产 $a'(z,a)$



价值函数 $V(z,a)$

注：图像展示了在固定资产分布下，个体状态 $(z,\ a)$ 与策略/价值函数的关系 (散点图)。

结论

- 本代码库实现了一种基于神经网络的方法，用于求解复杂的异质代理人模型。
- 该方法通过迭代过程，在策略改进 (`policy_bellman_training`) 和价值函数拟合 (`value_training`) 之间交替进行，并以贝尔曼方程为理论依据。
- 核心组成部分包括：
  - 灵活的神经网络架构 (`MyModel`)。
  - 复杂的状态空间采样机制 (`DomainSampling`)。
  - 核心的模拟与贝尔曼评估逻辑 (`define_objective.obj_sim_value`)。
  - 专门设计的训练流程 (`EqumTrainer`)。
  - 集中化的配置管理 (`config_v1.json`)。
- 神经网络的引入有效缓解了此类模型中固有的维度灾难问题。
- 状态转移的精确实现（包括通过 `calculate_G_batch` 进行的分布更新）在整个方法中具有重要作用。

文献综述-离散时

Maliar, L., Maliar, S., & Winant, P. (2021). Deep learning for solving dynamic economic models. Journal of Monetary Economics, 122, 76-101.

方法一：基于回报最大化的深度学习方法

核心思想

通过将求解 Krusell-Smith 模型转化为最大化**所有**异质性主体（代理人）预期终身效用总和 $E_0\left[\sum_{t=0}^{\infty}\beta^t u(c_{it})\right]$ 的优化问题，从而实现模型的求解。

损失函数（最大化等价于最小化负损失函数）

$$\Xi(\theta)=E_{\omega}[\xi(\omega;\theta)]=E_{(Y_0,W_0,z_0,\Sigma,\sigma)}[\sum_{0}^{T}\beta^t u(c_t)]$$

通过神经网络估计个体决策 $\frac{c_t^i}{w_t^i}=\psi(\cdot;\theta)$。

期望的处理方式

采用 AIO 方法，不直接给定个体具体的 Markov 转移矩阵参数，而是直接提供每一期可能受到的冲击数值大小，使得个体或群体通过大量冲击实验，学习并估计出具体的 Markov 转移矩阵。（以下方法同理）

方法二：基于欧拉方程残差最小化的深度学习方法

核心思想

将求解 Krusell-Smith 模型转化为一个**最小化**所有主体在所有状态下欧拉方程（一阶最优性条件）平均平方残差的优化问题。

- 欧拉方程:

$$u'(c_t^i) = \beta R_{t+1} E_\epsilon[u'(c_{t+1}^i)]$$

- 最小化欧拉方程残差（即欧拉方程左右两边的差异）：

$$\min\{\beta R_{t+1} E_\epsilon[u'(c_{t+1}^i)] - u'(c_t^i)\}$$

$$\min\{1 - \frac{\beta R_{t+1} E[u'(c')]}{u'(c)}\}$$

- Fischer-Burmeister (FB) 方程用于处理 KKT 条件:
  - 对于任意 $X \geq 0, Y \geq 0$ 且满足 $XY = 0$ 的条件，可重写为 FB 方程:

$$\Psi^{FB}(x, y) = x + y - \sqrt{x^2 + y^2}$$

  - 针对 Krusell-Smith 模型的 KKT 条件：$w_t^i - c_t^i \geq 0$ 和 $|u'(c) - \beta R_{t+1} E[u'(c')]| \geq 0$，可重写为:

$$x = 1 - \frac{c_t^i}{w_t^i}; y = 1 - \frac{\beta R_{t+1}^i E[u'(c_{t+1}^i)]}{u'(c_t^i)}$$

方法二：基于欧拉方程残差最小化的深度学习方法

- 神经网络估计
  - 损失函数 (最小化)

$$\Xi(\theta) = E_{\omega}[\xi(\omega;\theta)] = E_{(Y_t, W_t, z_t, \Sigma_1, \Sigma_2, \epsilon_1, \epsilon_2)}\{[\Psi(1 - \frac{c_t^i}{w_t^i}, 1 - h_t^i)]^2$$

$$+v[\frac{\beta R_{t+1} u'(c_{t+1}^i)|_{\Sigma=\Sigma_1, \epsilon=\epsilon_1}}{u'(c_t^i)} - h_{t]}^i][\frac{\beta R_{t+1} u'(c_{t+1}^i)|_{\Sigma=\Sigma_2, \epsilon=\epsilon_2}}{u'(c_t^i)} - h_{t]}^i]\}$$

  - 其中 v 是权重 $\Psi(\cdot)$ 是 FB 方程
  - 使用神经网络估计 $\frac{c_t^i}{w_t^i} = \psi(\cdot;\theta)$ 和 $h_t^i = h(\cdot;\theta)$

文献综述-离散时

Azinovic, M., Gaegauf, L., & Scheidegger, S. (2022). Deep equilibrium nets. International Economic Review, 63(4), 1471-1525.

DEQN I

核心思想

通过深度神经网络 (DNN) $N_\rho(x)$，以**无监督学习**的方式直接近似均衡函数 $\theta(x)$，旨在**最小化所有均衡条件**在模拟数据上的残差。

算法流程:

1. **目标定义:** 求解函数形式理性预期均衡 (FREE) $\theta(x)$，该均衡包含策略、价格、乘子等函数。
2. **网络近似:** 使用深度神经网络 $N_\rho(x)$ (参数为 ρ) 来近似 $\theta(x)$。
3. **损失函数构建 $\mathcal{L}(\rho)$:**
   - 基于模型的**均衡条件**（结合了 Euler 方程、Bellman 方程和市场出清条件）。
   - 计算在状态 x 下的各项残差:
     - 欧拉方程相对误差 (REE) $e^{\text{REE}}(x, \rho)$
     - KKT 条件 $e^{\text{KKT}}(x, \rho)$
     - 市场出清条件误差 $e^{\text{MC}}(x, \rho)$
     - (若存在递归效用) 贝尔曼方程误差 $e^{\text{Bellman}}(x, \rho)$

DEQN II

- 损失函数形式为：

$$\mathcal{L}(\rho) = \mathbb{E}_{x \sim D_{\text{train}}} \left[(e^{\text{REE}})^2 + (e^{\text{KKT}})^2 + (e^{\text{MC}})^2 + \dots \right]$$

其中，损失函数采用平均平方误差 (MSE)。

4. **迭代循环 (Iterative Loop):**
   a. **模拟 (Simulation):** 使用当前网络 $N_\rho(x)$ 模拟经济状态路径 $\{x_t\}_{t=1}^T$，生成训练数据集 D_{train}。此步骤计算成本较低。
   b. **训练 (Training):** 在 D_{train} 上，利用随机梯度下降 (SGD) 或其变种 (如 Adam) 计算损失函数梯度 $\nabla_\rho \mathcal{L}$，并更新网络参数 ρ:

$$\rho \leftarrow \rho - \alpha_{\text{learn}} \nabla_\rho \mathcal{L}(\rho)$$

5. **收敛:** 重复上述迭代步骤，直至损失函数 $\mathcal{L}(\rho)$ 收敛至足够小的值，并且均衡条件残差满足精度要求。

文献综述-离散时 (this method)

Han, J., & Yang, Y. (2021). Deepham: A global solution method for heterogeneous agent models with aggregate shocks. arXiv preprint arXiv:2112.14377.

DeepHAM 算法核心 I

核心思想

DeepHAM 算法通过深度神经网络 (DNN) 求解高维异质性主体模型，其关键在于引入**广义矩 (Generalized Moments)** Q_t，以有效且具有解释性地表示高维状态分布 $S_t = \{s_1, ..., s_N\}_t$，并利用 DNN 近似依赖于这些矩的价值函数和策略函数。

神经网络的算法求解步骤:

1. **状态分布的表示 (Representation of S_t):**
   - 引入一个深度神经网络 $Q_{NN}(\cdot; \Theta_Q)$ 作为**基函数**，将个体状态 s_i 映射为一个向量。
   - **广义矩** Q_t 定义为这些基函数在所有个体上的平均值:

$$Q_t = \frac{1}{N} \sum_{i=1}^{N} Q_{NN}(s_{it}; \Theta_Q) \in \mathbb{R}^{d_Q}$$

   - 参数 Θ_Q 是可训练的，能够使网络自动学习对决策最为重要的矩。
   - 使用两组独立的广义矩：Q_t^V (用于价值函数) 和 Q_t^C (用于策略函数)，其参数分别为 Θ_{VQ} 和 Θ_{CQ}。
2. **价值函数的近似 (Approximation of the Value Function V):**

DeepHAM 算法核心 II

- 使用另一个深度神经网络 $\tilde{V}_{NN}(\cdot;\Theta_{VO})$，其输入为**降维后**的状态 (s_{it}, X_t, Q_t^V)：

$$V_{NN}(s_{it}, X_t, S_t;\Theta_V) = \tilde{V}_{NN}(s_{it}, X_t, Q_t^V;\Theta_{VO})$$

  (其中 $\Theta_V = (\Theta_{VQ}, \Theta_{VO})$)
- **训练目标:** 最小化网络预测值 V_{NN} 与模拟路径上计算出的**实际累计折现效用** $\hat{V}_i$ 之间的均方误差 (MSE)，类似于监督学习：

$$\min_{\Theta_V} \mathbb{E}_{\mu(C)}\left[(V_{NN}(s_{i0}, X_0, S_0;\Theta_V) - \hat{V}_i)^2\right]$$

DeepHAM 算法核心 I

1. **近似策略函数 (Approximating Policy Function C):**
   - 使用 DNN $\tilde{C}_{NN}(\cdot;\Theta_{CO})$，输入为 (s_{it}, X_t, Q_t^C):

$$C_{NN}(s_{it}, X_t, S_t; \Theta_C) = \tilde{C}_{NN}(s_{it}, X_t, Q_t^C; \Theta_{CO})$$

   (其中 $\Theta_C = (\Theta_{CQ}, \Theta_{CO})$)
   - **训练目标:** 最大化智能体在模拟路径上的**有限期 T 期望效用 + 终端价值** (由当前 V_{NN} 给出):

$$\max_{\Theta_C} \mathbb{E}_{\mu(C)} \left[\sum_{t=0}^{T} \beta^t u(s_{it}, c_{it}) + \beta^T V_{NN}(s_{iT}, X_T, S_T; \Theta_V) \right]$$

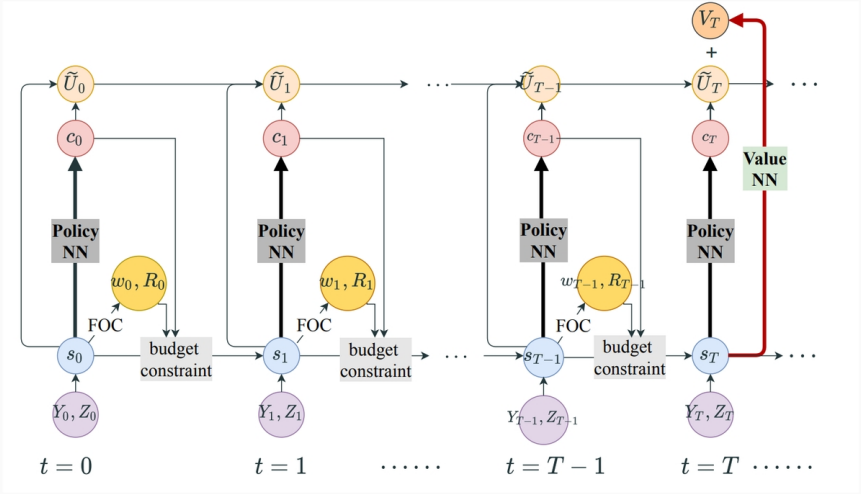
   (其中 $c_{it} = C_{NN}(s_{it}, X_t, S_t; \Theta_C)$)
2. **训练过程:**
   - 采用类似价值迭代的框架，交替更新价值网络 (Θ_V) 和策略网络 (Θ_C)。
   - 所有更新都基于从当前策略 C 模拟出的 T 期经济路径数据。
   - 使用随机梯度下降 (SGD) 或其变种优化 Θ_V 和 Θ_C (包括其中的矩参数 Θ_{VQ}, Θ_{CQ})。

DeepHAM 算法核心 II

关键特点

- 用**可学习的广义矩** Q_t 替代固定矩或完整分布，实现降维和解释性。
- 神经网络近似**价值函数**和**策略函数**，输入为 (s_i, X_t, Q_t)。
- 训练基于**模拟路径**和动态规划思想，优化总效用。
- 避免了维度灾难，适用于复杂模型和约束效率问题。

$$\max_{\Theta^C} \mathbb{E}_{\mu(\mathcal{C}^{(k-1)})} \left(\widetilde{U}_{i,T} + \beta^T V_{\mathsf{NN}}(s_{i,T}; \Theta^V) \right)$$



详细求解算法图