Deep Learning - Musterlösung Übung 2

Multi-Layer Perceptrons und Backpropagation

Fachhochschule Südwestfalen

23. Oktober 2025

Hinweise zur Musterlösung

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

1 Backpropagation-Algorithmus - Lösungen

1.1 Aufgabe 1.1: Mathematische Herleitung

Gegeben ist ein 3-Schicht-MLP mit Binary Cross-Entropy Loss:

$$L = -[y \log(\hat{y}) + (1-y)\log(1-\hat{y})] \tag{1}$$

Teil (a): Gradient der Ausgabe

$$\frac{\partial L}{\partial z^{(3)}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{(3)}} \tag{2}$$

$$\frac{\partial L}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}} \tag{3}$$

$$\frac{\partial \hat{y}}{\partial z^{(3)}} = \sigma(z^{(3)})(1-\sigma(z^{(3)})) = \hat{y}(1-\hat{y}) \tag{4}$$

Einsetzen:

$$\begin{aligned}
\frac{\partial L}{\partial z^{(3)}} &= \left(-\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}\right)\hat{y}(1-\hat{y}) && (5)\\
&= -y(1-\hat{y}) + (1-y)\hat{y} && (6)\\
&= -y + y\hat{y} + \hat{y} - y\hat{y} && (7)\\
&= \hat{y} - y && (8)
\end{aligned}$$

$$\boxed{\frac{\partial L}{\partial z^{(3)}} = \hat{y} - y}$$

Teil (b): Gradienten der Ausgabeschicht

$$\frac{\partial L}{\partial \mathbf{w}^{(3)}} = \frac{\partial L}{\partial z^{(3)}}\frac{\partial z^{(3)}}{\partial \mathbf{w}^{(3)}} = (\hat{y} - y)\mathbf{a}^{(2)} \tag{9}$$

$$\frac{\partial L}{\partial b^{(3)}} = \frac{\partial L}{\partial z^{(3)}}\frac{\partial z^{(3)}}{\partial b^{(3)}} = \hat{y} - y \tag{10}$$

Teil (c): Gradient für vorherige Schicht

$$\frac{\partial L}{\partial \mathbf{a}^{(2)}} = \frac{\partial L}{\partial z^{(3)}}\frac{\partial z^{(3)}}{\partial \mathbf{a}^{(2)}} = (\hat{y} - y)\mathbf{w}^{(3)} \tag{11}$$

Teil (d): Gradienten für Hidden Layer 2

$$\begin{align} \frac{\partial L}{\partial \mathbf{z}^{(2)}} &= \frac{\partial L}{\partial \mathbf{a}^{(2)}} \odot \frac{\partial \mathbf{a}^{(2)}}{\partial \mathbf{z}^{(2)}} \tag{12} \\ &= (\hat{y} - y)\mathbf{w}^{(3)} \odot \mathbf{a}^{(2)} \odot (1 - \mathbf{a}^{(2)}) \tag{13} \end{align}$$

Definiere $\boldsymbol{\delta}^{(2)} = \frac{\partial L}{\partial \mathbf{z}^{(2)}}$:

$$\frac{\partial L}{\partial \mathbf{W}^{(2)}} = \boldsymbol{\delta}^{(2)}(\mathbf{a}^{(1)})^T \tag{14}$$

$$\frac{\partial L}{\partial \mathbf{b}^{(2)}} = \boldsymbol{\delta}^{(2)} \tag{15}$$

Teil (e): Gradienten für Hidden Layer 1

$$\frac{\partial L}{\partial \mathbf{a}^{(1)}} = (\mathbf{W}^{(2)})^T\boldsymbol{\delta}^{(2)} \tag{16}$$

$$\boldsymbol{\delta}^{(1)} = \frac{\partial L}{\partial \mathbf{a}^{(1)}} \odot \mathbf{a}^{(1)} \odot (1 - \mathbf{a}^{(1)}) \tag{17}$$

$$\frac{\partial L}{\partial \mathbf{W}^{(1)}} = \boldsymbol{\delta}^{(1)}\mathbf{x}^T \tag{18}$$

$$\frac{\partial L}{\partial \mathbf{b}^{(1)}} = \boldsymbol{\delta}^{(1)} \tag{19}$$

1.2 Aufgabe 1.2: Numerisches Beispiel

Gegeben:

$$\mathbf{x} = \begin{pmatrix} 0.5 \\ 0.8 \end{pmatrix}, \quad y = 1 \tag{20}$$

$$\mathbf{W}^{(1)} = \begin{pmatrix} 0.2 & 0.1 \\ -0.3 & 0.4 \\ 0.6 & -0.2 \end{pmatrix}, \quad \mathbf{b}^{(1)} = \begin{pmatrix} 0.1 \\ -0.2 \\ 0.3 \end{pmatrix} \tag{21}$$

Forward Pass:

Layer 1:

$$\begin{aligned}
\mathbf{z}^{(1)} &= \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} && (22)\\
&= \begin{pmatrix} 0.2 & 0.1 \\ -0.3 & 0.4 \\ 0.6 & -0.2 \end{pmatrix} \begin{pmatrix} 0.5 \\ 0.8 \end{pmatrix} + \begin{pmatrix} 0.1 \\ -0.2 \\ 0.3 \end{pmatrix} && (23)\\
&= \begin{pmatrix} 0.18 \\ 0.12 \\ 0.14 \end{pmatrix} && (24)
\end{aligned}$$

$$\mathbf{a}^{(1)} = \sigma(\mathbf{z}^{(1)}) = \begin{pmatrix} 0.545 \\ 0.530 \\ 0.535 \end{pmatrix} \tag{25}$$

Layer 2:

$$\begin{aligned}
\mathbf{z}^{(2)} &= \mathbf{W}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)} && (26)\\
&= \begin{pmatrix} 0.4 & -0.1 & 0.3 \\ 0.2 & 0.5 & -0.4 \end{pmatrix} \begin{pmatrix} 0.545 \\ 0.530 \\ 0.535 \end{pmatrix} + \begin{pmatrix} 0.1 \\ -0.1 \end{pmatrix} && (27)\\
&= \begin{pmatrix} 0.419 \\ 0.195 \end{pmatrix} && (28)
\end{aligned}$$

$$\mathbf{a}^{(2)} = \sigma(\mathbf{z}^{(2)}) = \begin{pmatrix} 0.603 \\ 0.549 \end{pmatrix} \tag{29}$$

Output Layer:

$$\begin{aligned}
z^{(3)} &= \mathbf{w}^{(3)T}\mathbf{a}^{(2)} + b^{(3)} && (30)\\
&= 0.6 \cdot 0.603 + (-0.3) \cdot 0.549 + 0.2 && (31)\\
&= 0.397 && (32)\\
\hat{y} &= \sigma(0.397) = 0.598 && (33)
\end{aligned}$$

Loss:

$$L = -[1 \cdot \log(0.598) + 0 \cdot \log(0.402)] = -\log(0.598) = 0.515 \tag{34}$$

Backward Pass:
Output Layer:

$$\begin{aligned}
\frac{\partial L}{\partial z^{(3)}} &= 0.598 - 1 = -0.402 && (35)\\
\frac{\partial L}{\partial \mathbf{w}^{(3)}} &= -0.402 \begin{pmatrix} 0.603 \\ 0.549 \end{pmatrix} = \begin{pmatrix} -0.242 \\ -0.221 \end{pmatrix} && (36)\\
\frac{\partial L}{\partial b^{(3)}} &= -0.402 && (37)
\end{aligned}$$

Hidden Layer 2:

$$\frac{\partial L}{\partial \mathbf{a}^{(2)}} = -0.402 \begin{pmatrix} 0.6 \\ -0.3 \end{pmatrix} = \begin{pmatrix} -0.241 \\ 0.121 \end{pmatrix} \tag{38}$$

$$\boldsymbol{\delta}^{(2)} = \begin{pmatrix} -0.241 \\ 0.121 \end{pmatrix} \odot \begin{pmatrix} 0.603 \cdot 0.397 \\ 0.549 \cdot 0.451 \end{pmatrix} = \begin{pmatrix} -0.058 \\ 0.030 \end{pmatrix} \tag{39}$$

Hidden Layer 1:

$$\frac{\partial L}{\partial \mathbf{a}^{(1)}} = \begin{pmatrix} 0.4 & 0.2 \\ -0.1 & 0.5 \\ 0.3 & -0.4 \end{pmatrix} \begin{pmatrix} -0.058 \\ 0.030 \end{pmatrix} = \begin{pmatrix} -0.017 \\ 0.021 \\ -0.029 \end{pmatrix} \tag{40}$$

2 MLP-Implementierung - Musterlösung

2.1 Aufgabe 2.1: 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):
        self.layer_sizes = layer_sizes
        self.activation_name = activation
        self.learning_rate = learning_rate
        self.num_layers = len(layer_sizes)

        # Initialize weights and biases
        self.weights = {}
        self.biases = {}
        self._initialize_weights()

        # Store for backpropagation
        self.cache = {}
        self.losses = []

    def _initialize_weights(self):
        """Xavier/He initialization"""
        for i in range(1, self.num_layers):
            # Xavier initialization for sigmoid, He for ReLU
            if self.activation_name == 'relu':
                # He initialization
                std = np.sqrt(2.0 / self.layer_sizes[i-1])
            else:
                # Xavier initialization
                std = np.sqrt(1.0 / self.layer_sizes[i-1])

            self.weights[i] = np.random.normal(0, std,
                (self.layer_sizes[i], self.layer_sizes[i-1]))
            self.biases[i] = np.zeros((self.layer_sizes[i], 1))
```

```
    def _activation(self, z):
        """Activation function"""
        if self.activation_name == 'sigmoid':
            return self._sigmoid(z)
        elif self.activation_name == 'relu':
            return self._relu(z)
        elif self.activation_name == 'tanh':
            return np.tanh(z)
        else:
            raise ValueError(f"Unknown activation: {self.
               activation_name}")

    def _activation_derivative(self, z):
        """Derivative of activation function"""
        if self.activation_name == 'sigmoid':
            s = self._sigmoid(z)
            return s * (1 - s)
        elif self.activation_name == 'relu':
            return (z > 0).astype(float)
        elif self.activation_name == 'tanh':
            return 1 - np.tanh(z)**2
        else:
            raise ValueError(f"Unknown activation: {self.
               activation_name}")

    def _sigmoid(self, z):
        """Numerically stable sigmoid"""
        return np.where(z >= 0,
                        1 / (1 + np.exp(-z)),
                        np.exp(z) / (1 + np.exp(z)))

    def _relu(self, z):
        return np.maximum(0, z)

    def _forward(self, X):
        """Forward pass with caching for backpropagation"""
        self.cache = {}
        A = X.T  # Shape: (features, samples)
        self.cache[0] = A

        for i in range(1, self.num_layers):
            Z = self.weights[i] @ A + self.biases[i]
            if i == self.num_layers - 1:  # Output layer
                A = self._sigmoid(Z)  # Always sigmoid for binary
                   classification
            else:  # Hidden layers
                A = self._activation(Z)

            self.cache[i] = {'Z': Z, 'A': A}
            A = self.cache[i]['A']
```

```

        return A

    def _backward(self, X, y):
        """Backward pass - compute gradients"""
        m = X.shape[0]  # Number of samples
        y = y.reshape(1, -1)  # Shape: (1, samples)

        # Get final output
        A_final = self.cache[self.num_layers - 1]['A']

        # Gradients storage
        gradients = {}

        # Output layer gradient (assuming binary cross-entropy)
        dZ = A_final - y  # Shape: (1, samples)

        for i in range(self.num_layers - 1, 0, -1):
            # Get previous layer activation
            A_prev = self.cache[i-1] if i == 1 else self.cache[i-1]['
               A']

            # Compute gradients
            gradients[f'dW{i}'] = (1/m) * dZ @ A_prev.T
            gradients[f'db{i}'] = (1/m) * np.sum(dZ, axis=1, keepdims
               =True)

            if i > 1:  # Not input layer
                # Compute dZ for previous layer
                dA_prev = self.weights[i].T @ dZ
                Z_prev = self.cache[i-1]['Z']
                dZ = dA_prev * self._activation_derivative(Z_prev)

        return gradients

    def _update_parameters(self, gradients):
        """Update weights and biases using gradients"""
        for i in range(1, self.num_layers):
            self.weights[i] -= self.learning_rate * gradients[f'dW{i}
               ']
            self.biases[i] -= self.learning_rate * gradients[f'db{i}'
               ]

    def _compute_loss(self, y_true, y_pred):
        """Binary cross-entropy loss"""
        m = y_true.shape[0]
        # Clip predictions to prevent log(0)
        y_pred = np.clip(y_pred, 1e-15, 1 - 1e-15)
        loss = -(1/m) * np.sum(y_true * np.log(y_pred) +
                               (1 - y_true) * np.log(1 - y_pred))
        return loss
```

```python

    def train(self, X, y, epochs=1000, batch_size=None, verbose=False
        ):
        """Training with mini-batch gradient descent"""
        if batch_size is None:
            batch_size = X.shape[0]  # Full batch

        for epoch in range(epochs):
            # Shuffle data
            indices = np.random.permutation(X.shape[0])
            X_shuffled = X[indices]
            y_shuffled = y[indices]

            epoch_loss = 0
            num_batches = 0

            # Mini-batch training
            for i in range(0, X.shape[0], batch_size):
                X_batch = X_shuffled[i:i+batch_size]
                y_batch = y_shuffled[i:i+batch_size]

                # Forward pass
                y_pred = self._forward(X_batch)

                # Compute loss
                loss = self._compute_loss(y_batch, y_pred.T)
                epoch_loss += loss
                num_batches += 1

                # Backward pass
                gradients = self._backward(X_batch, y_batch)

                # Update parameters
                self._update_parameters(gradients)

            # Average loss for epoch
            avg_loss = epoch_loss / num_batches
            self.losses.append(avg_loss)

            if verbose and (epoch + 1) % 100 == 0:
                print(f"Epoch {epoch+1}/{epochs}, Loss: {avg_loss:.4f
                    }")

    def predict(self, X):
        """Make predictions"""
        y_pred = self._forward(X)
        return (y_pred.T > 0.5).astype(int).flatten()

    def predict_proba(self, X):
        """Predict probabilities"""
        return self._forward(X).T.flatten()
```

```python

    def score(self, X, y):
        """Compute accuracy"""
        predictions = self.predict(X)
        return np.mean(predictions == y)

    def plot_loss(self):
        """Plot training loss"""
        plt.figure(figsize=(10, 6))
        plt.plot(self.losses)
        plt.title('Training Loss')
        plt.xlabel('Epoch')
        plt.ylabel('Loss')
        plt.grid(True)
        plt.show()
```

2.2 Aufgabe 2.2: Experimentelle Evaluierung

XOR-Problem lösen:

```python
# XOR Dataset
X_xor = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_xor = np.array([0, 1, 1, 0])

# Create and train MLP
mlp_xor = MLP([2, 4, 1], activation='sigmoid', learning_rate=0.1)
mlp_xor.train(X_xor, y_xor, epochs=5000, verbose=True)

# Test predictions
print("XOR Results:")
for i in range(len(X_xor)):
    pred = mlp_xor.predict_proba(X_xor[i:i+1])[0]
    print(f"Input: {X_xor[i]}, Target: {y_xor[i]}, Prediction: {pred
        :.3f}")

# Plot decision boundary
def plot_decision_boundary(model, X, y, title="Decision Boundary"):
    plt.figure(figsize=(10, 8))

    # Create mesh
    h = 0.01
    x_min, x_max = X[:, 0].min() - 0.1, X[:, 0].max() + 0.1
    y_min, y_max = X[:, 1].min() - 0.1, X[:, 1].max() + 0.1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    # Predict on mesh
    mesh_points = np.c_[xx.ravel(), yy.ravel()]
    Z = model.predict_proba(mesh_points)
    Z = Z.reshape(xx.shape)

```

```python
    # Plot
    plt.contourf(xx, yy, Z, levels=50, alpha=0.8, cmap=plt.cm.RdYlBu)
    plt.colorbar(label='Prediction Probability')

    # Plot data points
    colors = ['blue', 'red']
    for i in range(len(X)):
        plt.scatter(X[i, 0], X[i, 1], c=colors[y[i]], s=100,
            edgecolors='black')

    plt.title(title)
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.grid(True)
    plt.show()

plot_decision_boundary(mlp_xor, X_xor, y_xor, "XOR Decision Boundary"
    )
mlp_xor.plot_loss()
```

Spiralen-Datensatz:

```python
def make_spirals(n_samples=200, noise=0.1):
    """Create spiral dataset"""
    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

# Generate spiral data
X_spiral, y_spiral = make_spirals(n_samples=400, noise=0.3)

# Normalize data
X_spiral = (X_spiral - X_spiral.mean(axis=0)) / X_spiral.std(axis=0)

# Train MLP
mlp_spiral = MLP([2, 16, 8, 1], activation='relu', learning_rate
    =0.01)
mlp_spiral.train(X_spiral, y_spiral, epochs=2000, batch_size=32,
    verbose=True)

print(f"Spiral Accuracy: {mlp_spiral.score(X_spiral, y_spiral):.3f}")
plot_decision_boundary(mlp_spiral, X_spiral, y_spiral, "Spiral
    Decision Boundary")
```

3 Vertiefende Fragen - Lösungen

3.1 Aufgabe 3.1: Vanishing Gradient Problem

Sigmoid-Problem: Die Sigmoid-Ableitung hat Maximum bei $x = 0$:

$$\sigma'(0) = \sigma(0)(1 - \sigma(0)) = 0.5 \cdot 0.5 = 0.25 \tag{41}$$

In tiefen Netzwerken werden Gradienten durch Multiplikation mit ≤ 0.25 exponentiell kleiner:

$$\frac{\partial L}{\partial W^{(1)}} \propto \prod_{i=2}^{L} W^{(i)}\sigma'(z^{(i)}) \leq (0.25)^{L-1} \tag{42}$$

ReLU-Lösung:

$$\text{ReLU}'(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases} \tag{43}$$

Für positive Aktivierungen ist der Gradient konstant 1, verhindert Vanishing.
Initialisierung:

- **Xavier:** $W \sim \mathcal{N}(0, \frac{1}{n_{in}})$ für Sigmoid/Tanh
- **He:** $W \sim \mathcal{N}(0, \frac{2}{n_{in}})$ für ReLU
- Verhindert zu große/kleine Aktivierungen

3.2 Aufgabe 3.2: Praktische Probleme

Overfitting-Demo:

```python
# Small dataset
X_small = X_xor
y_small = y_xor

# Overparameterized network
mlp_over = MLP([2, 50, 50, 1], learning_rate=0.1)

# Train with validation split
val_losses = []
train_losses = []

for epoch in range(1000):
    # Train
    mlp_over.train(X_small, y_small, epochs=1, verbose=False)

    # Track losses
    if epoch % 10 == 0:
        train_pred = mlp_over.predict_proba(X_small)
        train_loss = mlp_over._compute_loss(y_small, train_pred)
        train_losses.append(train_loss)

        # Validation on same data (for demo)
        val_losses.append(train_loss)
```

```python

# Plot overfitting
plt.figure(figsize=(10, 6))
plt.plot(range(0, 1000, 10), train_losses, label='Training Loss')
plt.plot(range(0, 1000, 10), val_losses, label='Validation Loss')
plt.title('Overfitting Demo')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```

Zusätzliche Implementierungshinweise

Numerische Stabilität

- Gradient clipping: $\nabla W = \text{clip}(\nabla W, -\theta, \theta)$
- Batch normalization vor Aktivierungen
- Learning rate scheduling: $\eta_t = \eta_0/(1 + \alpha t)$

Debugging-Techniken

- Gradient checking: Numerische vs. analytische Gradienten
- Aktivierung-Monitoring: Histogramme der Aktivierungen
- Weight-Monitoring: Norm der Gewichtsmatrizen