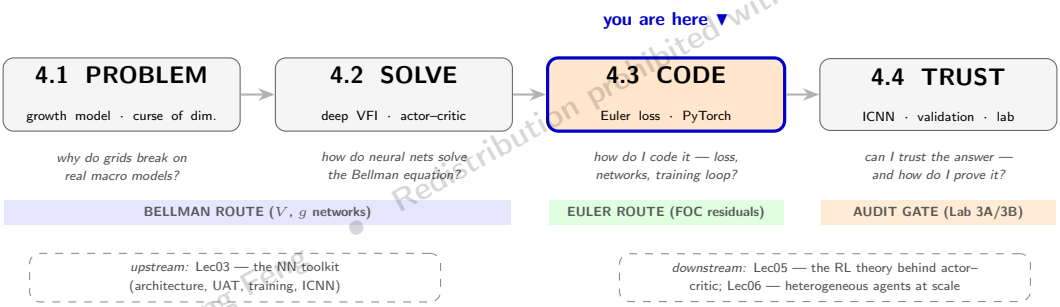
AI for Economic Research: Dynamic Models, Language, and Agents
Lecture 4.3: Euler-Equation Method & PyTorch Implementation

Zhigang Feng

2026-07-07 10:05:49

Lecture 4 in One Map

One lecture, four decks, one running example: solve the optimal growth model with deep learning — pose the problem, solve it two ways, code it, and prove the answer right.



Act 4.3 (here) *solves it the Euler way and writes the code: first-order conditions become the loss, PyTorch's autograd does the calculus, and PolicyNet/ValueNet train to the analytical solution (Lab 3B).*

Euler Equation Method + Supervised Learning

First-order conditions as a loss function: solve by minimizing residuals

Context: Two Approaches to the Same Problem

- We have seen the **actor-critic** approach: train two networks to satisfy the Bellman equation
- Now we consider an alternative: **Euler equation minimization**
  - Derive the first-order conditions (Euler equations)
  - Train a *single* policy network to satisfy them
- **Comparison**:
  - Actor-Critic (Bellman): Two networks, no need to derive FOCs, RL flavor
  - Euler Equation: One network, requires deriving FOCs, supervised learning flavor
- Both are valid; which is better depends on the model:
  - Euler equation method is very precise when FOCs are available
  - Actor-Critic is more general (handles kinks, discrete choices)

Euler-equation training is a **supervised** *problem (regress the FOC residual against zero); actor-critic is* **reinforcement** *(policy gradient on the Bellman objective). Lec05 T3 formalizes the actor-critic loss derivation; here we treat both as PyTorch optimization.*

The Euler Equation: Optimality Condition

- Capital accumulation: $k' = g(k)$, consumption: $c = f(k) + (1-\delta)k - g(k)$
- For CRRA utility $u(c) = \frac{c^{1-\sigma}}{1-\sigma}$ and Cobb-Douglas production, the equilibrium satisfies:

$$u'(c) = \beta\, u'(c')\left[\alpha\, g(k)^{\alpha-1} + 1 - \delta\right] \tag{EE}$$

- **Interpretation**: The marginal cost of saving today equals the discounted marginal benefit tomorrow
- **Goal**: Learn a policy g that makes the **Euler residual** vanish:

$$\mathcal{R}(k; g) = u'(c) - \beta\, u'(c')\left[\alpha\, g(k)^{\alpha-1} + 1 - \delta\right]$$

for all economically relevant k

Euler Equation: Necessary but Not Sufficient

- **Important caveat**: The Euler equation is a *necessary* condition for optimality, but **not sufficient** by itself
- For a complete characterization, we also need:
  - **Transversality condition**: $\lim_{t\to\infty} \beta^t u'(c_t)\, k_{t+1} = 0$
  - **Feasibility constraints**: $c \geq 0$, $k' \geq 0$
- **Handling inequality constraints**:
  - Kuhn-Tucker complementary slackness conditions arise
  - Can be converted to equalities via Fischer-Burmeister functions or penalty methods
  - Or handled via architectural constraints (sigmoid output)
- For now, we focus on **interior solutions** where the Euler equation holds with equality throughout

From Euler Residual to Loss Function

- Approximate the policy by a neural network: $g(k) \approx \Gamma_{\gamma_g}(k)$
- Sample $k_i \sim D(k)$ (e.g., stationary distribution or uniform)
- Define the **mean-squared Euler residual** loss as an expectation over the sampling distribution:

$$\mathcal{L}_g(\gamma_g) = \mathbb{E}_{k\sim D(k)}\left[\mathcal{R}(k;\Gamma_{\gamma_g})^2\right] \approx \frac{1}{N}\sum_{i=1}^{N}\left[\mathcal{R}(k_i;\Gamma_{\gamma_g})\right]^2$$

The empirical mean $\frac{1}{N}\sum_i$ is the special case of $\mathbb{E}_{k\sim D(k)}$ using N i.i.d. draws.

- **This is supervised learning**: the "label" for each state k_i is zero (the Euler equation should hold exactly)
- Optimal policy parameters satisfy: $\nabla_{\gamma_g}\mathcal{L}_g(\gamma_g^*) = 0$
- All derivatives computed automatically via `autograd`

The Mean Value Intuition

- We minimize the *average* Euler residual across sampled states:

$$\mathcal{L}_g(\gamma_g) = \mathbb{E}_{k\sim D(k)}\left[\mathcal{R}(k;\Gamma_{\gamma_g})^2\right] \approx \frac{1}{N}\sum_{i=1}^{N}\mathcal{R}(k_i;\Gamma_{\gamma_g})^2$$

- **Intuition**:
  - If a candidate policy satisfies the Euler equation *on average* across all sampled states, it should be a good approximation to the true equilibrium policy
  - If the average residual is large, the policy must be violating optimality somewhere

- This "average fitness" criterion:
  - Ensures global accuracy across the domain
  - Is computationally tractable via Monte Carlo sampling
  - Naturally integrates with gradient-based NN training

The Stochastic Case: Expectation in the Euler Equation

- With a productivity shock z, the Euler equation carries a **conditional expectation** over next period's shock z':

$$u'(c_t) = \beta\, \mathbb{E}_{z'|z}\big[u'(c_{t+1})(\alpha k_{t+1}^{\alpha-1} z_{t+1} + 1 - \delta)\big]$$

- The residual $\mathcal{R}$ now depends on that inner expectation — there is no closed form.

- **Monte-Carlo estimate**: replace $\mathbb{E}_{z'|z}$ by an average over J shock draws z'_{ij}:

$$\widehat{\mathcal{R}}(k_i; g) = u'(c_i) - \frac{\beta}{J} \sum_{j=1}^{J} u'(c'_{ij}) \big[\alpha k_i'^{\alpha-1} z'_{ij} + 1 - \delta\big]$$

- **Two layers of sampling**: an *outer* average over states $k_i \sim D(k)$ (the loss $\mathbb{E}_{k\sim D(k)}$) and an *inner* average over shocks z'_{ij} (this conditional expectation).

Variance Reduction: Gauss–Hermite Quadrature

- Plain Monte-Carlo over J random shock draws is *noisy*: its error falls only like $1/\sqrt{J}$, and that noise enters every gradient.

- **Gauss–Hermite quadrature** replaces the random draws with a few *deterministic* nodes z'_j and weights w_j:

$$\mathbb{E}_{z'|z}[h(z')] = \int h(z')\,\phi(z')\,dz' \approx \sum_{j=1}^{J} w_j\,h(z'_j),$$

  exact whenever h is a polynomial of degree $\leq 2J-1$ (against the Gaussian weight ϕ).

  - Log-normal shocks $z' = e^{\mu+\sigma\varepsilon}$, $\varepsilon \sim \mathcal{N}(0,1)$: change variables to the standard Gaussian, then read nodes/weights from a table.
  - **Why it works**: smooth integrands are nearly polynomial, so a handful of well-placed nodes ($J \approx 5$–7) match thousands of random draws — same accuracy, *zero* sampling noise.

- **Why the loss stays differentiable**: the nodes and weights are *constants*, so the loss is a fixed weighted sum of smooth network outputs — `autograd` differentiates straight through. Mini-batching over states only changes *which* states enter the sum; it never breaks the gradient (we sample exogenous shocks/states, not policy-dependent randomness, so no reparameterization trick is needed).

Neural Network for the Policy Function

- Two-hidden-layer MLP (64–64–1) with ReLU and Sigmoid:

$$\boxed{\Gamma_{\gamma_g}(k) = \sigma_2\Big(W_2 \cdot \sigma_1(W_1 k + b_1) + b_2\Big)}$$

- Hidden activation $\sigma_1 = \texttt{ReLU}$; output activation $\sigma_2 = \texttt{Sigmoid}$
- Sigmoid output is scaled to $[0, f(k) + (1-\delta)k - \varepsilon]$ to ensure feasibility:

$$0 \le \Gamma_{\gamma_g}(k) \le f(k) + (1-\delta)k$$

- All derivatives $\partial \mathcal{L}_g / \partial \gamma_g$ are computed automatically by `autograd`
- No need to derive or code analytical Jacobians

Feasibility by Construction

- **Why Sigmoid, then rescale?**
  - Sigmoid output $s \in (0, 1)$ ensures $g(k) = s \cdot [f(k) + (1 - \delta)k - \varepsilon]$ stays in the feasible interior
  - Avoids generating samples with $c \leq 0$ during training (which would blow up $u'(c) = c^{-\sigma}$)
- **Why subtract ε?** A small slack prevents marginal-utility divergence near $c = 0$
- **Alternatives**:
  - `Softplus` output: always positive, smooth
  - Hard `torch.clamp`: simple but kills gradients at the boundary
  - Log-parameterization: train on $\log c$ rather than c
- Architectural feasibility outperforms post-hoc clipping — Lab 3B verifies this empirically

Minimising the Euler Residual: Training Loop

- Initialize $\gamma_g^{(0)}$; choose learning rate α_g
- **Repeat until convergence**:
  1. Draw a mini-batch $\{k_i\}_{i=1}^{B}$
  2. Compute loss $\mathcal{L}_g(\gamma_g)$ and its gradient via back-propagation
  3. Gradient step:
$$\gamma_g^{(j+1)} = \gamma_g^{(j)} - \alpha_g \nabla_{\gamma_g} \mathcal{L}_g(\gamma_g^{(j)})$$
  4. Optionally resample k_i from the same distribution $D(k)$
- **Pros of Euler method**:
  - Very precise policy functions; theoretically grounded
  - Only one network to train (simpler than actor-critic)
- **Cons**: Does not yield the value function V; requires deriving FOCs

PyTorch Implementation

PolicyNet, ValueNet, and the training loop — the code behind Lab 3B

Implementation Overview: Two Approaches

- **Approach 1 — Deep VFI (Lab 3B, Part 1)**: `PolicyNet` + `ValueNet`, T-step lookahead with soft constraints; result = $g(k)$ *and* $V(k)$.
- **Approach 2 — Euler Equation (Lab 3B, Part 2)**: Single `PolicyNetwork`; loss = mean-squared Euler residual; result = policy $g(k)$.
- Both validated against the analytical solution from Lab 3A.

Common PyTorch skeleton (both approaches)

1. **Define networks**: `PolicyNet` (Sigmoid output, feasibility-rescaled) and/or `ValueNet`.
2. **Build loss**: Bellman MSE $= (V(k) - (u + \beta V(k')))^2$ *and/or* Euler residual $= u'(c) - \beta\,\mathbb{E}[u'(c')\,(\alpha k'^{\alpha-1} z' + 1 - \delta)]$.
3. **Train**: `AdamW` on mini-batches sampled from $D(k)$; alternate Critic/Actor steps when both nets are present.

Network Architecture: PolicyNet and ValueNet

Code targets PyTorch ≥ 2.3 (2024); AdamW optimizer per Lec03 T3:225. Lab 3A/3B notebooks use matching versions.

```python
class ValueNet(nn.Module):
    def __init__(self, nh=64):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(1, nh), nn.ReLU(),
            nn.Linear(nh, nh), nn.ReLU(),
            nn.Linear(nh, 1))
    def forward(self, k): return self.net(k)

class PolicyNet(nn.Module):
    def __init__(self, alpha, eps, nh=64):
        super().__init__()
        self.alpha, self.eps = alpha, eps
        self.net = nn.Sequential(
            nn.Linear(1, nh), nn.ReLU(),
            nn.Linear(nh, nh), nn.ReLU(),
            nn.Linear(nh, 1), nn.Sigmoid())  # Output in [0, 1]
    def forward(self, k):
        max_kp = torch.clamp(k**self.alpha - self.eps, 1e-9)
        return self.net(k) * max_kp  # Scale to [0, f(k)-eps]
```


- `Sigmoid` ensures $g(k) \in [0, f(k) - \varepsilon]$: feasibility by construction
- `ValueNet`: unconstrained output (value can be any real number)

Policy Evaluation: Bellman MSE

```python
def value_update_step(self):
    self.value_optimizer.zero_grad()
    # Sample batch of capital states
    k_batch = torch.rand(self.n_batch, 1, device=self.device) \
              * (self.k_max - self.k_min) + self.k_min

    v_current = self.value_net(k_batch)            # V(k)
    kp_batch  = self.policy_net(k_batch)            # k' = g(k)
    u_batch   = self.utility(k_batch, kp_batch)     # u(c)
    v_next    = self.value_net(kp_batch).detach()   # V(k') -- detach!
    bellman_rhs = u_batch + self.beta * v_next      # u + beta*V(k')

    loss = nn.functional.mse_loss(v_current, bellman_rhs)
    loss.backward()
    self.value_optimizer.step()
    return loss.item()
```

- Minimizes $\mathbb{E}[(V(k) - (u + \beta V(k')))^2]$
- `detach()`: prevents gradients from flowing through $V(k')$ — treats the target as fixed

Policy Improvement: Gradient Ascent

```python
def policy_step(self):
    self.policy_optimizer.zero_grad()
    # Sample batch of capital states
    k_batch = torch.rand(self.n_batch, 1, device=self.device) \
              * (self.k_max - self.k_min) + self.k_min

    kp_batch = self.policy_net(k_batch)              # k' = g(k)
    u_batch  = self.utility(k_batch, kp_batch)       # u(c)
    v_next   = self.value_net(kp_batch)               # V(k')

    objective = torch.mean(u_batch + self.beta * v_next)

    loss = -objective   # Maximize objective = Minimize -objective
    loss.backward()
    self.policy_optimizer.step()
    return objective.item()
```

- Maximizes $\mathbb{E}[u(k, g(k)) + \beta\, V(g(k))]$
- Gradients flow through **both** u and V into the policy network parameters

The Training Loop: Alternating Updates

```python
def train(self):
    print(f"Training started on device={self.device}")
    pbar = tqdm(range(self.n_epoch), desc="Outer Loop")

    for _ in pbar:
        # Phase 1: Update Value Net (several inner steps)
        for _ in range(self.n_inner_value):
            v_loss = self.value_update_step()
        self.value_losses.append(v_loss)

        # Phase 2: Update Policy Net (several inner steps)
        for _ in range(self.n_inner_policy):
            policy_obj = self.policy_update_step()
        self.policy_objectives.append(policy_obj)

        pbar.set_postfix({
            "ValueLoss": f"{v_loss:.4e}",
            "PolicyObj": f"{policy_obj:.4e}"
        })
```

- Alternates between updating the Critic and the Actor
- Multiple inner steps per outer iteration for stability

Deep Value Function Iteration (DVFI): Key Innovations

Lab 3B, Part 1 introduces three innovations beyond the basic actor-critic:

1. **Unified Objective (J)**: T-step discounted sum plus terminal value:

$$J(y_0) = \sum_{t=0}^{T-1} \beta^t u(c_t) + \beta^T V(y_T)$$

2. T-**Step Lookahead**: Simulate the economy for $T = 30$–50 periods
   - Gradients "see" long-term consequences of policy choices
   - Reduces bias compared to one-step bootstrapping

3. **Soft Constraints** (penalty method):
   - If $c > 0$: $u = \ln(c)$ (standard)
   - If $c \leq 0$: $u = \text{Penalty} \times (c - \varepsilon)$ (pushes back to feasibility)
   - Avoids hard clamping that kills gradients

DVFI: Separated Training Phases

- **Phase A — Policy Net** (maximize J):
  - Sample initial states y_0 uniformly in $[y_{\min}, y_{\max}]$
  - Simulate T periods using PolicyNet
  - Compute $J = \sum \beta^t u(c_t) + \beta^T V(y_T)$
  - Minimize $-J$ (gradient ascent on expected reward)
- **Phase B — Value Net** (fit V to J):
  - Target: J from Phase A, **detached** from policy gradients
  - Loss: $\|V(y_0) - J\|^2$ (standard regression)
- **Key detail**: `retain_graph=True` allows both phases to share the same computation graph; `detach()` in Phase B prevents policy updates through the value loss

Accuracy Benchmarks: Maximum Euler Residual

After training, evaluate the policy on a fine validation grid:

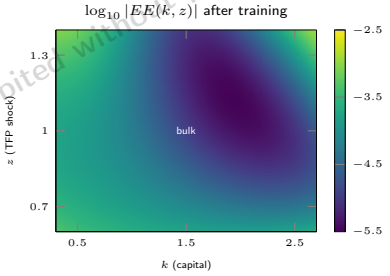
$$\log_{10}\left|EE(k_i, z_i)\right|, \quad i = 1, \ldots, M$$

Standard benchmarks (Judd, 1998)

$\log_{10}|EE| < -3$: acceptable ($< 0.1\%$).
$\log_{10}|EE| < -5$: good ($< 10^{-3}\%$).
$\log_{10}|EE| < -7$: excellent.

Optimal growth (log utility, Cobb–Douglas):

- **Lab 3B Part 1 (DVFI, $T = 30$)**: $\log_{10}|EE| \approx -3.5$ — slightly less precise but yields V too
- **Lab 3B Part 2 (Euler)**: $\log_{10}|EE| \approx -5$ (64-64-1 PolicyNet, 5,000 epochs, AdamW)



The validation gate: dark = low residual (well trained); bright = boundary effects. Same shape Lab 3A/3B produces.

Comparison Against Lab 3A Ground Truth

- Analytical policy for log utility, $\delta = 1$:

$$c^*(y) = (1 - \alpha\beta)\, y, \qquad k'^*(y) = \alpha\beta\, y$$

- Validate by plotting:
  1. Trained policy $g_\theta(k)$ versus $k'^*(y)$ on the same axes
  2. Pointwise error $|g_\theta(k) - k'^*(y)|$ as a function of k
  3. Histogram of $\log_{10} |EE(k_i)|$ across a validation sample

- **Sanity checks for Lab 3B (after 5,000 epochs)**:
  - Max policy error $< 10^{-3}$
  - Savings rate $g(k)/y$ pinned at $\alpha\beta$ across the grid
  - Loss curve monotone-decreasing on log scale, no oscillation

Bridge: It Runs — Now Make It Respect the Theory

Act 4.3 made it concrete: the Euler residual as a training loss, PolicyNet and ValueNet in PyTorch, and the training loop that drives both to the analytical solution.

- You can now code both routes to the same policy: Bellman (actor–critic) and Euler (residual minimization) — with sanity checks against the closed form.
- **Next (4.4):** generic networks can violate concavity and monotonicity exactly where it hurts the solver; ICNN and monotone architectures build the theory in — and the Lab 3A/3B audit decides whether to believe any of it.

Arc: 4.1 PROBLEM (growth model + curse) → 4.2 SOLVE (deep VFI + actor–critic) → **4.3 CODE** *(Euler + PyTorch) → 4.4 TRUST (structure + validation).*