Neural Networks for Differential Equations

Lecture Notes for Computational Methods in Macroeconomics
(4-Week Module)

Keio

2025 Autumn

Contents

1 Week 1: From Brains to Perceptrons - The Origins of AI

1.1 Course Overview and Motivation

This 4-week module introduces neural networks as a powerful computational tool for solving differential equations—a fundamental challenge in modern macroeconomics. While our focus is on the mathematical and computational techniques, these methods will be essential for the advanced macroeconomic models you'll encounter next: optimal growth, investment with frictions, and sovereign default.

1.1.1 Why This Matters for Economics

In macroeconomics, we constantly solve differential equations:

- **Bellman equations** in dynamic programming
- **Euler equations** for optimal consumption and investment
- **Asset pricing equations** (Black-Scholes and extensions)
- **Hamilton-Jacobi-Bellman equations** in continuous time

Traditional numerical methods (value function iteration, projection methods) struggle with:

- High-dimensional state spaces (heterogeneous agents)
- Occasionally binding constraints (borrowing limits, ZLB)
- Non-smooth value functions (default decisions)
- Continuous distributions as state variables

Neural networks offer a solution that scales to these complex problems.

1.2 Two Paths to Artificial Intelligence

The field of AI began with two competing philosophies in the 1950s:

1.2.1 Logic-Based AI (The Symbolic Approach)

The 1956 Dartmouth Conference, led by John McCarthy, Marvin Minsky, and Claude Shannon, viewed intelligence as symbol manipulation and logical reasoning. This approach dominated early AI and led to expert systems and rule-based programs.

Economic parallel: This is like assuming agents follow fixed decision rules or heuristics, rather than optimizing.

1.2.2 Brain-Inspired AI (The Connectionist Approach)

Psychologist Frank Rosenblatt believed intelligence emerges from learning in neural networks. Rather than programming rules, systems should learn from experience.

Economic parallel: This is like agents learning optimal policies through repeated interaction with the economy, similar to adaptive learning in macroeconomics.

1.3 The Biological Inspiration: 神経細胞とその結合

The human brain contains approximately 86 billion neurons, each connected to thousands of others. As described by 甘利俊一 (Shun-ichi Amari) in 1978:

> "A neuron receives signals through dendrites (樹状突起), processes them in the cell body (細胞体), and if the combined signal exceeds a threshold, sends output through the axon (軸索)."

This biological process inspired the mathematical models we use today.

1.4 Early Mathematical Models

1.4.1 McCulloch-Pitts Model (1943)

The first mathematical neuron model:

Definition 1.1 (McCulloch-Pitts Neuron)**.** Given binary inputs $x_1, \ldots, x_n \in \{0, 1\}$ and threshold θ:

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} x_i \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

Limitation: No learning—weights must be set manually.

1.4.2 Rosenblatt's Perceptron (1957)

Revolutionary addition of learning capability:

Definition 1.2 (Perceptron)**.**

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

where w_i are learnable weights, b is learnable bias, and f is an activation function.

Learning rule:

- If output correct: no change
- If output = 0, should be 1: increase weights
- If output = 1, should be 0: decrease weights

Convergence theorem: For linearly separable data, the perceptron finds a solution in finite time.

1.5 The XOR Problem and the First AI Winter

1.5.1 Minsky and Papert's Critique (1969)

The book "Perceptrons" proved that single-layer networks cannot solve non-linearly separable problems.

Example 1.1 (XOR Problem)**.**

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

No single line can separate the 1s from the 0s.

Theorem 1.1 (XOR Impossibility)**.** *No single-layer perceptron can compute XOR.*

Sketch. Assume weights w_1, w_2 and bias b exist. From the truth table:

- $(0,0) \to 0$: requires $b < 0$
- $(0,1) \to 1$: requires $w_2 + b \geq 0$
- $(1,0) \to 1$: requires $w_1 + b \geq 0$
- $(1,1) \to 0$: requires $w_1 + w_2 + b < 0$

These constraints are contradictory. □

Impact: Research funding disappeared, leading to the "AI Winter" (1969-1980s).

1.5.2 The Multi-Layer Solution

A 2-layer network CAN solve XOR:

- Hidden neuron 1: $a_1 = \text{step}(x_1 + x_2 - 0.5)$ (OR-like)
- Hidden neuron 2: $a_2 = \text{step}(-x_1 - x_2 + 1.5)$ (NAND-like)
- Output: $y = \text{step}(a_1 + a_2 - 1.5)$ (combines both)

The problem: How to train the hidden layer? This remained unsolved until...

1.6 Japanese Contributions: 甘利俊一 (Amari Shun-ichi)

1.6.1 The Forgotten Pioneer (1960s)

While Western researchers struggled, Amari was developing the mathematical foundations:

- **1966-1967**: Introduced stochastic gradient descent for multi-layer networks
- **1968**: Published『学習識別の理論』(Theory of Learning and Pattern Recognition)
- Developed convergence proofs and optimal learning rates
- Founded information geometry

Amari's learning rule (predating Western "discovery" by 20 years):

$$\Delta\theta = -\eta \frac{\partial \ell(x, \theta)}{\partial \theta}$$

1.6.2 The Drunk Walk Analogy

Amari's beautiful intuition for stochastic gradient descent:

> "Like a drunk person on a hillside who staggers randomly but more often downhill than uphill, eventually reaching the valley."

This captures the essence: random steps that on average descend toward the optimum.

1.7 The Revival: Backpropagation and Deep Learning

- **1986**: Rumelhart, Hinton, Williams popularize backpropagation
- **2012**: AlexNet wins ImageNet (deep learning explosion)
- **Today**: Neural networks solve complex economic models

2 Week 2: The Learning Engine

2.1 The Learning Framework

Neural network training is fundamentally an optimization problem—finding parameters that minimize a loss function. This parallels many economic problems: firms minimize costs, consumers maximize utility, and policymakers minimize loss functions.

2.1.1 Three Steps of Learning

1. **Define Loss Function**: Quantify how "wrong" the predictions are
2. **Calculate Gradients**: Find the direction to improve
3. **Update Parameters**: Take a step in that direction

This iterative process continues until convergence—similar to finding equilibrium in economic models.

2.2 Loss Functions

2.2.1 Mean Squared Error (Regression)

For continuous outputs (e.g., value functions, prices):

$$L = \frac{1}{2N} \sum_{i=1}^{N} (y_i^{\text{pred}} - y_i^{\text{true}})^2$$

Economic example: Fitting a consumption function to data.

2.2.2 Cross-Entropy (Classification)

For discrete choices (e.g., default/no-default, work/leisure):

$$L = -\sum_{i=1}^{N} \sum_{c=1}^{C} y_{ic} \log(\hat{p}_{ic})$$

Economic example: Predicting firm bankruptcy or sovereign default.

2.3 Gradient Descent: The Optimization Workhorse

2.3.1 The Algorithm

Key Concept

Gradient Descent Update Rule:

$$w_{t+1} = w_t - \eta \nabla L(w_t)$$

where:

- w_t: current parameters
- ∇L: gradient of loss function
- η: learning rate (step size)

2.3.2 The Mountain Descent Analogy

Imagine finding the lowest point in a valley while in fog:

- You can't see the bottom (global view)
- You can feel the slope at your feet (local gradient)
- Strategy: step downhill repeatedly
- Challenge: choosing step size (learning rate)

2.3.3 Learning Rate Dynamics

- **Too small** ($\eta < 0.001$): Extremely slow convergence
- **Too large** ($\eta > 1$): Overshooting, oscillations, divergence
- **Just right** ($\eta \approx 0.01 - 0.1$): Steady convergence

Economic parallel: Like adjustment speed in rational expectations models—too fast causes instability, too slow delays equilibrium.

2.4 Variants of Gradient Descent

2.4.1 Batch Gradient Descent

- Uses entire dataset
- Accurate but slow
- Like solving for general equilibrium with all agents simultaneously

2.4.2 Stochastic Gradient Descent (SGD)

- Uses single data point
- Fast but noisy
- Like adaptive learning where agents update beliefs with each observation

2.4.3 Mini-Batch (Modern Standard)

- Uses small batches (32-256 samples)
- Balances speed and stability
- Exploits GPU parallelization

2.5 Backpropagation: Computing Gradients Efficiently

2.5.1 The Challenge

Networks have millions of parameters. Computing gradients naively would require:

- Numerical differentiation: $\frac{\partial L}{\partial w} \approx \frac{L(w+\epsilon)-L(w-\epsilon)}{2\epsilon}$
- Cost: Two forward passes per parameter (intractable!)

2.5.2 The Solution: Chain Rule

Backpropagation efficiently computes all gradients using the chain rule of calculus.

Example 2.1 (Two-Layer Network)**.** Forward pass:

$$h = \sigma(W_1 x + b_1) \quad \text{(hidden layer)} \tag{1}$$
$$y = W_2 h + b_2 \quad \text{(output)} \tag{2}$$
$$L = \frac{1}{2}(y - t)^2 \quad \text{(loss)} \tag{3}$$

Backward pass (chain rule):

$$\frac{\partial L}{\partial y} = y - t \tag{4}$$
$$\frac{\partial L}{\partial W_2} = (y - t) \cdot h^T \tag{5}$$
$$\frac{\partial L}{\partial h} = W_2^T \cdot (y - t) \tag{6}$$
$$\frac{\partial L}{\partial W_1} = \left(W_2^T (y - t) \odot \sigma'(W_1 x + b_1)\right) x^T \tag{7}$$

Key insight: Reuse intermediate computations (like dynamic programming!).

2.5.3 Error Signal Propagation

The "error signal" flows backward through the network:

- Output layer: Direct error $(y - t)$
- Hidden layers: Weighted sum of next layer's errors
- Each layer: Error $\times$ local gradient

Economic analogy: Like how price signals propagate through supply chains.

2.6 Historical Note: Amari's Contribution

Amari's 1967 stochastic gradient descent preceded Western rediscovery:

Economic Application

Amari viewed learning as a stochastic approximation problem—similar to how economists model learning in markets with incomplete information.

3 Week 3: Building an ODE Solver

3.1 Motivation: Why Neural Networks for Differential Equations?

3.1.1 Traditional Methods vs. Neural Networks

Aspect	Traditional	Neural Network
Domain	Discrete grid	Continuous
Solution	At grid points only	Everywhere
Derivatives	Finite differences	Automatic differentiation
High dimensions	Curse of dimensionality	Scales well
Constraints	Requires special treatment	Natural handling

3.1.2 Economic Applications

Differential equations appear throughout economics:

- Capital accumulation: $\dot{k} = i - \delta k$
- Consumption smoothing: Euler equations
- Asset pricing: Black-Scholes and extensions
- Optimal control: Hamilton-Jacobi-Bellman

3.2 The Model Problem

We solve a first-order ODE that appears in many economic contexts:

Key Concept

ODE to Solve:

$$\frac{d\Psi}{dx} = e^{-2x} - 2\Psi \tag{8}$$

$$\Psi(0) = 0.1 \tag{9}$$

$$x \in [0, 2] \tag{10}$$

Economic interpretations:

- Adjustment cost models with time-varying productivity
- Capital dynamics with depreciation
- Learning models with forgetting

Analytical solution (for verification):

$$\Psi(x) = e^{-2x}(x + 0.1)$$

3.3 Key Innovation: The Trial Solution

3.3.1 The Problem with Direct Approximation

If we approximate $\Psi(x) \approx N(x; \theta)$ directly, the network must learn the initial condition, which can be inaccurate.

3.3.2 The Trial Solution Method

Definition 3.1 (Trial Solution)**.** For initial condition $\Psi(0) = A$:

$$\Psi_t(x; \theta) = A + x \cdot N(x; \theta)$$

where $N(x; \theta)$ is a neural network.

Why this works:

- At $x = 0$: $\Psi_t(0) = A + 0 \cdot N(0) = A$ ✓
- Initial condition satisfied *exactly* for any θ
- Network only needs to learn the deviation from initial value

3.3.3 Computing Derivatives

Using the chain rule:

$$\frac{d\Psi_t}{dx} = N(x;\theta) + x\frac{\partial N}{\partial x}$$

Modern automatic differentiation computes this exactly—no numerical errors!

3.4 Physics-Informed Loss Function

Instead of requiring solution data, we train the network to satisfy the differential equation itself.

Definition 3.2 (Physics-Informed Loss)**.**

$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left[\left.\frac{d\Psi_t}{dx}\right|_{x_i} - f(x_i, \Psi_t(x_i))\right]^2$$

where $f(x, \Psi) = e^{-2x} - 2\Psi$ is the ODE right-hand side.

The network learns to make the ODE residual zero everywhere.

3.5 Network Architecture

For this simple ODE:

- **Input**: $x \in \mathbb{R}$
- **Hidden layer**: 5 neurons, sigmoid activation
- **Output**: $N(x) \in \mathbb{R}$, no activation
- **Parameters**: 15 total (5 weights + 5 biases + 5 output weights)

Why so small? The ODE is smooth and simple. Complex economic models need larger networks.

3.6 Implementation in MATLAB

3.6.1 Main Training Loop

Listing 1: Core Training Loop

```
for iter = 1:max_iterations
    % Forward pass
    [Psi, dPsi] = forward_pass(x_train, v, theta, w, A);

    % Compute ODE residual (physics-informed loss)
    target = exp(-2*x_train) - 2*Psi;  % RHS of ODE
    residual = dPsi - target;          % Should be zero
    loss = mean(residual.^2);

    % Check convergence
    if loss < tolerance
        break;
    end

    % Compute gradients
    [grad_v, grad_theta, grad_w] = compute_gradients(...);

    % Update parameters
```

```
    v = v - learning_rate * grad_v;
    theta = theta - learning_rate * grad_theta;
    w = w - learning_rate * grad_w;
end
```

3.6.2 Forward Pass

Listing 2: Network Forward Pass

```
function [Psi, dPsi] = forward_pass(x, v, theta, w, A)
    % Hidden layer
    z = v .* x + theta;        % Linear combination
    y = sigmoid(z);            % Activation

    % Network output
    Nx = y' * w;               % N(x)

    % Derivative (via chain rule)
    sigmoid_deriv = y .* (1 - y);
    dNdx = (v .* sigmoid_deriv)' * w;

    % Trial solution and its derivative
    Psi = A + x .* Nx;         % Satisfies IC
    dPsi = Nx + x .* dNdx;     % Chain rule
end
```

3.7 Results and Analysis

3.7.1 Convergence Behavior

Typical training:

- Iterations: 5,000-10,000
- Initial loss: $\sim 10^{-1}$
- Final loss: $< 10^{-6}$
- Training time: ~ 10 seconds
- Maximum error vs analytical: $< 10^{-5}$

3.7.2 Experimental Insights

1. **Network size**: 5 neurons sufficient for this problem; diminishing returns beyond
2. **Learning rate**: Optimal around 0.1; instability above 0.5
3. **Sampling**: 50 training points adequate; uniform spacing works well
4. **Initialization**: Xavier/Glorot initialization important for convergence

3.8 Common Pitfalls and Solutions

Problem	Solution
No convergence	Check learning rate, increase neurons
Slow training	Use vectorization, reduce points
Poor accuracy	More neurons, different activation
Unstable training	Reduce learning rate, gradient clipping

4 Week 4: Extension to PDEs

4.1 From ODEs to PDEs

The transition to partial differential equations opens up most economic applications:

- **Variables**: Multiple independent variables (e.g., state and time)
- **Derivatives**: Partial derivatives $\frac{\partial}{\partial x}$, $\frac{\partial}{\partial t}$
- **Domains**: 2D regions, 3D volumes, or higher
- **Conditions**: Initial AND boundary conditions

4.2 Network Architecture for PDEs

Key changes from ODE case:

- **Input layer**: Multiple neurons for multiple variables
- **Hidden layers**: Often need more neurons (20-100)
- **Output**: Still scalar for single equation
- **Derivatives**: Partial derivatives via automatic differentiation

Listing 3: 2D Network Structure

```
function N = network_2d(x, t, params)
    % Multiple inputs
    inputs = [x, t];

    % Hidden layers (may need more)
    h1 = tanh(inputs * params.W1 + params.b1);
    h2 = tanh(h1 * params.W2 + params.b2);

    % Output
    N = h2 * params.W3 + params.b3;
end
```

4.3 Application: Black-Scholes Equation

The Black-Scholes equation is fundamental in financial economics:

Economic Application

Black-Scholes PDE:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

with terminal condition: $V(S,T) = \max(S-K, 0)$ for call option

4.3.1 Economic Relevance

- Option pricing in financial markets
- Real options in investment decisions
- Sovereign debt pricing with default option
- Executive compensation (stock options)

4.3.2 Time Transformation

Black-Scholes has a terminal condition (payoff at expiry), not initial condition. Solution: transform time.

Let $\tau = T - t$ (time to maturity). Then:

- Terminal condition becomes initial: at $\tau = 0$, $V = \max(S - K, 0)$
- PDE sign changes: $\frac{\partial V}{\partial t} = -\frac{\partial V}{\partial \tau}$

4.3.3 Trial Solution for Black-Scholes

$$V_t(S, \tau) = \max(S - K, 0) + \tau \cdot S \cdot N(S, \tau; \theta)$$

This ensures:

- At expiry ($\tau = 0$): $V = \max(S - K, 0)$ $\surd$
- Network learns time value of option
- S factor ensures proper scaling with stock price

4.3.4 Implementation Considerations

1. **Domain**: $S \in [0.5K, 2K]$, $\tau \in [0, T]$
2. **Sampling**: Dense near strike price K (most nonlinearity)
3. **Network**: 20-50 hidden neurons typical
4. **Training**: 10,000-20,000 iterations
5. **Validation**: Compare with analytical Black-Scholes formula

4.4 High-Dimensional PDEs

The real power of neural networks: solving high-dimensional problems.

4.4.1 The Curse of Dimensionality

Traditional methods fail in high dimensions:

Dimension	Grid Points	NN Parameters
2D	$100^2 = 10^4$	$\sim 10^3$
5D	$100^5 = 10^{10}$	$\sim 10^3$
10D	100^{10} (impossible!)	$\sim 10^4$
100D	—	$\sim 10^5$

Neural networks scale linearly with dimension!

4.4.2 Economic Applications

High-dimensional problems in economics:

- **Heterogeneous agents**: Each agent type adds dimensions
- **Multi-asset portfolios**: Each asset adds a state variable
- **International trade**: Each country pair adds dimensions
- **Supply chains**: Each firm adds state variables

4.5 Summary and Future Directions

4.5.1 What We've Accomplished

Over four weeks, we've built a complete toolkit:

1. Understanding neural networks from first principles
2. Learning algorithms (gradient descent, backpropagation)
3. Solving ODEs with physics-informed neural networks
4. Extending to PDEs and high dimensions

4.5.2 Applications to Your Future Courses

These methods directly apply to:

- **Optimal Growth Models**:
  - Solve Bellman equations in high dimensions
  - Handle non-convexities and constraints
  - Continuous state spaces
- **Investment Models**:
  - Irreversibility constraints
  - Real options valuation
  - Firm heterogeneity
- **Sovereign Default**:
  - Discrete default choice with continuous debt
  - State-dependent borrowing costs
  - Multiple equilibria

4.5.3 Best Practices

1. **Start simple**: Basic network, few neurons, then scale up
2. **Validate carefully**: Check against known solutions
3. **Monitor training**: Watch loss function, check for overfitting
4. **Economic sense**: Ensure results satisfy economic intuition
5. **Document well**: Others should be able to reproduce your results

4.5.4 Resources for Further Learning

- **Code repository**: All MATLAB codes from lectures
- **DeepXDE**: Python library for physics-informed neural networks
- **Papers**:
  - Fernández-Villaverde et al. "Solving high-dimensional dynamic models"
  - Duarte "Gradient-based learning for sovereign default"
- **Online courses**: Fast.ai, Coursera Deep Learning Specialization

Appendix: Key Formulas and Code Snippets

Essential Formulas

Gradient Descent:

$$w_{t+1} = w_t - \eta \nabla L(w_t)$$

Backpropagation (2-layer):

$$\frac{\partial L}{\partial W_1} = \left(W_2^T \frac{\partial L}{\partial y} \odot f'(z_1) \right) x^T$$

Trial Solution (ODE):

$$\Psi_t(x) = \Psi(0) + x \cdot N(x; \theta)$$

Physics-Informed Loss:

$$L = \frac{1}{n} \sum_i [\text{ODE residual at } x_i]^2$$

Key MATLAB Functions

Sigmoid and derivative:

```
sigmoid = @(x) 1./(1 + exp(-x));
sigmoid_deriv = @(x) sigmoid(x).*(1 - sigmoid(x));
```

Xavier initialization:

```
W = randn(n_out, n_in) * sqrt(2/(n_in + n_out));
```

Computing ODE residual:

```
residual = dPsi_dx - (exp(-2*x) - 2*Psi);
loss = mean(residual.^2);
```