Section 3: Introduction to Deep Learning & Neural Networks

Comprehensive PyTorch Theory for Power System Optimization

Dr Bibin Wilson & Prof Anand Singh

September 10, 2025

Indian Institute of Technology Bombay

Section Overview

Neural Network Fundamentals

PyTorch Fundamentals

Multi-Layer Perceptron Architectures

Energy-Specific Applications

Advanced Topics

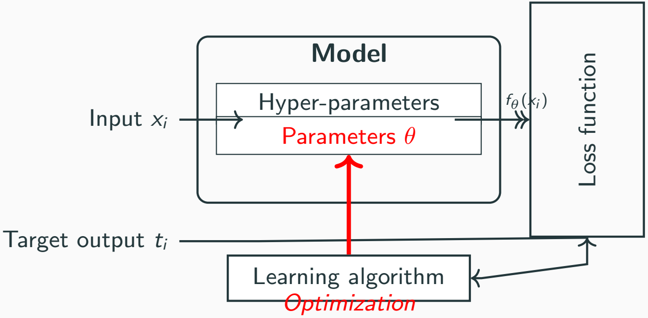
Supervised Machine Learning System - Training

Elements of a model:

- Input x_i
- Function $f_\theta(x_i)$

Utility of the model:

- Target output t_i
- Bring $f_\theta(x_i)$ close to t_i
- Minimize loss $L(t_i, f_\theta(x_i))$



Neural Network Fundamentals

The Perceptron: Foundation of Neural Networks

Mathematical Model:

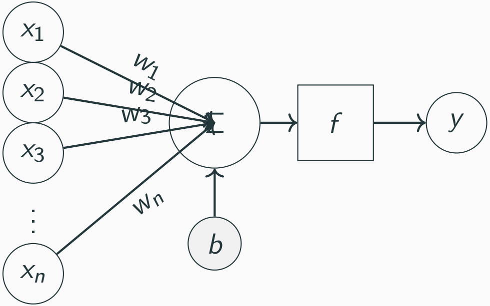
$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

Components:

- **Inputs**: $\mathbf{x} = [x_1, x_2, ..., x_n]^T$
- **Weights**: $\mathbf{w} = [w_1, w_2, ..., w_n]^T$
- **Bias**: b (threshold adjustment)
- **Activation**: $f(\cdot)$ (non-linearity)

Learning Rule (Rosenblatt):

$$w_i^{(t+1)} = w_i^{(t)} + \eta(y_{true} - y_{pred})x_i$$



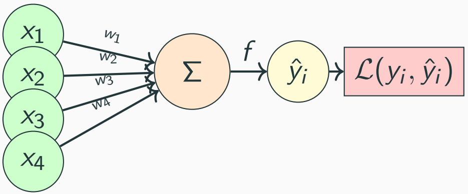
Energy Application:

- Load threshold detection
- Fault classification (binary)
- Peak/off-peak identification

Neural Network Loss Function and Optimization

Problem Formulation:

- **Input**: $\mathbf{x}_i \in \mathbb{R}^n$
- **Output**: $y_i \in \mathbb{R}$
- **Prediction**: $\hat{y}_i = f(\mathbf{w}^T \mathbf{x}_i)$



Loss Function Definition:

$$\mathcal{L}(y_i, \hat{y}_i) = \frac{1}{2}(y_i - \hat{y}_i)^2$$

Total Loss (MSE):

$$\mathcal{L} = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Optimization: Finding Optimal Weights

Optimization Problem:

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} \sum_{i=1}^{n} (y_i - f(\mathbf{w}^T \mathbf{x}_i))^2 + \lambda \|\mathbf{w}\|^2$$

Components:

- **Loss term**: $\sum_{i=1}^{n} (y_i - \hat{y}_i)^2$
- **Regularization**: $\lambda \|\mathbf{w}\|^2$ (prevents overfitting)
- **Activation**: $y_i = f(\mathbf{w}^T \mathbf{x}_i)$

When f is Identity (Linear Regression):

$$\min_{\mathbf{w}} \sum_{i=1}^{n} (y_i - \mathbf{w}^T \mathbf{x}_i)^2$$

Closed-form solution: $\hat{\mathbf{w}} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$

Key Concepts

Why Regularization?

- Controls model complexity
- Reduces overfitting
- Improves generalization

Gradient Descent: Iterative Optimization

Gradient Computation:

$$\nabla_{\mathbf{w}}\mathcal{L} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial w_1} \\ \frac{\partial \mathcal{L}}{\partial w_2} \\ \vdots \\ \frac{\partial \mathcal{L}}{\partial w_n} \end{bmatrix} \in \mathbb{R}^n$$

Update Rule:

$$\mathbf{w}_{new} = \mathbf{w}_{old} - \eta \cdot [\nabla_{\mathbf{w}}\mathcal{L}]_{\mathbf{w}_{old}}$$

Component-wise Update:

$$(w_i)_{new} = (w_i)_{old} - \eta \cdot \left[\frac{\partial \mathcal{L}}{\partial w_i}\right]_{w_i^{old}}$$

Algorithm: SGD

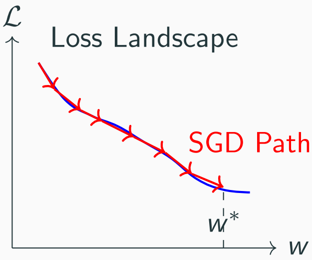
Initialize: $\mathbf{w} \sim \mathcal{N}(0, \sigma^2)$ (random) iter $= 1$ to K Sample mini-batch $\mathcal{B}$ Compute $\nabla_{\mathbf{w}}\mathcal{L}_{\mathcal{B}}$ $\mathbf{w} \leftarrow \mathbf{w} - \eta \cdot \nabla_{\mathbf{w}}\mathcal{L}_{\mathcal{B}}$ convergence criteria met **break** **Return:** $\mathbf{w}$

Hyperparameters:

- η: Learning rate (e.g., 0.01)
- K: Max iterations
- Batch size: 32, 64, 128...

Gradient Descent Variants and Convergence

1. **Batch Gradient Descent:**
   - Uses entire dataset
   - Stable convergence
   - Computationally expensive
2. **Stochastic GD (SGD):**
   - Single sample per update
   - Noisy but faster
   - Can escape local minima
3. **Mini-batch GD:**
   - Balance between Batch and SGD
   - GPU-efficient
   - Most commonly used



Convergence Criteria:

- $|\mathcal{L}_t - \mathcal{L}_{t-1}| < \epsilon$
- $\|\nabla_{\mathbf{w}}\mathcal{L}\| < \epsilon$
- Validation loss plateaus

Activation Functions: Mathematical Analysis

1. Sigmoid:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

- Range: $(0, 1)$
- Issue: Vanishing gradient for $|x| > 4$

2. Tanh:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$\tanh'(x) = 1 - \tanh^2(x)$$

3. ReLU:

$$\mathrm{ReLU}(x) = \max(0, x)$$

$$\mathrm{ReLU}'(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$$

- Sparse activation
- Dead neuron problem

Implementation in Notebook:

`03_deep_learning_advanced.ipynb`

Advanced Activation Functions

4. Leaky ReLU:

$$\text{LeakyReLU}(x) = \begin{cases} x & x > 0 \\ \alpha x & x \leq 0 \end{cases}$$

5. ELU (Exponential Linear Unit):

$$\text{ELU}(x) = \begin{cases} x & x > 0 \\ \alpha(e^x - 1) & x \leq 0 \end{cases}$$

6. GELU (Gaussian Error Linear):

$$\text{GELU}(x) = x \cdot \Phi(x)$$

where $\Phi(x)$ is the CDF of standard normal

7. Swish/SiLU:

$$\text{Swish}(x) = x \cdot \sigma(\beta x)$$

Comparison for Energy Data:

- ReLU: Fast, simple, good for deep networks
- GELU: Better for transformers
- ELU: Smooth, handles negative values
- Swish: Self-gated, adaptive

See Activation Comparisons:

`03_deep_learning_advanced.ipynb`

Gradient Descent: Mathematical Foundation

Objective Function:

$$J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(f(\mathbf{x}^{(i)};\boldsymbol{\theta}), y^{(i)})$$

Update Rules:

1. Vanilla SGD:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta\nabla J(\boldsymbol{\theta}_t)$$

2. Momentum:

$$\mathbf{v}_{t+1} = \beta\mathbf{v}_t + \eta\nabla J(\boldsymbol{\theta}_t) \tag{1}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_{t+1} \tag{2}$$

Learning Rate Schedules:

1. Step Decay:

$$\eta_t = \eta_0 \cdot \gamma^{\lfloor t/s \rfloor}$$

2. Exponential Decay:

$$\eta_t = \eta_0 \cdot e^{-\lambda t}$$

3. Cosine Annealing:

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})(1 + \cos(\frac{t\pi}{T}))$$

Backpropagation Algorithm

Forward Pass:

$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]} \tag{3}$$
$$a^{[l]} = f^{[l]}(z^{[l]}) \tag{4}$$

Backward Pass:

$$\delta^{[L]} = \nabla_{a^{[L]}} J \odot f'^{[L]}(z^{[L]}) \tag{5}$$
$$\delta^{[l]} = (W^{[l+1]})^T \delta^{[l+1]} \odot f'^{[l]}(z^{[l]}) \tag{6}$$

Gradient Computation:

$$\frac{\partial J}{\partial W^{[l]}} = \delta^{[l]} (a^{[l-1]})^T \tag{7}$$
$$\frac{\partial J}{\partial b^{[l]}} = \delta^{[l]} \tag{8}$$

Chain Rule Application:

$$\frac{\partial J}{\partial w_{ij}^{[l]}} = \frac{\partial J}{\partial z_i^{[l]}} \cdot \frac{\partial z_i^{[l]}}{\partial w_{ij}^{[l]}}$$

Key Insights:

- Error propagates backward
- Gradients flow through network
- Activation derivatives crucial
- Vanishing/exploding gradients

Manual Implementation:

`03_deep_learning_advanced.ipynb`

Universal Approximation Theorem

Theorem Statement:

> A feedforward network with a single hidden layer containing a finite number of neurons can approximate any continuous function on a compact subset of $\mathbb{R}^n$ to arbitrary accuracy.

Formal Definition: For any $\epsilon > 0$ and continuous $f : K \to \mathbb{R}$ where $K \subset \mathbb{R}^n$ is compact, there exists:

$$F(\mathbf{x}) = \sum_{i=1}^{N} \alpha_i \sigma(\mathbf{w}_i^T \mathbf{x} + b_i)$$

such that $|F(\mathbf{x}) - f(\mathbf{x})| < \epsilon$ for all $\mathbf{x} \in K$.

Depth Efficiency:

- Shallow: $O(2^n)$ neurons
- Deep: $O(n)$ neurons

Energy System Example: Can approximate:

- Load curves
- Price functions
- Demand response

Implications:

- Width vs Depth trade-off
- Expressiveness guarantee
- No training guarantee

PyTorch Fundamentals

PyTorch Tensors: Core Concepts

Tensor Properties:

- **Shape**: Dimensions of data
- **dtype**: Data type (float32, int64, etc.)
- **device**: CPU or GPU location
- **requires_grad**: Track gradients

Key Operations:

- Creation: zeros, ones, randn, arange
- Math: matmul, pow, exp, log
- Reduction: mean, std, max, sum
- Reshaping: view, reshape, squeeze
- Indexing: Advanced slicing, masking

Memory Management:

- Contiguous memory layout
- In-place operations (_suffix)
- View vs Copy semantics
- GPU memory optimization

Tensor Operations Demo:

`03_deep_learning_advanced.ipynb`

Complete tensor manipulation examples
with energy data applications

Autograd: Automatic Differentiation

Computational Graph:

- Dynamic graph construction
- Forward pass builds graph
- Backward pass computes gradients
- Automatic chain rule application

Gradient Computation: For $loss = (x \cdot y)^2$ where $x = 2, y = 3$:

$$\frac{\partial loss}{\partial x} = 2xy^2 = 36 \tag{9}$$

$$\frac{\partial loss}{\partial y} = 2x^2y = 24 \tag{10}$$

Custom Autograd Functions:

- Define forward pass
- Define backward pass
- Save tensors for backward
- Handle non-differentiable ops

Gradient Control:

- `no_grad()` context
- `detach()` operations
- Gradient accumulation
- Gradient clipping

Autograd Examples:

`03_deep_learning_advanced.ipynb`

Optimizers Comparison

SGD with Momentum:

- Classic, well-understood
- Good for convex problems
- Requires careful LR tuning

Adam and Variants:

- **Adam**: Adaptive moments
- **AdamW**: Decoupled weight decay
- **RAdam**: Rectified Adam
- **LAMB**: Layer-wise adaptive

Other Optimizers:

- **RMSprop**: Running average of gradients
- **AdaGrad**: Adaptive learning rates

Selection Guidelines:

- **SGD**: Final fine-tuning
- **Adam**: Default choice
- **AdamW**: With weight decay
- **LAMB**: Large batch training

Learning Rate Scheduling:

- StepLR: Step decay
- ExponentialLR: Exponential decay
- CosineAnnealingLR: Cosine schedule
- ReduceLROnPlateau: Adaptive

Optimizer Comparisons:

Multi-Layer Perceptron Architectures

Building Deep MLPs

Architecture Components:

- Linear transformations
- Activation functions
- Normalization layers
- Dropout for regularization
- Skip connections

Design Patterns:

- Pyramidal: Decreasing width
- Constant: Same width
- Expanding: Increasing width
- Bottleneck: Narrow middle

Initialization Strategies:

- **Xavier/Glorot**: For tanh, sigmoid
- **He/Kaiming**: For ReLU variants
- **Orthogonal**: For RNNs
- **LSUV**: Layer-sequential unit-variance

Deep MLP Implementation:

`03_deep_learning_advanced.ipynb`

Complete architectures with
batch norm, dropout, and residuals

Regularization Techniques

Weight Regularization:

- **L2 (Weight Decay)**: $\lambda \sum w_i^2$
- **L1 (Sparsity)**: $\lambda \sum |w_i|$
- **Elastic Net**: L1 + L2

Dropout Variants:

- Standard: Random neuron dropping
- Alpha: For SELU activation
- Variational: Consistent mask
- Spatial: For convolutional layers

Normalization:

- **Batch Norm**: Across batch
- **Layer Norm**: Across features
- **Instance Norm**: Per sample
- **Group Norm**: Feature groups

Other Techniques:

- Early stopping
- Data augmentation
- Label smoothing
- Mixup training

Skip Connections and Residual Blocks

Residual Blocks:

$$\mathbf{y} = \mathbf{x} + \mathcal{F}(\mathbf{x})$$

Benefits:

- Gradient flow preservation
- Identity mapping option
- Deeper network training
- Reduced vanishing gradients

Dense Connections:

$$\mathbf{x}_l = H_l([\mathbf{x}_0, \mathbf{x}_1, ..., \mathbf{x}_{l-1}])$$

Highway Networks:

$$\mathbf{y} = T(\mathbf{x}) \cdot H(\mathbf{x}) + (1 - T(\mathbf{x})) \cdot \mathbf{x}$$

where T is the transform gate.

ResNet MLP Implementation:

`03_deep_learning_advanced.ipynb`

Skip connections, residual blocks, and highway networks

Energy-Specific Applications

Load Forecasting Architecture

Input Features:

- Historical load (24h, 168h)
- Temperature, humidity
- Calendar features
- Holiday indicators
- Economic indicators

Architecture Design:

- Temporal convolutions
- Calendar embeddings
- Weather processing subnet
- LSTM for sequences
- Multi-horizon output

Feature Engineering:

- Cyclical encoding (hour, day)
- Lag features (t-1, t-24, t-168)
- Rolling statistics
- Peak indicators
- Ramp rates

Complete Implementation:

`03_deep_learning_advanced.ipynb`

Energy-specific networks with feature engineering pipelines

Handling Seasonality and Trends

Decomposition Approach:

$$Y_t = T_t + S_t + R_t$$

where:

- T_t: Trend component
- S_t: Seasonal component
- R_t: Residual component

Neural Decomposition:

- Trend extraction CNN
- Seasonal pattern networks
- Residual processing
- Component combination

Multi-Scale Processing:

- Daily patterns (24h)
- Weekly patterns (168h)
- Monthly patterns
- Annual patterns

Adaptive Methods:

- Online learning updates
- Concept drift handling
- Window-based retraining

Advanced Topics

Advanced Training Techniques

Mixed Precision Training:

- FP16 for forward/backward
- FP32 master weights
- Gradient scaling
- Memory savings (50%)
- Speed improvements (2-3x)

Distributed Training:

- Data parallel (DDP)
- Model parallel
- Pipeline parallel
- Gradient accumulation

Model Optimization:

- Quantization (INT8)
- Pruning (structured/unstructured)
- Knowledge distillation
- Neural architecture search

Advanced Implementations:

`03_deep_learning_advanced.ipynb`

Mixed precision, quantization,
and production deployment

Summary: Deep Learning Foundations

Key Concepts Covered:

- Neural Network Fundamentals
- Activation Functions
- Backpropagation
- Universal Approximation
- PyTorch Tensors & Autograd
- Optimizers & Loss Functions
- MLP Architectures
- Regularization Techniques

Energy Applications:

- Load Forecasting
- Feature Engineering
- Seasonality Handling
- Real-time Systems

Next: CNNs for Solar Panel Defects

`Section 4 & 04_cnn_solar_advanced.ipynb`