Deep Learning - Musterlösung Übung 1

Mathematische Grundlagen und Einführung

Fachhochschule Südwestfalen

23. Oktober 2025

Hinweise zur Musterlösung

Diese Musterlösung bietet detaillierte Herleitungen und vollständige Implementierungen. Alternative Lösungsansätze sind oft ebenfalls korrekt.

1 Mathematische Grundlagen - Lösungen

1.1 Aufgabe 1.1: Lineare Algebra

Gegeben:

$$\mathbf{A} = \begin{pmatrix} 2 & 1 & -1 \\ 0 & 3 & 2 \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} 1 & 0 \\ -1 & 2 \\ 3 & 1 \end{pmatrix} \tag{1}$$

(a) Berechnung von $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$:
Dimensionsprüfung: $\mathbf{A}_{2\times3} \cdot \mathbf{B}_{3\times2} = \mathbf{C}_{2\times2}$

$$c_{11} = 2 \cdot 1 + 1 \cdot (-1) + (-1) \cdot 3 = 2 - 1 - 3 = -2 \tag{2}$$
$$c_{12} = 2 \cdot 0 + 1 \cdot 2 + (-1) \cdot 1 = 0 + 2 - 1 = 1 \tag{3}$$
$$c_{21} = 0 \cdot 1 + 3 \cdot (-1) + 2 \cdot 3 = 0 - 3 + 6 = 3 \tag{4}$$
$$c_{22} = 0 \cdot 0 + 3 \cdot 2 + 2 \cdot 1 = 0 + 6 + 2 = 8 \tag{5}$$

Ergebnis:

$$\mathbf{C} = \mathbf{A} \cdot \mathbf{B} = \begin{pmatrix} -2 & 1 \\ 3 & 8 \end{pmatrix} \tag{6}$$

(b) Verifikation von $(\mathbf{A} \cdot \mathbf{B})^T = \mathbf{B}^T \cdot \mathbf{A}^T$:

$$\mathbf{A}^T = \begin{pmatrix} 2 & 0 \\ 1 & 3 \\ -1 & 2 \end{pmatrix}, \quad \mathbf{B}^T = \begin{pmatrix} 1 & -1 & 3 \\ 0 & 2 & 1 \end{pmatrix} \tag{7}$$

Links: $(\mathbf{A} \cdot \mathbf{B})^T = \begin{pmatrix} -2 & 3 \\ 1 & 8 \end{pmatrix}$

Rechts: $\mathbf{B}^T \cdot \mathbf{A}^T$:

$$(\mathbf{B}^T \cdot \mathbf{A}^T)_{11} = 1 \cdot 2 + (-1) \cdot 1 + 3 \cdot (-1) = 2 - 1 - 3 = -2 \quad (8)$$
$$(\mathbf{B}^T \cdot \mathbf{A}^T)_{12} = 1 \cdot 0 + (-1) \cdot 3 + 3 \cdot 2 = 0 - 3 + 6 = 3 \quad (9)$$
$$(\mathbf{B}^T \cdot \mathbf{A}^T)_{21} = 0 \cdot 2 + 2 \cdot 1 + 1 \cdot (-1) = 0 + 2 - 1 = 1 \quad (10)$$
$$(\mathbf{B}^T \cdot \mathbf{A}^T)_{22} = 0 \cdot 0 + 2 \cdot 3 + 1 \cdot 2 = 0 + 6 + 2 = 8 \quad (11)$$

$\mathbf{B}^T \cdot \mathbf{A}^T = \begin{pmatrix} -2 & 3 \\ 1 & 8 \end{pmatrix}$

(c) Interpretation: Eine 2×3 Matrix kann 3 Eingaben auf 2 Ausgaben abbilden. In einem neuronalen Netzwerk würde dies 3 Eingangsneuronen mit 2 Neuronen in der nächsten Schicht verbinden.

1.2 Aufgabe 1.2: Aktivierungsfunktionen

(a) Sigmoid-Funktion: $\sigma(x) = \frac{1}{1+e^{-x}}$

Ableitung berechnen:

$$\begin{aligned}
\sigma'(x) &= \frac{d}{dx}\left(\frac{1}{1+e^{-x}}\right) && (12)\\
&= \frac{d}{dx}(1+e^{-x})^{-1} && (13)\\
&= -(1+e^{-x})^{-2} \cdot (-e^{-x}) && (14)\\
&= \frac{e^{-x}}{(1+e^{-x})^2} && (15)
\end{aligned}$$

Nachweis von $\sigma'(x) = \sigma(x)(1-\sigma(x))$:

$$\begin{aligned}
\sigma(x)(1-\sigma(x)) &= \frac{1}{1+e^{-x}} \cdot \left(1 - \frac{1}{1+e^{-x}}\right) && (16)\\
&= \frac{1}{1+e^{-x}} \cdot \frac{1+e^{-x}-1}{1+e^{-x}} && (17)\\
&= \frac{1}{1+e^{-x}} \cdot \frac{e^{-x}}{1+e^{-x}} && (18)\\
&= \frac{e^{-x}}{(1+e^{-x})^2} = \sigma'(x) && (19)
\end{aligned}$$

Werte berechnen:

$$\begin{aligned}
\sigma(0) &= \frac{1}{1+e^0} = \frac{1}{2} = 0.5 && (20)\\
\sigma(2) &= \frac{1}{1+e^{-2}} \approx \frac{1}{1+0.135} \approx 0.881 && (21)\\
\sigma(-2) &= \frac{1}{1+e^{2}} \approx \frac{1}{1+7.389} \approx 0.119 && (22)
\end{aligned}$$

(b) ReLU-Funktion: ReLU$(x) = \max(0, x)$

Ableitung:

$$\text{ReLU}'(x) = \begin{cases} 0 & \text{wenn } x < 0 \\ \text{undefiniert} & \text{wenn } x = 0 \\ 1 & \text{wenn } x > 0 \end{cases} \quad (23)$$

In der Praxis setzt man $\text{ReLU}'(0) = 0$ oder $\text{ReLU}'(0) = 1$.

Dying ReLU Problem: Wenn die Eingaben eines ReLU-Neurons immer negativ sind, ist die Ausgabe konstant 0 und der Gradient ist 0. Das Neuron kann nicht mehr lernen und ist "tot".

2 Grundlagen Neuronaler Netze - Lösungen

2.1 Aufgabe 2.1: Einfaches Neuron

Gegeben: $\mathbf{x} = (2, 3)^T$, $\mathbf{w} = (0.5, -0.2)^T$, $b = 0.1$

(a) Netzausgabe ohne Aktivierungsfunktion:

$$z = \mathbf{w}^T\mathbf{x} + b = 0.5 \cdot 2 + (-0.2) \cdot 3 + 0.1 = 1 - 0.6 + 0.1 = 0.5 \tag{24}$$

(b) Mit Sigmoid-Aktivierung:

$$y = \sigma(z) = \sigma(0.5) = \frac{1}{1 + e^{-0.5}} \approx \frac{1}{1 + 0.607} \approx 0.622 \tag{25}$$

(c) Mit ReLU-Aktivierung:

$$y = \text{ReLU}(z) = \text{ReLU}(0.5) = \max(0, 0.5) = 0.5 \tag{26}$$

2.2 Aufgabe 2.2: XOR-Problem

Warum kann ein einschichtiges Perceptron XOR nicht lösen?

Das XOR-Problem ist nicht linear separierbar. Ein einschichtiges Perceptron kann nur lineare Entscheidungsgrenzen lernen.

XOR-Wahrheitstabelle:

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

Lösung mit mehrschichtigem Netz:

Hidden Layer implementiert: - Neuron 1: NAND-Gate: $z_1 = -x_1 - x_2 + 1.5$ - Neuron 2: OR-Gate: $z_2 = x_1 + x_2 - 0.5$

Output Layer: - AND der beiden Hidden-Neuronen: $y = \sigma(z_1 + z_2 - 1.5)$

3 Programmieraufgaben - Lösungen

3.1 Aufgabe 3.1: Aktivierungsfunktionen implementieren

Listing 1: Aktivierungsfunktionen mit NumPy

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

def sigmoid(x):
```

```
    """Numerisch stabile Sigmoid-Implementierung"""
    return np.where(x >= 0,
                    1 / (1 + np.exp(-x)),
                    np.exp(x) / (1 + np.exp(x)))

def sigmoid_derivative(x):
    """Ableitung der Sigmoid-Funktion"""
    s = sigmoid(x)
    return s * (1 - s)

def relu(x):
    """ReLU-Aktivierungsfunktion"""
    return np.maximum(0, x)

def relu_derivative(x):
    """Ableitung der ReLU-Funktion"""
    return (x > 0).astype(float)

# Visualisierung
x = np.linspace(-10, 10, 1000)

plt.figure(figsize=(12, 8))

# Sigmoid
plt.subplot(2, 2, 1)
plt.plot(x, sigmoid(x), 'b-', linewidth=2, label='Sigmoid')
plt.title('Sigmoid-Funktion')
plt.grid(True)
plt.legend()

plt.subplot(2, 2, 2)
plt.plot(x, sigmoid_derivative(x), 'r-', linewidth=2, label="Sigmoid'
  ")
plt.title('Sigmoid-Ableitung')
plt.grid(True)
plt.legend()

# ReLU
plt.subplot(2, 2, 3)
plt.plot(x, relu(x), 'g-', linewidth=2, label='ReLU')
plt.title('ReLU-Funktion')
plt.grid(True)
plt.legend()

plt.subplot(2, 2, 4)
plt.plot(x, relu_derivative(x), 'm-', linewidth=2, label="ReLU'")
plt.title('ReLU-Ableitung')
plt.grid(True)
plt.legend()

plt.tight_layout()
```

```python
plt.show()

# Verifikation
print(f"sigmoid(0) = {sigmoid(0):.3f} (sollte 0.5 sein)")
print(f"relu(-1) = {relu(-1):.3f} (sollte 0.0 sein)")
print(f"relu(2) = {relu(2):.3f} (sollte 2.0 sein)")
```

3.2 Aufgabe 3.2: Perceptron für AND-Gate

Listing 2: Perceptron-Implementierung

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

class Perceptron:
    def __init__(self, learning_rate=0.1):
        self.learning_rate = learning_rate
        self.weights = None
        self.bias = None

    def fit(self, X, y, epochs=100):
        n_samples, n_features = X.shape

        # Initialisierung
        self.weights = np.random.randn(n_features) * 0.01
        self.bias = 0

        self.errors = []

        for epoch in range(epochs):
            total_error = 0
            for xi, target in zip(X, y):
                # Forward pass
                linear_output = np.dot(xi, self.weights) + self.bias
                prediction = self.activation_function(linear_output)

                # Update
                error = target - prediction
                self.weights += self.learning_rate * error * xi
                self.bias += self.learning_rate * error

                total_error += abs(error)

            self.errors.append(total_error)

            if total_error == 0:
                print(f"Konvergiert nach {epoch + 1} Epochen")
                break

    def activation_function(self, x):
        return np.where(x >= 0, 1, 0)
```

```python

    def predict(self, X):
        linear_output = np.dot(X, self.weights) + self.bias
        return self.activation_function(linear_output)

    def plot_decision_boundary(self, X, y):
        plt.figure(figsize=(10, 4))

        # Plot 1: Datenpunkte und Entscheidungsgrenze
        plt.subplot(1, 2, 1)
        colors = ['red', 'blue']
        for i in range(2):
            plt.scatter(X[y == i, 0], X[y == i, 1],
                        c=colors[i], label=f'Klasse {i}')

        # Entscheidungsgrenze
        if self.weights[1] != 0:
            x_line = np.linspace(-0.5, 1.5, 100)
            y_line = -(self.weights[0] * x_line + self.bias) / self.weights[1]
            plt.plot(x_line, y_line, 'k--', label='Entscheidungsgrenze')

        plt.xlabel('x1')
        plt.ylabel('x2')
        plt.title('AND-Gate Klassifikation')
        plt.legend()
        plt.grid(True)

        # Plot 2: Fehlerentwicklung
        plt.subplot(1, 2, 2)
        plt.plot(self.errors)
        plt.xlabel('Epoche')
        plt.ylabel('Gesamtfehler')
        plt.title('Konvergenz')
        plt.grid(True)

        plt.tight_layout()
        plt.show()

# AND-Gate Daten
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 0, 0, 1])  # AND-Gate Ausgabe

# Training
perceptron = Perceptron(learning_rate=0.1)
perceptron.fit(X, y, epochs=100)

# Test
predictions = perceptron.predict(X)
print("AND-Gate Vorhersagen:")
```

```python
for i in range(len(X)):
    print(f"Input: {X[i]}, Target: {y[i]}, Prediction: {predictions[i]}")

print(f"\nFinale Gewichte: {perceptron.weights}")
print(f"Finaler Bias: {perceptron.bias}")

# Visualisierung
perceptron.plot_decision_boundary(X, y)
```

4 Verständnisfragen - Lösungen

4.1 Aufgabe 4.1: Theoretische Fragen

(a) Warum brauchen neuronale Netze Aktivierungsfunktionen?

Ohne Aktivierungsfunktionen wäre ein mehrschichtiges Netz nur eine Komposition linearer Transformationen, was wiederum eine lineare Transformation wäre. Aktivierungsfunktionen führen Nichtlinearität ein, wodurch komplexe Mappings gelernt werden können.

(b) Universeller Approximationssatz:

Ein Feed-Forward-Netzwerk mit einer versteckten Schicht und ausreichend vielen Neuronen kann jede stetige Funktion auf einem kompakten Bereich beliebig genau approximieren. Dies garantiert jedoch nicht: - Effiziente Lernbarkeit - Kleine Netzwerkgröße - Gute Generalisierung

(c) Vanishing Gradient Problem:

Bei tiefen Netzen werden Gradienten durch wiederholte Multiplikation mit Gewichten und Ableitungen der Aktivierungsfunktionen exponentiell kleiner. Besonders problematisch bei Sigmoid/Tanh-Funktionen, deren Ableitungen maximal 0.25 bzw. 1 sind.

Lösungsansätze: - ReLU-Aktivierungsfunktionen - Bessere Gewichtsinitialisierung (Xavier, He) - Batch Normalization - Residual Connections - LSTM/GRU für RNNs

5 Zusätzliche Implementierungen

5.1 Erweiterte Aktivierungsfunktionen

Listing 3: Weitere Aktivierungsfunktionen

```python
def tanh(x):
    """Tangens Hyperbolicus"""
    return np.tanh(x)

def tanh_derivative(x):
    """Ableitung von Tanh"""
    return 1 - np.tanh(x)**2

def leaky_relu(x, alpha=0.01):
    """Leaky ReLU mit kleiner Steigung für negative Werte"""
    return np.where(x > 0, x, alpha * x)
```

```

def leaky_relu_derivative(x, alpha=0.01):
    """Ableitung von Leaky ReLU"""
    return np.where(x > 0, 1, alpha)

def softmax(x):
    """Softmax für Multiclass-Klassifikation"""
    exp_x = np.exp(x - np.max(x))  # Numerische Stabilität
    return exp_x / np.sum(exp_x)

# Beispiel für verschiedene Aktivierungsfunktionen
x = np.linspace(-5, 5, 100)

plt.figure(figsize=(15, 10))

functions = [
    (sigmoid, "Sigmoid"),
    (tanh, "Tanh"),
    (relu, "ReLU"),
    (lambda x: leaky_relu(x, 0.1), "Leaky ReLU (=0.1)")
]

for i, (func, name) in enumerate(functions):
    plt.subplot(2, 2, i+1)
    plt.plot(x, func(x), linewidth=2)
    plt.title(name)
    plt.grid(True)
    plt.xlabel('x')
    plt.ylabel('f(x)')

plt.tight_layout()
plt.show()
```

6 Zusammenfassung und Ausblick

6.1 Wichtige Erkenntnisse

- **Mathematische Grundlagen:** Lineare Algebra und Differentialrechnung sind fundamental
- **Aktivierungsfunktionen:** Ermöglichen nichtlineare Mappings
- **Perceptron:** Einfachstes Modell, kann nur linear separierbare Probleme lösen
- **Mehrschichtige Netze:** Können komplexe Funktionen approximieren
- **Gradientenabstieg:** Grundlegender Optimierungsalgorithmus

6.2 Ausblick auf Übung 2

- Backpropagation-Algorithmus

- Mehrschichtige Perzeptrons (MLPs)
- Kostenfunktionen und ihre Ableitungen
- Praktische Implementierung eines MLP
- Optimierungsverfahren (SGD, Adam, etc.)

6.3 Weiterführende Literatur

- **Bücher:**
  - ”Deep Learning Goodfellow, Bengio, Courville (Kapitel 2-6)
  - ”Neural Networks and Deep Learning Michael Nielsen
  - ”Pattern Recognition and Machine Learning Christopher Bishop
- **Online-Ressourcen:**
  - 3Blue1Brown: ”Neural NetworksSSerie
  - CS231n: Convolutional Neural Networks for Visual Recognition
  - Fast.ai Practical Deep Learning Course