Deep Learning - Übungsblatt 2

Multi-Layer Perceptrons und Backpropagation

Fachhochschule Südwestfalen

23. Oktober 2025

Voraussetzungen

- Übungsblatt 1 sollte erfolgreich bearbeitet worden sein
- Grundkenntnisse in Python und NumPy
- Verständnis von Matrixoperationen und partiellen Ableitungen

1 Backpropagation-Algorithmus

1.1 Mathematische Herleitung

Aufgabe 1.1: Gegeben sei ein 3-Schicht-MLP für binäre Klassifikation:
Architektur:

$$\text{Input Layer:} \quad \mathbf{x} \in \mathbb{R}^2 \tag{1}$$

$$\text{Hidden Layer 1:} \quad \mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}, \quad \mathbf{a}^{(1)} = \sigma(\mathbf{z}^{(1)}) \in \mathbb{R}^3 \tag{2}$$

$$\text{Hidden Layer 2:} \quad \mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)}, \quad \mathbf{a}^{(2)} = \sigma(\mathbf{z}^{(2)}) \in \mathbb{R}^2 \tag{3}$$

$$\text{Output Layer:} \quad z^{(3)} = \mathbf{w}^{(3)T}\mathbf{a}^{(2)} + b^{(3)}, \quad \hat{y} = \sigma(z^{(3)}) \in \mathbb{R} \tag{4}$$

Verlustfunktion: $L = -[y \log(\hat{y}) + (1-y)\log(1-\hat{y})]$ (Binary Cross-Entropy)

(a) Berechnen Sie $\frac{\partial L}{\partial z^{(3)}}$ (zeigen Sie, dass $\frac{\partial L}{\partial z^{(3)}} = \hat{y} - y$)

(b) Berechnen Sie die Gradienten für die Ausgabeschicht:

- $\frac{\partial L}{\partial \mathbf{w}^{(3)}}$
- $\frac{\partial L}{\partial b^{(3)}}$

(c) Berechnen Sie $\frac{\partial L}{\partial \mathbf{a}^{(2)}}$ mittels Kettenregel

(d) Berechnen Sie die Gradienten für Hidden Layer 2:

- $\frac{\partial L}{\partial \mathbf{z}^{(2)}}$ (verwenden Sie $\frac{\partial \sigma}{\partial z} = \sigma(z)(1-\sigma(z))$)
- $\frac{\partial L}{\partial \mathbf{W}^{(2)}}$ und $\frac{\partial L}{\partial \mathbf{b}^{(2)}}$

(e) Berechnen Sie analog die Gradienten für Hidden Layer 1: $\frac{\partial L}{\partial \mathbf{W}^{(1)}}$ und $\frac{\partial L}{\partial \mathbf{b}^{(1)}}$

1.2 Numerisches Beispiel

Aufgabe 1.2: Führen Sie einen kompletten Forward- und Backward-Pass durch:
Gegeben:

$$\mathbf{x} = \begin{pmatrix} 0.5 \\ 0.8 \end{pmatrix}, \quad y = 1 \tag{5}$$

$$\mathbf{W}^{(1)} = \begin{pmatrix} 0.2 & 0.1 \\ -0.3 & 0.4 \\ 0.6 & -0.2 \end{pmatrix}, \quad \mathbf{b}^{(1)} = \begin{pmatrix} 0.1 \\ -0.2 \\ 0.3 \end{pmatrix} \tag{6}$$

$$\mathbf{W}^{(2)} = \begin{pmatrix} 0.4 & -0.1 & 0.3 \\ 0.2 & 0.5 & -0.4 \end{pmatrix}, \quad \mathbf{b}^{(2)} = \begin{pmatrix} 0.1 \\ -0.1 \end{pmatrix} \tag{7}$$

$$\mathbf{w}^{(3)} = \begin{pmatrix} 0.6 \\ -0.3 \end{pmatrix}, \quad b^{(3)} = 0.2 \tag{8}$$

(a) Berechnen Sie den Forward Pass: $\mathbf{z}^{(1)}, \mathbf{a}^{(1)}, \mathbf{z}^{(2)}, \mathbf{a}^{(2)}, z^{(3)}, \hat{y}$

(b) Berechnen Sie den Loss L

(c) Berechnen Sie alle Gradienten des Backward Pass

2 Implementierung eines MLP

2.1 MLP-Klasse

Aufgabe 2.1: Implementieren Sie eine vollständige MLP-Klasse:

```python
import numpy as np
import matplotlib.pyplot as plt

class MLP:
    def __init__(self, layer_sizes, activation='sigmoid',
        learning_rate=0.01):
        """
        layer_sizes: Liste mit Anzahl Neuronen pro Schicht [input,
            hidden1, hidden2, ..., output]
        activation: 'sigmoid', 'relu', oder 'tanh'
        learning_rate: Lernrate
        """
        # Ihre Implementierung hier
        pass

    def _initialize_weights(self):
        """Xavier/He-Initialisierung der Gewichte"""
        # Ihre Implementierung hier
        pass

    def _forward(self, X):
        """Forward Pass - speichert Zwischenergebnisse fuer Backprop"
            ""
        # Ihre Implementierung hier
```

```
        pass

    def _backward(self, X, y):
        """Backward Pass - berechnet Gradienten"""
        # Ihre Implementierung hier
        pass

    def train(self, X, y, epochs=1000, batch_size=None, verbose=False
        ):
        """Training mit Mini-Batch Gradient Descent"""
        # Ihre Implementierung hier
        pass

    def predict(self, X):
        """Vorhersagen fuer neue Daten"""
        # Ihre Implementierung hier
        pass

    def score(self, X, y):
        """Accuracy fuer Klassifikation"""
        # Ihre Implementierung hier
        pass
```

Bewertungskriterien:

- Korrekte Gewichtsinitialisierung
- Forward Pass Implementation
- Backward Pass Implementation
- Training Loop mit Mini-Batches
- Predict und Score Methoden
- Code-Qualität und Dokumentation

2.2 Experimentelle Evaluierung

Aufgabe 2.2: Testen Sie Ihr MLP auf verschiedenen Datensätzen:

(a) **XOR-Problem:** Trainieren Sie ein MLP mit [2, 4, 1] Architektur

- Plotten Sie den Loss über die Epochen
- Visualisieren Sie die Entscheidungsgrenze
- Erreichen Sie 100% Accuracy

(b) **Spiralen-Datensatz:** Erstellen Sie einen 2D-Spiralen-Datensatz und klassifizieren Sie ihn

```python
def make_spirals(n_samples=200, noise=0.1):
    """Erstellt 2D-Spiralen-Datensatz"""
    t = np.linspace(0, 4*np.pi, n_samples//2)
    x1 = t * np.cos(t) + noise * np.random.randn(n_samples//2)
    y1 = t * np.sin(t) + noise * np.random.randn(n_samples//2)
    x2 = -t * np.cos(t) + noise * np.random.randn(n_samples//2)
    y2 = -t * np.sin(t) + noise * np.random.randn(n_samples//2)

    X = np.vstack([np.column_stack([x1, y1]), np.column_stack([x2,
        y2])])
    y = np.hstack([np.zeros(n_samples//2), np.ones(n_samples//2)])
    return X, y
```

(c) **Hyperparameter-Tuning:** Experimentieren Sie mit verschiedenen Architekturen und Hyperparametern

- Anzahl versteckter Schichten: [2, 8], [2, 16, 8], [2, 32, 16, 8]
- Lernraten: 0.001, 0.01, 0.1
- Aktivierungsfunktionen: sigmoid, relu, tanh
- Dokumentieren Sie die besten Ergebnisse

3 Vertiefende Fragen

Aufgabe 3.1: Theoretische Analyse

(a) **Vanishing Gradient Problem:**

- Erklären Sie, warum tiefe Netzwerke mit Sigmoid-Aktivierung Probleme beim Training haben
- Berechnen Sie die maximale Ableitung der Sigmoid-Funktion
- Wie löst ReLU dieses Problem?

(b) **Initialisierung:** Erklären Sie Xavier- und He-Initialisierung mathematisch. Warum ist zufällige Initialisierung wichtig?

Aufgabe 3.2: Praktische Probleme

(a) **Overfitting:**

- Erstellen Sie einen kleinen Datensatz (50 Samples) und trainieren Sie ein überparametrisiertes Netzwerk
- Implementieren Sie Early Stopping
- Vergleichen Sie Training- und Validation-Loss

(b) **Learning Rate:** Experimentieren Sie mit verschiedenen Lernraten und dokumentieren Sie den Einfluss auf das Training

4 Bonusaufgaben (15)

Aufgabe 4.1: Erweiterte Implementierungen

(a) **Momentum:** Erweitern Sie Ihr MLP um Momentum-basierte Optimierung

$$v_t = \beta v_{t-1} + (1 - \beta)\nabla_\theta L(\theta_{t-1}) \tag{9}$$

$$\theta_t = \theta_{t-1} - \alpha v_t \tag{10}$$

(b) **Regularisierung:** Implementieren Sie L2-Regularisierung

$$L_{reg} = L + \lambda \sum_l ||\mathbf{W}^{(l)}||_2^2 \tag{11}$$

(c) **Adaptive Learning Rates:** Implementieren Sie RMSprop oder Adam

Abgabehinweise

- Abgabe als Jupyter Notebook (.ipynb) oder Python-Skript mit separatem PDF-Report
- Alle Plots und numerischen Ergebnisse dokumentieren
- Code muss reproduzierbar sein (feste Random Seeds)
- Mathematische Herleitungen vollständig ausschreiben
- Diskussion der Ergebnisse und Beobachtungen

Bewertungsschema

Aufgabenbereich	**Punkte**
Backpropagation-Algorithmus	30
MLP-Implementierung	35
Vertiefende Fragen	20
Gesamt	**85**
Bonusaufgaben	+15

Tipps für die Implementierung

- Verwenden Sie `np.random.seed()` für reproduzierbare Ergebnisse
- Implementieren Sie numerische Gradientenprüfung zur Verifikation
- Starten Sie mit einfachen Problemen (XOR) vor komplexeren Datensätzen
- Visualisieren Sie Zwischenergebnisse für Debugging
- Nutzen Sie Vektorisierung für Effizienz