Multi-Layer Perceptron Calculator

Technical Manual and Mathematical Foundations

English

Wolfgang Boehme

wolfgang.boehme@economic-modeling.de

www.economic-modeling.de

25 October 2026

The documentation on this website was developed with the support of the AI Claude (Anthropic).

Table of Contents

1 Introduction **3**
1.1 Overview . . . 3
1.2 Application Areas . . . 3

2 Mathematical Foundations **3**
2.1 Network Architecture . . . 3
2.2 Neuron Model . . . 4
2.3 Matrix Notation . . . 4

3 Activation Functions **4**
3.1 Sigmoid Function . . . 4
3.2 ReLU (Rectified Linear Unit) . . . 5
3.3 Tanh Function . . . 5
3.4 Linear Function . . . 5

4 Forward Propagation **6**
4.1 Algorithm . . . 6
4.2 Example Calculation . . . 6

5 Loss Functions **6**
5.1 Mean Squared Error (MSE) . . . 6
5.2 Binary Cross-Entropy (BCE) . . . 7

6 Backpropagation **7**
6.1 Fundamental Principle . . . 7
6.2 Output Layer Gradient . . . 7
6.3 Hidden Layer Gradients . . . 7
6.4 Weight and Bias Gradients . . . 8
6.5 Backpropagation Algorithm . . . 8

7 Gradient Descent **8**
7.1 Standard Gradient Descent . . . 8
7.2 Batch Gradient Descent . . . 9
7.3 Mini-Batch Gradient Descent . . . 9

8 Training Algorithm **9**
8.1 Complete Training Loop . . . 9

9 Data Normalisation **10**
9.1 Motivation . . . 10
9.2 Min-Max Normalisation . . . 10
9.3 Denormalisation . . . 10
9.4 Normalisation Parameters . . . 10

10 Weight Initialisation **10**
10.1 Random Initialisation . . . 10
10.2 Xavier/Glorot Initialisation (recommended) . . . 11

11 Hyperparameters **11**
11.1 Learning Rate η . . . 11
11.2 Number of Epochs . . . 11
11.3 Batch Size B . . . 11
11.4 Network Architecture . . . 12

12 Using the Calculator **12**
12.1 Workflow . . . 12
12.2 Example: Currency Prediction . . . 13

13 Troubleshooting **13**
13.1 Training Does Not Converge . . . 13
13.2 Overfitting . . . 13
13.3 Underfitting . . . 14
13.4 Numerical Instability . . . 14

14 Mathematical Proofs **14**
14.1 Universal Approximation Theorem . . . 14
14.2 Gradient Descent Convergence . . . 14

15 Implementation Details **15**
15.1 Matrix Operations . . . 15
15.2 Complexity Analysis . . . 15

16 Visualisations **15**
16.1 Network Graph . . . 15
16.2 Loss Curve . . . 15
16.3 Weight Distribution . . . 16

17 Export Formats **16**
17.1 JSON Format for Weights . . . 16
17.2 CSV Format for Training Data . . . 16

18 Future Extensions **16**
18.1 Planned Features . . . 16
18.2 Dropout . . . 17
18.3 Momentum . . . 17
18.4 Adam Optimizer . . . 17

19 References and Further Reading **17**
19.1 Foundational Works . . . 17
19.2 Textbooks . . . 18

20 Appendix **18**
20.1 Formula Reference . . . 18
20.2 Glossary . . . 18
20.3 Notation . . . 19

21 Closing Remarks **19**

Abstract

This technical manual describes the mathematical foundations, algorithms and implementation details of the Multi-Layer Perceptron (MLP) Calculator. The document is aimed at users who wish to gain a deeper understanding of the underlying neural network theory, as well as at developers who want to follow or extend the implementation. All formulae used, the backpropagation algorithm, activation functions, loss functions and optimisation methods are explained in detail.

1 Introduction

1.1 Overview

The Multi-Layer Perceptron (MLP) is a fundamental artificial neural network consisting of multiple layers of neurons. The present calculator implements a complete MLP with the following characteristics:

- Configurable number of input neurons (1–10)
- Variable number of hidden layers (2–5)
- Flexible number of neurons per hidden layer (1–20)
- One output neuron for regression or binary classification
- Different activation functions per layer
- Backpropagation-based training
- Real-time visualisation of training progress

1.2 Application Areas

MLPs are suitable for a wide range of tasks:

- **Regression:** Prediction of continuous values (e.g. exchange rates, prices)
- **Binary classification:** Assignment to two classes (e.g. spam / not spam)
- **Function approximation:** Learning complex nonlinear relationships
- **Pattern recognition:** Identification of patterns in data

2 Mathematical Foundations

2.1 Network Architecture

Definition 2.1 (Multi-Layer Perceptron)**.** A Multi-Layer Perceptron is a directed acyclic graph consisting of L layers, where:

- Layer 0: Input layer with n_0 neurons
- Layer l for $1 \leq l < L-1$: Hidden layers with n_l neurons
- Layer $L-1$: Output layer with n_{L-1} neurons

The network architecture is described by the vector:

$$\mathcal{N} = (n_0, n_1, n_2, \ldots, n_{L-1}) \tag{1}$$

Example: $(7, 16, 8, 1)$ describes a network with 7 input neurons, two hidden layers with 16 and 8 neurons respectively, and one output neuron.

2.2 Neuron Model

Definition 2.2 (Artificial Neuron)**.** An artificial neuron j in layer l computes its output in two steps:

1. **Weighted sum (pre-activation):**

$$z_j^{(l)} = \sum_{i=1}^{n_{l-1}} w_{ji}^{(l)} \cdot a_i^{(l-1)} + b_j^{(l)} \tag{2}$$

2. **Activation (post-activation):**

$$a_j^{(l)} = f^{(l)}(z_j^{(l)}) \tag{3}$$

where:

- $w_{ji}^{(l)}$ is the weight from neuron i in layer $l-1$ to neuron j in layer l
- $b_j^{(l)}$ is the bias of neuron j in layer l
- $a_i^{(l-1)}$ is the activation of neuron i in the previous layer
- $f^{(l)}$ is the activation function in layer l

2.3 Matrix Notation

For efficient computation we use matrix notation. For layer l:

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)} \tag{4}$$

$$\mathbf{a}^{(l)} = f^{(l)}(\mathbf{z}^{(l)}) \tag{5}$$

where:

- $\mathbf{W}^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$ is the weight matrix
- $\mathbf{b}^{(l)} \in \mathbb{R}^{n_l}$ is the bias vector
- $\mathbf{a}^{(l)} \in \mathbb{R}^{n_l}$ is the activation vector
- $f^{(l)}$ is applied element-wise to the vector

3 Activation Functions

Activation functions enable the network to learn nonlinear relationships.

3.1 Sigmoid Function

The sigmoid function maps values to the interval $(0, 1)$:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{6}$$

Derivative:

$$\sigma'(z) = \sigma(z) \cdot (1 - \sigma(z)) \tag{7}$$

Properties:

- Output between 0 and 1
- Smooth and differentiable
- Interpretable as a probability
- Problem: Vanishing gradients when $|z| \gg 0$

3.2 ReLU (Rectified Linear Unit)

$$\mathrm{ReLU}(z) = \max(0, z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases} \tag{8}$$

Derivative:

$$\mathrm{ReLU}'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases} \tag{9}$$

Properties:

- Very efficient to compute
- No vanishing gradient for $z > 0$
- Standard for deep networks
- Problem: "Dying ReLU" when $z \leq 0$

3.3 Tanh Function

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = \frac{e^{2z} - 1}{e^{2z} + 1} \tag{10}$$

Derivative:

$$\tanh'(z) = 1 - \tanh^2(z) \tag{11}$$

Properties:

- Output between -1 and 1
- Centred at 0
- Stronger gradients than sigmoid
- Similar vanishing-gradient problem to sigmoid

3.4 Linear Function

$$\mathrm{linear}(z) = z \tag{12}$$

Derivative:

$$\mathrm{linear}'(z) = 1 \tag{13}$$

Use:

- Typical for the output layer in regression
- No restriction on the value range

4 Forward Propagation

4.1 Algorithm

The forward pass computes the network output for a given input $\mathbf{x}$.

Algorithm 1 Forward Propagation

Require: Input vector $\mathbf{x} \in \mathbb{R}^{n_0}$, weights $\{\mathbf{W}^{(l)}\}_{l=1}^{L-1}$, biases $\{\mathbf{b}^{(l)}\}_{l=1}^{L-1}$
Ensure: Output $\mathbf{a}^{(L-1)}$

$\mathbf{a}^{(0)} \leftarrow \mathbf{x}$ ▷ Input layer
for $l = 1$ to $L-1$ **do** ▷ For each layer
  $\mathbf{z}^{(l)} \leftarrow \mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}$ ▷ Weighted sum
  $\mathbf{a}^{(l)} \leftarrow f^{(l)}(\mathbf{z}^{(l)})$ ▷ Activation
end for
return $\mathbf{a}^{(L-1)}$ ▷ Output

4.2 Example Calculation

Given a network with architecture $(2, 3, 1)$ and input $\mathbf{x} = (0.5, 0.8)^T$.

Layer 1 (Input → Hidden):

$$\mathbf{W}^{(1)} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{pmatrix}, \quad \mathbf{b}^{(1)} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} \tag{14}$$

$$\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} = \begin{pmatrix} w_{11} \cdot 0.5 + w_{12} \cdot 0.8 + b_1 \\ w_{21} \cdot 0.5 + w_{22} \cdot 0.8 + b_2 \\ w_{31} \cdot 0.5 + w_{32} \cdot 0.8 + b_3 \end{pmatrix} \tag{15}$$

$$\mathbf{a}^{(1)} = \mathrm{ReLU}(\mathbf{z}^{(1)}) \tag{16}$$

Layer 2 (Hidden → Output):

$$z^{(2)} = w_1^{(2)} a_1^{(1)} + w_2^{(2)} a_2^{(1)} + w_3^{(2)} a_3^{(1)} + b^{(2)} \tag{17}$$

$$a^{(2)} = \sigma(z^{(2)}) \tag{18}$$

5 Loss Functions

The loss function $\mathcal{L}$ measures the discrepancy between the prediction $\hat{y}$ and the true value y.

5.1 Mean Squared Error (MSE)

For regression problems:

$$\mathcal{L}_{\mathrm{MSE}}(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2 \tag{19}$$

Derivative:

$$\frac{\partial \mathcal{L}_{\mathrm{MSE}}}{\partial \hat{y}} = -(y - \hat{y}) = \hat{y} - y \tag{20}$$

For a dataset with N samples:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{21}$$

5.2 Binary Cross-Entropy (BCE)

For binary classification:

$$\mathcal{L}_{\text{BCE}}(y, \hat{y}) = -[y \log(\hat{y}) + (1-y)\log(1-\hat{y})] \tag{22}$$

Derivative:

$$\frac{\partial \mathcal{L}_{\text{BCE}}}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})} \tag{23}$$

For a dataset:

$$\mathcal{L}_{\text{BCE}} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)] \tag{24}$$

6 Backpropagation

Backpropagation is the algorithm for computing the gradients of the loss function with respect to all network parameters.

6.1 Fundamental Principle

We apply the chain rule to propagate gradients backwards from the output layer through the network.

Theorem 6.1 (Chain Rule for Neural Networks)**.** For the derivative of the loss function with respect to the weights:

$$\frac{\partial \mathcal{L}}{\partial w_{ji}^{(l)}} = \frac{\partial \mathcal{L}}{\partial z_j^{(l)}} \cdot \frac{\partial z_j^{(l)}}{\partial w_{ji}^{(l)}} \tag{25}$$

6.2 Output Layer Gradient

For the output layer $L-1$ with one neuron:

$$\delta^{(L-1)} = \frac{\partial \mathcal{L}}{\partial z^{(L-1)}} = \frac{\partial \mathcal{L}}{\partial a^{(L-1)}} \cdot \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} \tag{26}$$

With MSE loss and sigmoid activation:

$$\delta^{(L-1)} = (a^{(L-1)} - y) \cdot \sigma'(z^{(L-1)}) = (a^{(L-1)} - y) \cdot a^{(L-1)} \cdot (1 - a^{(L-1)}) \tag{27}$$

6.3 Hidden Layer Gradients

For hidden layers $l < L-1$:

$$\delta_j^{(l)} = \left(\sum_{k=1}^{n_{l+1}} w_{kj}^{(l+1)} \delta_k^{(l+1)}\right) \cdot f'^{(l)}(z_j^{(l)}) \tag{28}$$

In matrix notation:

$$\boldsymbol{\delta}^{(l)} = \left[(\mathbf{W}^{(l+1)})^T \boldsymbol{\delta}^{(l+1)}\right] \odot f'^{(l)}(\mathbf{z}^{(l)}) \tag{29}$$

where $\odot$ is the Hadamard product (element-wise multiplication).

6.4 Weight and Bias Gradients

Using the deltas we can compute the gradients:

Weights:

$$\frac{\partial \mathcal{L}}{\partial w_{ji}^{(l)}} = \delta_j^{(l)} \cdot a_i^{(l-1)} \tag{30}$$

Biases:

$$\frac{\partial \mathcal{L}}{\partial b_j^{(l)}} = \delta_j^{(l)} \tag{31}$$

In matrix form:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} = \boldsymbol{\delta}^{(l)} (\mathbf{a}^{(l-1)})^T \tag{32}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}} = \boldsymbol{\delta}^{(l)} \tag{33}$$

6.5 Backpropagation Algorithm

Algorithm 2 Backpropagation

Require: Input $\mathbf{x}$, target y, activations $\{\mathbf{a}^{(l)}\}$, pre-activations $\{\mathbf{z}^{(l)}\}$
Ensure: Gradients $\{\nabla_{\mathbf{W}^{(l)}} \mathcal{L}\}$, $\{\nabla_{\mathbf{b}^{(l)}} \mathcal{L}\}$

// **Output layer**
$\boldsymbol{\delta}^{(L-1)} \leftarrow \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(L-1)}} \odot f'^{(L-1)}(\mathbf{z}^{(L-1)})$

// **Hidden layers (backwards)**
for $l = L-2$ down to 1 **do**
    $\boldsymbol{\delta}^{(l)} \leftarrow [(\mathbf{W}^{(l+1)})^T \boldsymbol{\delta}^{(l+1)}] \odot f'^{(l)}(\mathbf{z}^{(l)})$
end for

// **Compute gradients**
for $l = 1$ to $L-1$ **do**
    $\nabla_{\mathbf{W}^{(l)}} \mathcal{L} \leftarrow \boldsymbol{\delta}^{(l)} (\mathbf{a}^{(l-1)})^T$
    $\nabla_{\mathbf{b}^{(l)}} \mathcal{L} \leftarrow \boldsymbol{\delta}^{(l)}$
end for
return Gradients

7 Gradient Descent

7.1 Standard Gradient Descent

Parameter update with learning rate η:

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta \cdot \nabla_{\mathbf{W}^{(l)}} \mathcal{L} \tag{34}$$

$$\mathbf{b}^{(l)} \leftarrow \mathbf{b}^{(l)} - \eta \cdot \nabla_{\mathbf{b}^{(l)}} \mathcal{L} \tag{35}$$

7.2 Batch Gradient Descent

For a dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$:

$$\nabla_{\mathbf{W}^{(l)}} \mathcal{L}_{\text{total}} = \frac{1}{N} \sum_{i=1}^{N} \nabla_{\mathbf{W}^{(l)}} \mathcal{L}(\mathbf{x}_i, y_i) \tag{36}$$

7.3 Mini-Batch Gradient Descent

Divide the dataset into batches of size B:

$$\nabla_{\mathbf{W}^{(l)}} \mathcal{L}_{\text{batch}} = \frac{1}{B} \sum_{i \in \text{batch}} \nabla_{\mathbf{W}^{(l)}} \mathcal{L}(\mathbf{x}_i, y_i) \tag{37}$$

Advantages:

- Faster convergence than batch GD
- Lower memory requirements than batch GD
- Regularisation effect through noise

8 Training Algorithm

8.1 Complete Training Loop

Algorithm 3 Neural Network Training

Require: Dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$, network parameters, learning rate η, epochs E, batch size B

Ensure: Trained parameters

for epoch $= 1$ to E **do**
  Shuffle $\mathcal{D}$ ▷ Shuffle dataset
  $\mathcal{L}_{\text{epoch}} \leftarrow 0$
  for each batch in $\mathcal{D}$ **do**
    $\nabla_{\mathbf{W}} \mathcal{L} \leftarrow 0$, $\nabla_{\mathbf{b}} \mathcal{L} \leftarrow 0$ ▷ Initialise gradients
    for each $(\mathbf{x}, y)$ in batch **do**
      $\hat{y} \leftarrow \text{ForwardProp}(\mathbf{x})$ ▷ Forward pass
      $\mathcal{L}_{\text{epoch}} \leftarrow \mathcal{L}_{\text{epoch}} + \mathcal{L}(y, \hat{y})$
      $\nabla_{\mathbf{W}} \mathcal{L} \leftarrow \nabla_{\mathbf{W}} \mathcal{L} + \text{BackProp}(\mathbf{x}, y)$ ▷ Accumulate gradients
    end for
    $\nabla_{\mathbf{W}} \mathcal{L} \leftarrow \nabla_{\mathbf{W}} \mathcal{L} / B$ ▷ Average
    $\mathbf{W} \leftarrow \mathbf{W} - \eta \cdot \nabla_{\mathbf{W}} \mathcal{L}$ ▷ Update
    $\mathbf{b} \leftarrow \mathbf{b} - \eta \cdot \nabla_{\mathbf{b}} \mathcal{L}$ ▷ Update
  end for
  $\mathcal{L}_{\text{epoch}} \leftarrow \mathcal{L}_{\text{epoch}} / N$
  Log progress and loss
end for

9 Data Normalisation

9.1 Motivation

Data with very different magnitudes can cause problems:

- Numerical instability (overflow/underflow)
- Slow convergence
- Vanishing or exploding gradients

9.2 Min-Max Normalisation

For feature j with values $x_j^{(1)}, \ldots, x_j^{(N)}$:

$$x_j^{\text{norm}} = \frac{x_j - \min(x_j)}{\max(x_j) - \min(x_j)} \tag{38}$$

Properties:

- Scales to the interval $[0, 1]$
- Preserves relative distances
- All features on the same scale

9.3 Denormalisation

For predictions we need to transform back:

$$x_j^{\text{orig}} = x_j^{\text{norm}} \cdot (\max(x_j) - \min(x_j)) + \min(x_j) \tag{39}$$

9.4 Normalisation Parameters

We store for each feature:

$$\text{params}_j = \begin{cases} \min_j = \min(x_j^{(1)}, \ldots, x_j^{(N)}) \\ \max_j = \max(x_j^{(1)}, \ldots, x_j^{(N)}) \\ \text{range}_j = \max_j - \min_j \end{cases} \tag{40}$$

10 Weight Initialisation

10.1 Random Initialisation

Weights are randomly initialised to break symmetry:

$$w_{ji}^{(l)} \sim \mathcal{U}(-1, 1) \tag{41}$$

10.2 Xavier/Glorot Initialisation (recommended)

For better convergence:

$$w_{ji}^{(l)} \sim \mathcal{N}\left(0, \frac{2}{n_{l-1} + n_l}\right) \tag{42}$$

or uniformly:

$$w_{ji}^{(l)} \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_{l-1} + n_l}}, \sqrt{\frac{6}{n_{l-1} + n_l}}\right) \tag{43}$$

11 Hyperparameters

11.1 Learning Rate η

Meaning: Step size for gradient descent

- Too large: Instability, oscillation, divergence
- Too small: Slow convergence, local minima
- Typical values: 0.001 to 0.1

$$\eta_{\text{optimal}} \approx \frac{1}{\lambda_{\max}(\mathbf{H})} \tag{44}$$

where $\lambda_{\max}(\mathbf{H})$ is the largest eigenvalue of the Hessian matrix.

11.2 Number of Epochs

Meaning: How many times the network passes through the entire dataset

- Too few: Underfitting, poor generalisation
- Too many: Overfitting to training data
- Typical values: 1 000 to 10 000 depending on the problem

11.3 Batch Size B

Meaning: Number of samples per gradient update

- Small ($B = 1$): Stochastic GD, high noise, fast updates
- Large ($B = N$): Batch GD, stable, slow updates
- Medium ($B = 32$–256): Good compromise

$$\text{Updates per epoch} = \left\lceil \frac{N}{B} \right\rceil \tag{45}$$

11.4 Network Architecture

Number of hidden layers:

- 0–1: Linear / simple nonlinear problems
- 2–3: Complex nonlinear problems (standard)
- ≥ 4: Deep learning, very complex problems

Neurons per layer:

$$n_{\text{hidden}} \approx \frac{n_{\text{input}} + n_{\text{output}}}{2} \tag{46}$$

or according to the Kolmogorov-Nielsen theorem:

$$n_{\text{hidden}} \geq 2 \cdot n_{\text{input}} + 1 \tag{47}$$

12 Using the Calculator

12.1 Workflow

Step 1: Configure network

1. Input neurons corresponding to number of features
2. Choose 2–3 hidden layers
3. Neurons per layer: start with $2 \times n_{\text{input}}$
4. Activation functions: ReLU for hidden, sigmoid for output
5. Learning rate: 0.01 or 0.001
6. Loss function: MSE for regression, BCE for classification

Step 2: Load data

1. Prepare CSV file (inputs, then output in last column)
2. For different magnitudes: activate normalisation
3. Upload data and check statistics

Step 3: Training

1. Set epochs (start: 1 000–3 000)
2. Batch size: 32 (standard)
3. Start training
4. Observe loss curve
5. Adjust learning rate if necessary

Step 4: Evaluation

1. Check network visualisation
2. Analyse weight distribution
3. Perform test predictions
4. Export results

12.2 Example: Currency Prediction

Given: 7 currency features to predict RUB.

Problem: Very different magnitudes:

- USD, EUR, GBP, CHF: $\sim 10^4$
- TJS: ~ 1
- RUB (output): $\sim 10^{-4}$

Solution:

1. Activate normalisation
2. Architecture: $(7, 16, 8, 1)$
3. Activations: ReLU, ReLU, Sigmoid
4. Learning rate: 0.001 (low for stability)
5. Epochs: 3 000–5 000
6. Batch size: 32

13 Troubleshooting

13.1 Training Does Not Converge

Symptoms: Loss remains constant or oscillates wildly

Causes and solutions:

- Learning rate too high ⇒ Reduce to 0.001 or 0.0001
- Data not normalised ⇒ Activate normalisation
- Network too small ⇒ Add more neurons or layers
- Vanishing gradients ⇒ Use ReLU instead of sigmoid

13.2 Overfitting

Symptoms: Training loss low, but poor generalisation

Solutions:

- Collect more training data
- Use a smaller network
- Stop earlier (fewer epochs)
- Add regularisation (future version)

13.3 Underfitting

Symptoms: High training loss, network does not learn
Solutions:

- Larger network (more layers/neurons)
- Train for more epochs
- Increase learning rate
- More complex activation functions

13.4 Numerical Instability

Symptoms: NaN or Infinity in loss/weights
Solutions:

- Normalise data unconditionally
- Strongly reduce learning rate
- Implement gradient clipping
- For BCE: keep sigmoid outputs away from 0/1

14 Mathematical Proofs

14.1 Universal Approximation Theorem

Theorem 14.1 (Cybenko, 1989; Hornik et al., 1989)**.** A feedforward network with one hidden layer and finitely many neurons can approximate any continuous function on a compact subset of $\mathbb{R}^n$ arbitrarily well, provided the activation function is nonlinear, bounded and monotonically increasing.

Formally: For every continuous function $f : K \to \mathbb{R}$ on compact $K \subset \mathbb{R}^n$ and $\epsilon > 0$, there exists a network F with:

$$\sup_{x \in K} |f(x) - F(x)| < \epsilon \tag{48}$$

14.2 Gradient Descent Convergence

Theorem 14.2 (Convergence for convex loss function)**.** For an L-Lipschitz-continuous, convex function $\mathcal{L}$, gradient descent with constant learning rate $\eta \leq \frac{1}{L}$ converges to a global minimum:

$$\lim_{t \to \infty} \mathcal{L}(\mathbf{w}_t) = \mathcal{L}^* \tag{49}$$

Note: For neural networks the loss function is generally not convex, so only convergence to local minima is guaranteed.

15 Implementation Details

15.1 Matrix Operations

The implementation uses a matrix class with the following operations:

Matrix multiplication:

$$(\mathbf{AB})_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \tag{50}$$

Complexity: $\mathcal{O}(n^3)$

Transpose:

$$(\mathbf{A}^T)_{ij} = a_{ji} \tag{51}$$

Hadamard product:

$$(\mathbf{A} \odot \mathbf{B})_{ij} = a_{ij} \cdot b_{ij} \tag{52}$$

15.2 Complexity Analysis

Forward pass:

$$\mathcal{O}\left(\sum_{l=1}^{L-1} n_l \cdot n_{l-1}\right) = \mathcal{O}(W) \tag{53}$$

where W is the total number of weights.

Backward pass: $\mathcal{O}(W)$

Training (one epoch): $\mathcal{O}(N \cdot W)$, where N is the number of training samples.

16 Visualisations

16.1 Network Graph

The visualisation shows:

- **Neurons:** Circles, colour-coded by layer
- **Weights:** Lines between neurons
- **Colour:** Red = positive weights, Blue = negative weights
- **Thickness:** Proportional to the magnitude of the weight

16.2 Loss Curve

The loss curve $\mathcal{L}(t)$ shows:

- Y-axis: Loss value
- X-axis: Epoch number
- Ideal: Monotonically decreasing and converging

Interpretation:

- Steep descent at the start: Good learning
- Plateau: Convergence reached
- Oscillation: Learning rate too high
- Rising: Overfitting or numerical problems

16.3 Weight Distribution

Histogram of all weights:

- Well trained: Normally distributed around 0
- Problems: Very small or very large weights
- Dead neurons: Many weights exactly 0

17 Export Formats

17.1 JSON Format for Weights

```
{
  "architecture": {
    "layers": [7, 16, 8, 1],
    "activations": ["relu", "relu", "sigmoid"],
    "learningRate": 0.001,
    "lossFunction": "mse"
  },
  "weights": [
    [[w_11, w_12, ...], ...],  // Layer 1
    [[w_11, w_12, ...], ...]   // Layer 2
  ],
  "biases": [
    [[b_1], [b_2], ...],       // Layer 1
    [[b_1], [b_2], ...]        // Layer 2
  ]
}
```

17.2 CSV Format for Training Data

```
Epoch,Loss
1,0.524892
2,0.487321
3,0.456789
...
```

18 Future Extensions

18.1 Planned Features

- **Dropout:** Regularisation by randomly deactivating neurons
- **Momentum:** Accelerated convergence
- **Adam Optimizer:** Adaptive learning rates
- **Batch Normalisation:** More stable training
- **Multi-output:** Multiple output neurons
- **Convolutional Layers:** For image data

- **Recurrent Layers:** For sequence data
- **Cross-Validation:** Better evaluation
- **Early Stopping:** Automatic stopping on overfitting
- **Learning Rate Scheduling:** Adjustment during training

18.2 Dropout

$$\tilde{a}_j^{(l)} = \begin{cases} 0 & \text{with probability } p \\ \dfrac{a_j^{(l)}}{1-p} & \text{with probability } 1-p \end{cases} \tag{54}$$

18.3 Momentum

$$v_t = \gamma v_{t-1} + \eta \nabla_{\mathbf{W}} \mathcal{L} \tag{55}$$
$$\mathbf{W}_t = \mathbf{W}_{t-1} - v_t \tag{56}$$

where $\gamma \in [0, 1]$ is the momentum parameter (typically: $\gamma = 0.9$).

18.4 Adam Optimizer

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla_{\mathbf{W}} \mathcal{L} \tag{57}$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla_{\mathbf{W}} \mathcal{L})^2 \tag{58}$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \tag{59}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{60}$$
$$\mathbf{W}_t = \mathbf{W}_{t-1} - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{61}$$

Typical values: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$

19 References and Further Reading

19.1 Foundational Works

1. Rosenblatt, F. (1958). "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain"
2. Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). "Learning representations by back-propagating errors"
3. Cybenko, G. (1989). "Approximation by superpositions of a sigmoidal function"
4. Hornik, K., Stinchcombe, M., & White, H. (1989). "Multilayer feedforward networks are universal approximators"
5. LeCun, Y., Bengio, Y., & Hinton, G. (2015). "Deep learning"

19.2 Textbooks

1. Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning.* MIT Press.
2. Nielsen, M. A. (2015). *Neural Networks and Deep Learning.* Determination Press.
3. Bishop, C. M. (2006). *Pattern Recognition and Machine Learning.* Springer.

20 Appendix

20.1 Formula Reference

Activation functions:

$$\sigma(z) = \frac{1}{1+e^{-z}}, \quad \sigma'(z) = \sigma(z)(1-\sigma(z)) \tag{62}$$

$$\mathrm{ReLU}(z) = \max(0, z), \quad \mathrm{ReLU}'(z) = \mathbb{1}_{z>0} \tag{63}$$

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}, \quad \tanh'(z) = 1 - \tanh^2(z) \tag{64}$$

Loss functions:

$$\mathcal{L}_{\mathrm{MSE}} = \tfrac{1}{2}(y-\hat{y})^2, \quad \frac{\partial \mathcal{L}_{\mathrm{MSE}}}{\partial \hat{y}} = \hat{y} - y \tag{65}$$

$$\mathcal{L}_{\mathrm{BCE}} = -[y \log \hat{y} + (1-y)\log(1-\hat{y})], \quad \frac{\partial \mathcal{L}_{\mathrm{BCE}}}{\partial \hat{y}} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})} \tag{66}$$

Backpropagation:

$$\boldsymbol{\delta}^{(L-1)} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(L-1)}} \odot f'^{(L-1)}(\mathbf{z}^{(L-1)}) \tag{67}$$

$$\boldsymbol{\delta}^{(l)} = [(\mathbf{W}^{(l+1)})^T \boldsymbol{\delta}^{(l+1)}] \odot f'^{(l)}(\mathbf{z}^{(l)}) \tag{68}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} = \boldsymbol{\delta}^{(l)} (\mathbf{a}^{(l-1)})^T \tag{69}$$

Gradient descent:

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} \tag{70}$$

Normalisation:

$$x_{\mathrm{norm}} = \frac{x - x_{\min}}{x_{\max} - x_{\min}} \tag{71}$$

$$x_{\mathrm{orig}} = x_{\mathrm{norm}} \cdot (x_{\max} - x_{\min}) + x_{\min} \tag{72}$$

20.2 Glossary

Activation function Nonlinear function applied to pre-activations

Backpropagation Algorithm for computing gradients

Batch Subset of the training dataset

Bias Additive parameter in neurons (threshold)

Epoch One complete pass through the training dataset

Forward pass Computation of the network output for an input

Gradient Vector of partial derivatives

Hidden layer Layer between input and output

Learning rate Step size for parameter updates

Loss function Function measuring the error

Neuron Computational unit in the network

Overfitting Model adapts too closely to training data

Weight Parameter weighting connections between neurons

20.3 Notation

Symbol	Meaning
L	Number of layers
n_l	Number of neurons in layer l
$w_{ji}^{(l)}$	Weight from neuron i (layer $l-1$) to j (layer l)
$b_j^{(l)}$	Bias of neuron j in layer l
$z_j^{(l)}$	Pre-activation of neuron j in layer l
$a_j^{(l)}$	Activation of neuron j in layer l
$\mathbf{W}^{(l)}$	Weight matrix for layer l
$\mathbf{b}^{(l)}$	Bias vector for layer l
$f^{(l)}$	Activation function in layer l
$\mathcal{L}$	Loss function
η	Learning rate
$\delta_j^{(l)}$	Error signal of neuron j in layer l
$\nabla_{\mathbf{W}}\mathcal{L}$	Gradient of the loss w.r.t. weights
$\odot$	Hadamard product (element-wise multiplication)
$\mathbb{1}$	Indicator function

21 Closing Remarks

This manual provides a comprehensive mathematical and practical introduction to multi-layer perceptrons. The implementation in the calculator follows the algorithms described here and enables the theory to be applied directly in practice.

For questions, improvement suggestions or bug reports, please contact the developers or use the feedback system in the calculator.

Good luck experimenting with neural networks!