Lecture 9: Parameterization and Neural Network

Yasuyuki Sawada, Yaolang Zhong

University of Tokyo

yaolang.zhong@e.u-tokyo.ac.jp

December 17, 2025

Lecture Overview

- Motivation and introduction to parameterization of value/policy functions
- Classical bases: Polynomials (Chebyshev) and Splines and optimization techniques
- Introduction to Neural Networks
  - Universal Approximation Theorem
  - Automatic Differentiation
  - Training Neural Networks

Motivation: The Limits of Discretization

- Consider a standard Bellman equation for an infinite horizon problem:

$$V(s) = \max_{a \in \pi(s)} \{r(s,a) + \beta \mathbb{E}[V(s')|s,a]\}$$

  The Non-Parameterized / Tabular Representation approach involves discretizing the state space S into a grid $\{s_1, ..., s_N\}$ and storing $V(s_i)$ as a vector in $\mathbb{R}^N$.

- Curse of Dimensionality: Tabular methods require storing values on a grid G_N. For state dimension d and resolution G, memory scales as $N = G^d$.

- Conceptually: Approximates $V \in \mathcal{V}$ by recording values on a discrete grid $G_N \subset S$. The "parameters" θ are the values $V(s_i)$ themselves.

$$\dim(\theta) = |G_N| \propto e^d \quad \text{(Scales with domain resolution)}$$

- Parameterized Representation: Restricts the search to a finite-dimensional manifold $\hat{\mathcal{V}} \subset \mathcal{V}$, indexed by a fixed parameter vector $\theta \in \mathbb{R}^K$, independent of state space cardinality.

Concept: Parameterized Function Approximation

- General Approximation Forms:
  - State Value Function:

$$V(s) \approx \hat{V}(s;\theta) = \sum_{k=1}^{K} \theta_k \phi_k(s)$$

  - State-action Value Function (Q-function):

$$Q(s,a) \approx \hat{Q}(s,a;\xi) = \sum_{j=1}^{M} \xi_j \psi_j(s,a)$$

    Here, inputs (s,a) are concatenated or processed via tensor product bases.

  - Policy Function:

$$\pi(s) \approx \hat{\pi}(s;\eta) = \sum_{l=1}^{L} \eta_l \chi_l(s)$$

- Implication: We replace the functional operator problem $V = TV$ with a non-linear parameter optimization problem in Euclidean space $\mathbb{R}^K$.

Parameterization Bases I: Global Methods - Ordinary Polynomials (1/2)

- Mathematical Definition
  - The approximation is a weighted sum of simple powers of s:

$$\hat{V}(s;\theta) = \theta_0 + \theta_1 s + \theta_2 s^2 + \cdots + \theta_{K-1} s^{K-1}$$

- Property 1: Global Support (The "Ripple Effect")
  - Observation: Every basis function (like s^2 or s^5) is non-zero across the entire domain.
  - Intuition: The parameters are globally coupled. If you adjust θ_2 to fix an error at $s = 0.1$, you unintentionally shift the curve at $s = 0.9$ as well.
  - Contrast: This is unlike "Local" methods (e.g., linear interpolation), where moving one point has no effect on distant regions.

Ordinary Polynomial Basis Functions

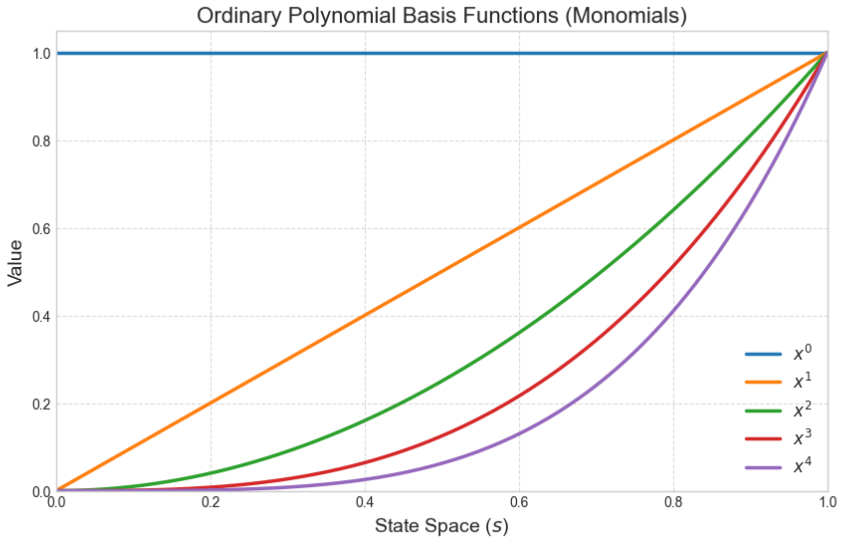


Figure 1: Ordinary polynomial basis functions showing global support

Parameterization Bases I: Global Methods - Ordinary Polynomials (2/2)

- Property 2: The Multicollinearity Problem (Redundancy)
  - Visual Intuition: Plot s^{10} and s^{11} on $[0, 1]$. They look nearly identical: flat near zero, then a sharp spike to 1.
  - The Problem: Because the shapes are so similar, they provide redundant information. The computer struggles to distinguish which parameter (θ_{10} or θ_{11}) accounts for the data.
  - Consequence: The solver becomes confused ("ill-conditioned"). It may output huge positive and negative numbers (e.g., $1,000,000$ and $-999,999$) just to balance them out.
- Property 3: Runge's Phenomenon (Wiggling)
  - The Trap: We often assume "higher degree $=$ better accuracy." For ordinary polynomials on standard grids, the opposite is true.
  - Mechanism: To force a high-degree curve through specific fixed points, the polynomial must take sharp turns.
  - Result: These sharp turns create violent oscillations ("wiggles") between the data points, especially near the edges of the graph. The error explodes as K increases.

The Multicollinearity Problem

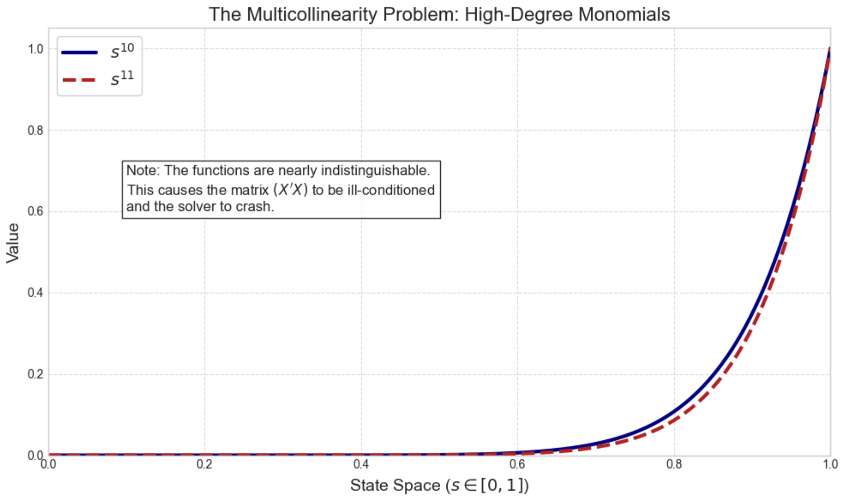


Figure 2: High-degree monomials are nearly indistinguishable, causing ill-conditioned systems

Runge's Phenomenon

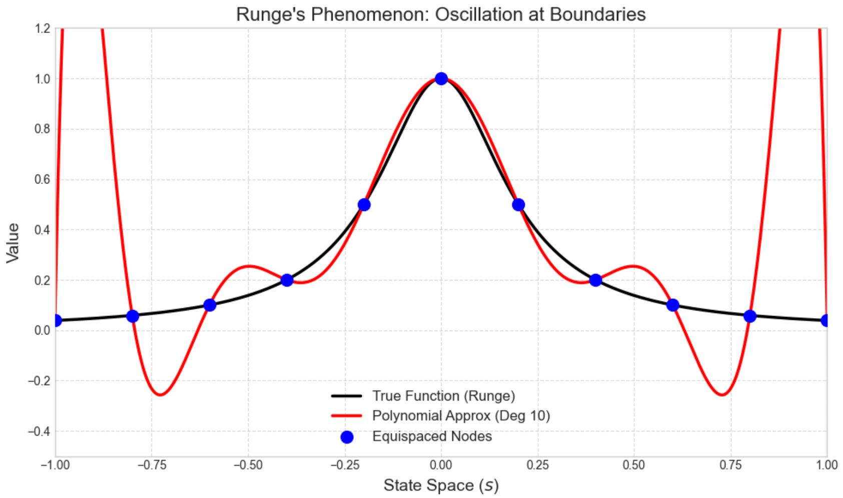


Figure 3: High-degree polynomial interpolation creates violent oscillations

Parameterization Bases I: Global Methods - Chebyshev Polynomials (1/2)

- Mathematical Definition
  - Defined on $[-1, 1]$ via the cosine identity:

$$T_k(x) = \cos(k \arccos(x))$$

  - Since economic variables $s \in [a, b]$ are rarely in $[-1, 1]$, we must use a linear map $\phi(s)$ to transform the domain before approximating:

$$\hat{V}(s; \theta) = \sum_{k=0}^{K-1} \theta_k T_k(\phi(s))$$

- Property 1: Orthogonality (Solving Multicollinearity)
  - Concept: Unlike monomials, Chebyshev polynomials are "orthogonal" (perpendicular) to each other under a specific weighting.
  - Intuition: Each basis function T_k provides completely unique, independent information. T_{10} and T_{11} look nothing alike, so there is no redundancy.
  - Consequence: The solver encounters a perfectly conditioned system (diagonal matrix). We can safely use high degrees ($K = 50+$) without parameters exploding.

Parameterization Bases I: Global Methods - Chebyshev Polynomials (2/2)

- Property 2: Spectral Convergence (Solving Runge's Phenomenon)
  - The "Jackson Theorem": For smooth functions (C^∞), the approximation error drops exponentially as we add parameters.
  - The Trick: We do not use equispaced grids. We sample at "Chebyshev Nodes," which are clustered near the boundaries of the domain.
  - Consequence: This specific spacing neutralizes the violent edge-oscillations (Runge's Phenomenon) seen in ordinary polynomials.
- Limitation: The Gibbs Phenomenon (Handling Kinks)
  - The Trap: Chebyshev methods assume the target function is smooth.
  - Intuition: If the true solution has a "kink" (e.g., a sharp change in consumption due to a binding borrowing constraint), the polynomial cannot turn sharply enough.
  - Result: The approximation "rings" (ripples) across the entire domain just to fit that one sharp corner, degrading accuracy everywhere.

Chebyshev Polynomial Basis Functions

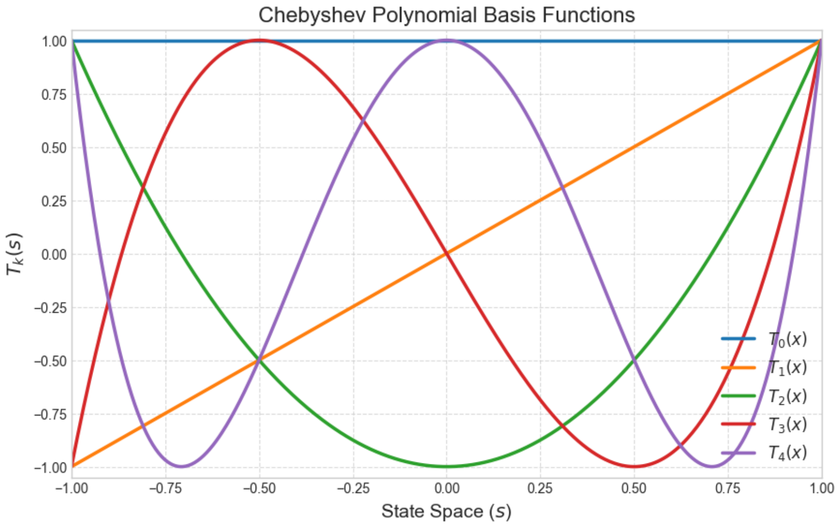


Figure 4: Chebyshev polynomial basis functions showing orthogonality

Parameterization Bases II: Local Methods - B-Splines (1/2)

- Mathematical Definition
  - The domain is divided into sub-intervals using "knots" $\{t_i\}$. The function is approximated by piecing together low-order polynomials:

    $$\hat{V}(s;\theta) = \sum_{i=1}^{N} \theta_i B_{i,p}(s)$$

    where $B_{i,p}(s)$ is a "bell-shaped" basis function that is zero everywhere except for a narrow range around t_i.
- Property 1: Local Support (The "Tent" Intuition)
  - Visual Intuition: Imagine a series of tents pitched side-by-side. If you raise the pole of one tent (θ_i), it only changes the height of that specific tent. It has zero effect on tents far away.
  - Contrast: This is the exact opposite of the "Ripple Effect" in global polynomials.
  - Computational Gain: The matrix we solve is "sparse" (filled with zeros). Computers can solve these systems in linear time $O(N)$, making them extremely fast for large grids.

Parameterization Bases II: Local Methods - B-Splines (2/2)

- Property 2: Flexibility (Handling Kinks)
  - The Feature: Splines allow us to control continuity. By stacking multiple knots at the same point, we can allow the function to have a sharp corner (continuous but not differentiable).
  - Economic Relevance: This is crucial for models with binding constraints (e.g., $\textit{Investment} \geqslant 0$).
  - Consequence: We can capture a sharp kink exactly at the constraint without causing the rest of the function to oscillate (No Gibbs Phenomenon).
- Limitation: The Curse of Dimensionality
  - The Bottleneck: To use splines in multi-dimensional models ($d > 1$), we usually multiply 1D bases together (Tensor Product).
  - The Math: If we need 10 knots per dimension, a 2D model needs $10^2 = 100$ parameters. A 6D model needs $10^6 = 1,000,000$.
  - Consequence: Splines become computationally impossible for medium-to-high dimensional problems ($d \geqslant 4$), paving the way for Neural Networks.

Cubic B-Spline Basis Functions

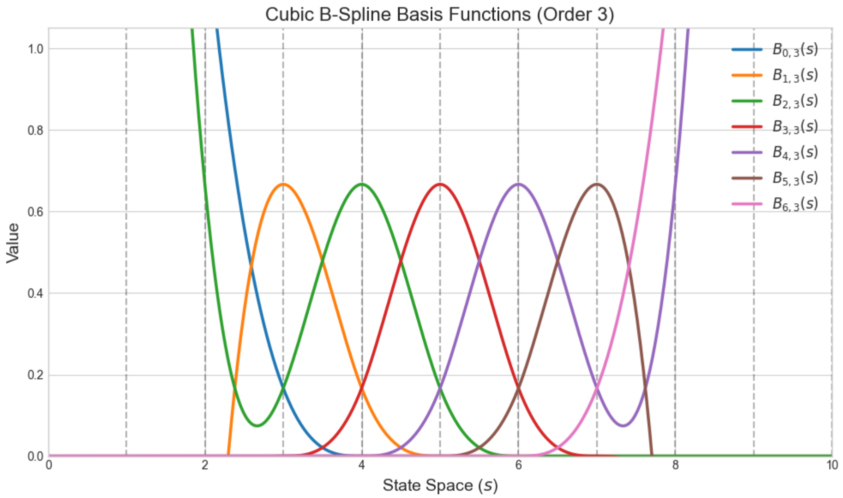


Figure 5: Cubic B-spline basis functions showing local support

B-Splines: Handling Kinks

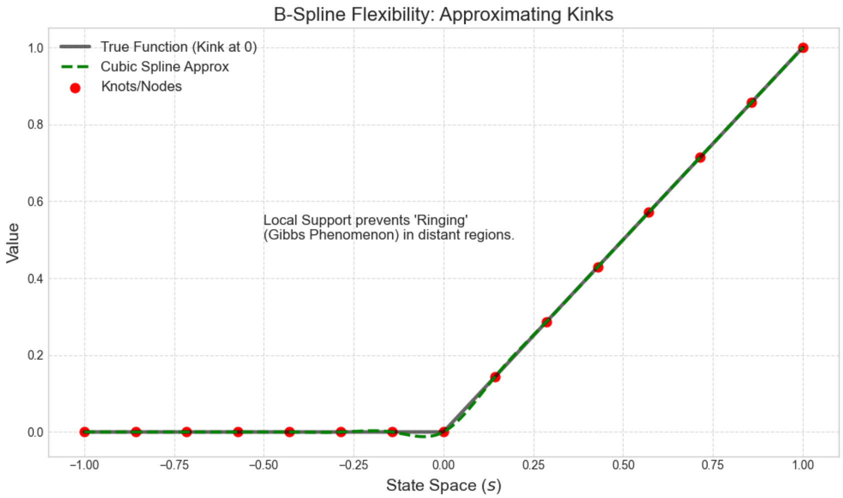


Figure 6: B-splines can capture sharp kinks without global oscillations

Step 1: Construct the Residual Function (1/2)

- Goal: Find a parameter vector $\theta \in \mathbb{R}^K$ such that the approximation $\hat{V}(s;\theta)$ or $\hat{\pi}(s;\theta)$ satisfies the functional equation.
- Since it is an approximation, we define the approximation error on a particular state s as the Residual Function $R(s;\theta)$.
- Case A: Bellman Optimality (Value Function Iteration)
  - The residual captures the error in the non-linear max-operator fixed point:

$$R(s;\theta) = \hat{V}(s;\theta) - \max_{a \in \Gamma(s)} \left\{ r(s,a) + \beta \mathbb{E}[\hat{V}(s';\theta)|s,a] \right\}$$

  - This creates a highly non-linear dependency on θ because the optimal policy $\pi^*(s)$ changes implicitly as θ changes.
- Case B: Policy Evaluation (Fixed Policy π)
  - We seek the value of a specific, fixed policy rule $a = \pi(s)$.

$$R(s;\theta) = \hat{V}(s;\theta) - \left(r(s,\pi(s)) + \beta \mathbb{E}[\hat{V}(s';\theta)|s,\pi(s)] \right)$$

  - If $\hat{V}$ is linear in θ (e.g., polynomials), this residual becomes linear in θ.

Step 1: Construct the Residual Function (2/2)

- Case C: Euler Equation Error (First-Order Condition)
  - Instead of the Bellman equation, we can directly target the optimality condition (Euler equation):
    $$u'(c_t) = \beta \mathbb{E}_t \left[u'(c_{t+1})(1 + r_{t+1}) \right]$$
  - Given a policy $c = \hat{\pi}(s; \theta)$, the Euler residual is:
    $$R(s; \theta) = u'(\hat{\pi}(s; \theta)) - \beta \mathbb{E} \left[u'(\hat{\pi}(s'; \theta))(1 + r(s, s')) \mid s \right]$$
- Note that in practice we often focus on the unit-free error:
  $$\varepsilon(s) = \left| \frac{R(s; \theta)}{\hat{V}(s; \theta)} \right| \quad \text{or} \quad \left| \frac{\beta \mathbb{E} \left[u'(c')(1 + r') \right]}{u'(c)} - 1 \right|$$
- Interpretation:
  - $\varepsilon(s) = 10^{-3}$: The agent makes a 0.1% mistake in consumption allocation.
  - $\varepsilon(s) = 10^{-6}$: The solution is accurate to 6 decimal places.
  - Standard target: $\max_s \varepsilon(s) < 10^{-4}$ (Judd, 1998).

*Derivation: Linearity of the Residual in Policy Evaluation (1/2)

Claim: If $\hat{V}(s;\theta)$ is linear in θ, then $R(s;\theta)$ is linear in θ.

- Let the approximation be linear in parameters:

$$\hat{V}(s;\theta) = \sum_{k=1}^{K} \theta_k \phi_k(s)$$

- Substitute into Residual Definition

$$R(s;\theta) = \underbrace{\sum_{k=1}^{K} \theta_k \phi_k(s)}_{\hat{V}(s)} - \left[r(s) + \beta \mathbb{E}_{s'} \left[\underbrace{\sum_{k=1}^{K} \theta_k \phi_k(s')}_{\hat{V}(s')} \right] \right]$$

*Derivation: Linearity of the Residual in Policy Evaluation (2/2)

- Since expectation is a linear operator, we can pull the constants θ_k out:

$$\mathbb{E}_{s'}\left[\sum_{k=1}^{K} \theta_k \phi_k(s')\right] = \sum_{k=1}^{K} \theta_k \mathbb{E}_{s'}[\phi_k(s')]$$

- Grouping Terms by θ:

$$R(s;\theta) = \sum_{k=1}^{K} \theta_k \underbrace{\left(\phi_k(s) - \beta\mathbb{E}_{s'}[\phi_k(s')]\right)}_{\text{Effective Basis } \tilde{\phi}_k(s)} - r(s)$$

- Matrix Form:

$$R(s;\theta) = \tilde{\Phi}(s)\theta - z(s)$$

where $\tilde{\Phi}(s)$ is the row vector of bases adjusted for discounting, and $z(s) = r(s)$.

Step 2: Construct the Objective Function (1/3)

- The residual function is the approximation error in each state s. When the state space S is continuous, we cannot enforce $R(s;\theta) = 0$ everywhere.
- We aggregate the errors by projecting them against a set of M test functions.
- General Framework (Weighted Residual):

$$\text{Minimize or satisfy: } \int_S R(s;\theta)\psi_j(s)w(s)ds = 0, \quad j = 1, \ldots, M$$

  where:
  - $\psi_j(s)$: Test functions (features of the state space).
  - $w(s)$: Weighting function (e.g., probability distribution).
  - M: Number of conditions. We need at least as many conditions as parameters ($M \geqslant K$) to solve for θ.
- Two Common Approaches:
  1. Collocation Method (Uses Dirac deltas; usually $M \geqslant K$).
  2. Galerkin Method (Uses Basis functions; usually $M = K$).

Step 2: Construct the Objective Function (2/3)

Method A: Collocation (Point Evaluation)

- Choose M grid points $\{s_1, \ldots, s_M\}$.
- Define the Residual Vector $\mathbf{R}(\theta)$ by stacking the errors at each point:

$$\mathbf{R}(\theta) = \begin{bmatrix} R(s_1;\theta) \\ \vdots \\ R(s_M;\theta) \end{bmatrix}$$

- Case 1: Exact-Identification ($M = K$): We have exactly as many equations (M) as parameters (K). Objective: Find the root of the vector system:

$$\mathbf{R}(\theta) = \mathbf{0}$$

- Case 2: Over-Identification ($M > K$): We have more data points than parameters. The system $\mathbf{R}(\theta) = \mathbf{0}$ has no solution. Objective: Minimize the "size" of the residual vector (Mean Squared Error):

$$\min_{\theta} \mathcal{L}(\theta) = \frac{1}{M}\|\mathbf{R}(\theta)\|^2 = \frac{1}{M}\sum_{i=1}^{M} R(s_i;\theta)^2$$

Step 2: Construct the Objective Function (3/3)

- Method B: Galerkin (Orthogonal Projection)
  - Definitions: Set test functions equal to the basis functions $\psi_j(s) = \phi_j(s)$, for example, the Chebyshev polynomials.
  - This implies $M = K$ (Exact-Identification): We generate exactly one integral condition per parameter.
  - Objective: Force the residual to be orthogonal to every basis function:

$$\int_S R(s;\theta)\phi_j(s)w(s)ds = 0, \quad j = 1, \ldots, K$$

  - Implementation: The integral must be computed numerically (e.g., Gaussian Quadrature) at every optimization step.
  - Trade-off: Much more computationally expensive than Collocation, but ensures the error is minimized globally rather than just at specific points.

Step 3: Choose the Solver Algorithm (1/3)

- We now have a numerical problem defined by our Step 2 choice:
  - Exact-Identification ($M = K$): We must find the root of a square system $\mathbf{R}(\theta) = \mathbf{0}$.
  - Over-Identification ($M > K$): We must find the minimum of a scalar loss function $\min \mathcal{L}(\theta)$.
- The Algorithm Selection Matrix:

Criteria	Algorithm A: Newton-Raphson	Algorithm B: Gradient Descent
Problem Type	Root Finding ($M = K$)	Minimization ($M > K$)
Complexity	High (Matrix Inversion $O(K^3)$)	Low (Linear $O(K)$)
Convergence	Quadratic (Very Fast)	Linear (Slow)
Use Case	High-precision, small models	Neural Networks, large models

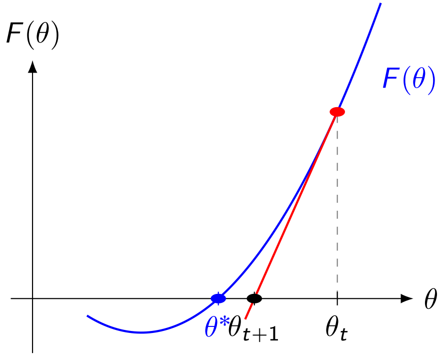
Step 3: Newton-Raphson (Root Finding)

- General Goal: Find the root θ such that $\mathbf{F}(\theta) = \mathbf{0}$.
- Defining $\mathbf{F}$ by Case:
  - Case 1 ($M = K$): Solve $\mathbf{R}(\theta) = \mathbf{0}$. Here $\mathbf{F} = \mathbf{R}$. The derivative $\mathbf{F}'$ is the Jacobian matrix $\mathbf{J}_R$.
  - Case 2 ($M > K$): Minimize $\mathcal{L}(\theta)$. We solve for zero gradient: $\mathbf{F} = \nabla\mathcal{L}$. The derivative $\mathbf{F}'$ is the Hessian matrix $\mathbf{H}$.
- Linearization (1st Order Taylor):

$$\mathbf{F}(\theta) \approx \mathbf{F}(\theta_t) + \mathbf{F}'(\theta_t)(\theta - \theta_t) = \mathbf{0}$$

- Newton Update Rule:

$$\theta_{t+1} = \theta_t - [\mathbf{F}'(\theta_t)]^{-1}\mathbf{F}(\theta_t)$$



Step 3: Gradient Descent (Minimization)

- Context: Used for Over-Identified systems ($M > K$) to minimize the Loss $\mathcal{L}(\theta)$, or when K is too large for Newton.
- Relation to Newton's Method (Optimization):
  - To minimize $\mathcal{L}$, Newton's update would use the *Hessian* (Matrix of 2nd derivatives $\mathbf{H}$):
    $$\theta_{t+1} = \theta_t - \mathbf{H}(\theta_t)^{-1} \nabla \mathcal{L}(\theta_t)$$
  - Gradient Descent simplifies this by approximating the inverse Hessian with a scalar identity matrix $\alpha\mathbf{I}$ (where α is step size):
    $$\mathbf{H}(\theta_t)^{-1} \approx \alpha \mathbf{I} \implies \theta_{t+1} = \theta_t - \alpha \nabla \mathcal{L}(\theta_t)$$
- Implication:
  - Newton jumps directly to the minimum of the local quadratic approximation (uses curvature info in $\mathbf{H}$).
  - Gradient Descent takes small steps based on local slope only (ignores curvature).

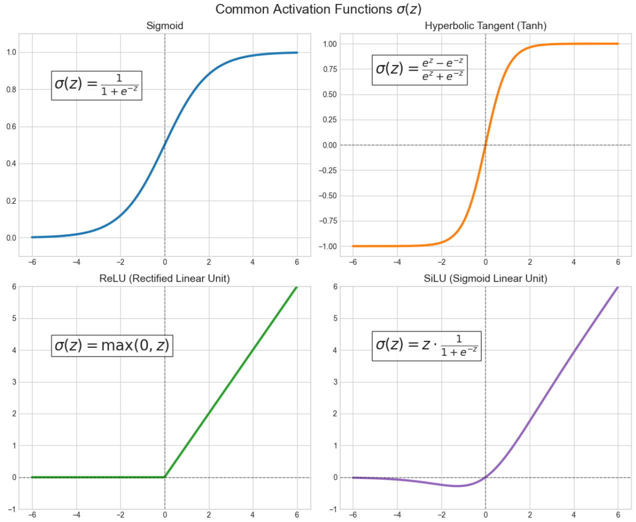
Introduction to Neural Networks

- A Neural Network (NN) is a specific type of non-linear parameterization $\hat{V}(s;\theta)$. Unlike linear bases where $\hat{V} = \sum \theta_i \phi_i(s)$, NNs nest parameters inside non-linear activation functions.
- Specifically, a feedforward neural network (FFN, or Multilayer Perceptron (MLP)) has each of its layer ℓ:

$$x_\ell = \sigma_\ell(W_\ell x_{\ell-1} + b_\ell) \quad \text{for } \ell = 1, \dots, L$$

  - Parameters $\theta = \{W_\ell, b_\ell\}_{\ell=1}^{L}$. Weights W and biases b.
  - Activation Function $\sigma(\cdot)$: Non-linear function applied element-wise.
  - Common choices for σ: ReLU ($\max(0, z)$), Tanh, Sigmoid, SiLU.

Visualization of Activation Functions



Interactive visualization: `https://playground.tensorflow.org/`

Universal Approximation Theorem

Theorem (Hornik et al., 1989; Cybenko, 1989)

Let $f(s)$ be any continuous function on a compact domain S. For any error tolerance $\epsilon > 0$, there exists a Neural Network $\hat{V}(s;\theta)$ with a single hidden layer and finite width N such that:

$$\sup_{s \in S} |f(s) - \hat{V}(s;\theta)| < \epsilon$$

- Implications:
  - Universality (Density): This property is mathematically known as being "dense" in the function space. It simply means that Neural Networks are not restricted to specific shapes. Unlike linear models (planes) or low-order polynomials (smooth curves), an NN can approximate any valid economic function to any desired degree of accuracy.
  - Existence vs Construction: The theorem guarantees that a solution (the optimal θ) exists, but it does not tell us how to find it. Finding these parameters requires numerical optimization (e.g., Stochastic Gradient Descent).

NN vs conventional approximators

- Adaptive Basis vs Fixed Basis
  - Polynomials (Fixed): You fix the shapes (s, s^2, s^3) ex-ante. To fit a kink or a specific curve, you often need hundreds of global terms.
  - Neural Nets (Adaptive): The basis functions $\sigma(ws + b)$ contain parameters (w, b) inside the function. The network learns to shift and rotate the basis to put the "complexity" exactly where the function changes, and leaves the rest flat.
- Beating the Curse of Dimensionality (Barron's Theorem)
  - Conventional Methods: The error decays as $O(N^{-1/d})$. In high dimensions (d), you need exponentially more parameters to reduce error (filling the volume).
  - Neural Networks: The error decays as $O(N^{-1/2})$. The rate is independent of the dimension d.
  - Intuition: NNs act like a Monte Carlo simulation. Instead of filling the entire grid, they sample the directions where the function varies the most. The error depends on the complexity of the target function, not the volume of the input space.

Training NNs: Optimization via General Framework

Training a Neural Network is just a specific instance of our general numerical framework.

- Step 1 (Residual Function): Define the error at a single state s.
  - The residual is the Bellman error: $R(s;\theta) = \hat{V}(s;\theta) - \mathcal{T}\hat{V}(s;\theta)$.
  - Here $\hat{V}(s;\theta)$ is the Neural Network output.
- Step 2 (Objective Function): Aggregate residuals (Method C: Least Squares).
  - Since we are in the over-identified case ($M \gg K$), we minimize the Mean Squared Error over a set of sampled states:

$$\min_{\theta} \mathcal{L}(\theta) = \frac{1}{M} \sum_{i=1}^{M} \|R(s_i;\theta)\|^2$$

  - The sampling distribution $\mu(s)$ implicitly acts as the weighting function $w(s)$.
- Step 3 (Solver Algorithm): Gradient Descent.
  - We cannot use Newton-Raphson because the Jacobian is too large ($K \sim 10^5 +$).
  - Instead, we use First-Order methods: $\theta_{t+1} = \theta_t - \alpha \nabla_\theta \mathcal{L}(\theta)$.

Training NNs: Automatic Differentiation

- A Neural Network is a nested chain of non-linear functions:

$$\hat{V}(s) = \sigma(W_L \dots \sigma(W_1 s + b_1) \dots)$$

  Deriving gradients $\frac{\partial \mathcal{L}}{\partial W_1}$ by hand (symbolic differentiation) is error-prone and impossible for deep networks.
- Automatic Differentiation (Auto-diff):
  - Modern frameworks (PyTorch, TensorFlow) build a "Computational Graph" as you calculate the residual $R(s; \theta)$.
  - They apply the Chain Rule automatically from the output back to the inputs (Backpropagation).
  - Result: We get exact gradients $\nabla_\theta \mathcal{L}$ with computational cost proportional to just one forward pass.
- Simple Example: $y = (w \cdot x)^2$
  - Forward (Graph Construction): Input $w, x \xrightarrow{z = w \cdot x}$ Node $z \xrightarrow{y = z^2}$ Output y
  - Backward (Gradient Calculation): $\frac{\partial y}{\partial w} = \underbrace{\frac{\partial y}{\partial z}}_{2z} \cdot \underbrace{\frac{\partial z}{\partial w}}_{x} = 2(wx) \cdot x$

Training NNs: Stochastic Gradient Descent (SGD)

With Auto-diff handling the gradient computation, we use SGD to handle the large data scale.

- The Problem with Full Gradient Descent:
  - Computing $\nabla_\theta \mathcal{L}$ requires summing errors over all M states (e.g., $M = 10^6$). This is too slow per step.
- The SGD Solution (Mini-Batch):
  - Sample a small batch $B \subset \{s_1, \ldots, s_M\}$ (e.g., size 64).
  - Estimate the gradient using only this batch:

$$\mathbf{g}_{batch} = \nabla_\theta \frac{1}{|B|} \sum_{s \in B} \|R(s;\theta)\|^2$$

  - Update: $\theta \leftarrow \theta - \alpha \cdot \mathbf{g}_{batch}$
- Summary of the Training Loop:
  1. Sample batch B.
  2. Forward Pass: Compute Residuals $R(s)$.
  3. Backward Pass (Auto-diff): Compute Gradients $\nabla_\theta \mathcal{L}$.
  4. Optimizer Step: Update θ using SGD.