Deep Learning - Musterlösung Übung 3

Convolutional Neural Networks

Fachhochschule Südwestfalen

23. Oktober 2025

Hinweise zur Musterlösung

Diese Musterlösung enthält detaillierte mathematische Herleitungen und vollständige Implementierungen für CNNs.

1 Convolution-Mathematik - Lösungen

1.1 Aufgabe 1.1: Grundlegende Convolution

Gegeben:

$$\text{Input: } X = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \tag{1}$$

$$\text{Kernel: } K = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{2}$$

Valid Convolution (ohne Padding):
Output-Größe: $(3 - 2 + 1) \times (3 - 2 + 1) = 2 \times 2$

$$\begin{aligned} Y[0,0] &= \sum_{i=0}^{1}\sum_{j=0}^{1} X[i,j] \cdot K[i,j] && (3) \\ &= X[0,0] \cdot K[0,0] + X[0,1] \cdot K[0,1] + X[1,0] \cdot K[1,0] + X[1,1] \cdot K[1,1] && (4) \\ &= 1 \cdot 1 + 2 \cdot 0 + 4 \cdot 0 + 5 \cdot (-1) = 1 - 5 = -4 && (5) \end{aligned}$$

$$\begin{aligned} Y[0,1] &= X[0,1] \cdot 1 + X[0,2] \cdot 0 + X[1,1] \cdot 0 + X[1,2] \cdot (-1) && (6) \\ &= 2 \cdot 1 + 3 \cdot 0 + 5 \cdot 0 + 6 \cdot (-1) = 2 - 6 = -4 && (7) \end{aligned}$$

$$\begin{aligned} Y[1,0] &= X[1,0] \cdot 1 + X[1,1] \cdot 0 + X[2,0] \cdot 0 + X[2,1] \cdot (-1) && (8) \\ &= 4 \cdot 1 + 5 \cdot 0 + 7 \cdot 0 + 8 \cdot (-1) = 4 - 8 = -4 && (9) \end{aligned}$$

$$Y[1,1] = X[1,1] \cdot 1 + X[1,2] \cdot 0 + X[2,1] \cdot 0 + X[2,2] \cdot (-1) \quad (10)$$
$$= 5 \cdot 1 + 6 \cdot 0 + 8 \cdot 0 + 9 \cdot (-1) = 5 - 9 = -4 \quad (11)$$

$$\boxed{Y = \begin{pmatrix} -4 & -4 \\ -4 & -4 \end{pmatrix}}$$

Same Convolution (mit Padding):
Padding $= \lfloor \frac{2}{2} \rfloor = 1$, erweiterte Matrix:

$$X_{padded} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 0 \\ 0 & 4 & 5 & 6 & 0 \\ 0 & 7 & 8 & 9 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad (12)$$

Output-Größe: 3×3 (gleich wie Input)

$$Y_{same} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & -4 & -4 \\ 7 & -4 & -4 \end{pmatrix} \quad (13)$$

1.2 Aufgabe 1.2: Multi-Channel Convolution

Gegeben: 3-Kanal Input ($3 \times 3 \times 3$), 2 Filter ($2 \times 2 \times 3$)
Mathematische Formulierung:

$$Y^{(f)}[i,j] = \sum_{c=0}^{C-1} \sum_{u=0}^{K-1} \sum_{v=0}^{K-1} X^{(c)}[i+u, j+v] \cdot W^{(f,c)}[u,v] + b^{(f)} \quad (14)$$

Für Filter 1:

$$Y^{(1)}[0,0] = \sum_{c=0}^{2} \sum_{u=0}^{1} \sum_{v=0}^{1} X^{(c)}[u,v] \cdot W^{(1,c)}[u,v] + b^{(1)} \quad (15)$$

Implementierung:

```python
import numpy as np

def convolution_3d(input_volume, filters, biases, stride=1, padding
    =0):
    """
    3D Convolution for multi-channel inputs

    Args:
        input_volume: Shape (H, W, C)
        filters: Shape (F, K, K, C) - F filters, each KxK with C
            channels
        biases: Shape (F,)
        stride: Stride
```

```
        padding: Padding

    Returns:
        output: Shape (H_out, W_out, F)
    """
    H, W, C = input_volume.shape
    F, K, _, _ = filters.shape

    # Add padding
    if padding > 0:
        input_padded = np.pad(input_volume,
                              ((padding, padding), (padding, padding),
                                  (0, 0)),
                              mode='constant')
    else:
        input_padded = input_volume

    # Calculate output dimensions
    H_out = (H + 2*padding - K) // stride + 1
    W_out = (W + 2*padding - K) // stride + 1

    # Initialize output
    output = np.zeros((H_out, W_out, F))

    # Perform convolution
    for f in range(F):  # For each filter
        for i in range(H_out):
            for j in range(W_out):
                # Extract region
                region = input_padded[i*stride:i*stride+K,
                                      j*stride:j*stride+K, :]

                # Convolution operation
                output[i, j, f] = np.sum(region * filters[f]) +
                   biases[f]

    return output

# Example usage
input_vol = np.random.randn(5, 5, 3)
filters = np.random.randn(8, 3, 3, 3)  # 8 filters, 3x3, 3 channels
biases = np.random.randn(8)

result = convolution_3d(input_vol, filters, biases, padding=1)
print(f"Output shape: {result.shape}")
```

1.3 Aufgabe 1.3: Pooling-Operationen

Max Pooling (2×2):

$$\text{Input: } X = \begin{pmatrix} 1 & 3 & 2 & 4 \\ 5 & 6 & 1 & 2 \\ 3 & 2 & 4 & 7 \\ 1 & 8 & 3 & 9 \end{pmatrix} \tag{16}$$

$$Y_{max}[0,0] = \max(1,3,5,6) = 6 \tag{17}$$
$$Y_{max}[0,1] = \max(2,4,1,2) = 4 \tag{18}$$
$$Y_{max}[1,0] = \max(3,2,1,8) = 8 \tag{19}$$
$$Y_{max}[1,1] = \max(4,7,3,9) = 9 \tag{20}$$

$$\boxed{Y_{max} = \begin{pmatrix} 6 & 4 \\ 8 & 9 \end{pmatrix}}$$

Average Pooling:

$$Y_{avg}[0,0] = \frac{1+3+5+6}{4} = 3.75 \tag{21}$$
$$Y_{avg}[0,1] = \frac{2+4+1+2}{4} = 2.25 \tag{22}$$
$$Y_{avg}[1,0] = \frac{3+2+1+8}{4} = 3.5 \tag{23}$$
$$Y_{avg}[1,1] = \frac{4+7+3+9}{4} = 5.75 \tag{24}$$

$$\boxed{Y_{avg} = \begin{pmatrix} 3.75 & 2.25 \\ 3.5 & 5.75 \end{pmatrix}}$$

2 CNN-Implementierung - Musterlösung

2.1 Aufgabe 2.1: CNN von Grund auf

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

class Conv2D:
    def __init__(self, in_channels, out_channels, kernel_size, stride
        =1, padding=0):
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_size = kernel_size
        self.stride = stride
        self.padding = padding

        # He initialization for weights
        self.weights = np.random.randn(out_channels, in_channels,
```

```
                                          kernel_size, kernel_size) * np.sqrt(2.0 / (in_channels * kernel_size * kernel_size))
        self.biases = np.zeros(out_channels)

        # For backpropagation
        self.last_input = None
        self.dW = None
        self.db = None

    def forward(self, x):
        """Forward pass"""
        self.last_input = x
        batch_size, in_channels, height, width = x.shape

        # Add padding
        if self.padding > 0:
            x_padded = np.pad(x, ((0, 0), (0, 0),
                                  (self.padding, self.padding),
                                  (self.padding, self.padding)),
                              mode='constant')
        else:
            x_padded = x

        # Calculate output dimensions
        out_height = (height + 2*self.padding - self.kernel_size) // self.stride + 1
        out_width = (width + 2*self.padding - self.kernel_size) // self.stride + 1

        # Initialize output
        output = np.zeros((batch_size, self.out_channels, out_height, out_width))

        # Convolution operation
        for b in range(batch_size):
            for f in range(self.out_channels):
                for i in range(out_height):
                    for j in range(out_width):
                        # Extract region
                        region = x_padded[b, :,
                                          i*self.stride:i*self.stride+self.kernel_size,
                                          j*self.stride:j*self.stride+self.kernel_size]

                        # Convolution
                        output[b, f, i, j] = np.sum(region * self.weights[f]) + self.biases[f]

        return output
```

```

    def backward(self, dout):
        """Backward pass"""
        batch_size, in_channels, height, width = self.last_input.shape
        _, out_channels, out_height, out_width = dout.shape

        # Add padding to input
        if self.padding > 0:
            x_padded = np.pad(self.last_input, ((0, 0), (0, 0),
                                              (self.padding, self.padding),
                                              (self.padding, self.padding)),
                            mode='constant')
        else:
            x_padded = self.last_input

        # Initialize gradients
        self.dW = np.zeros_like(self.weights)
        self.db = np.zeros_like(self.biases)
        dx_padded = np.zeros_like(x_padded)

        # Compute gradients
        for b in range(batch_size):
            for f in range(out_channels):
                for i in range(out_height):
                    for j in range(out_width):
                        # Gradient w.r.t. weights
                        region = x_padded[b, :,
                                        i*self.stride:i*self.stride+self.kernel_size,
                                        j*self.stride:j*self.stride+self.kernel_size]
                        self.dW[f] += dout[b, f, i, j] * region

                        # Gradient w.r.t. bias
                        self.db[f] += dout[b, f, i, j]

                        # Gradient w.r.t. input
                        dx_padded[b, :,
                                i*self.stride:i*self.stride+self.kernel_size,
                                j*self.stride:j*self.stride+self.kernel_size] += \
                                dout[b, f, i, j] * self.weights[f]

        # Remove padding from gradient
        if self.padding > 0:
            dx = dx_padded[:, :, self.padding:-self.padding, self.padding:-self.padding]
```

```
        else:
            dx = dx_padded

        return dx

class MaxPool2D:
    def __init__(self, pool_size=2, stride=2):
        self.pool_size = pool_size
        self.stride = stride
        self.mask = None

    def forward(self, x):
        batch_size, channels, height, width = x.shape

        out_height = height // self.stride
        out_width = width // self.stride

        output = np.zeros((batch_size, channels, out_height, out_width))
        self.mask = np.zeros_like(x)

        for b in range(batch_size):
            for c in range(channels):
                for i in range(out_height):
                    for j in range(out_width):
                        # Extract pooling region
                        region = x[b, c,
                                 i*self.stride:i*self.stride+self.pool_size,
                                 j*self.stride:j*self.stride+self.pool_size]

                        # Max pooling
                        max_val = np.max(region)
                        output[b, c, i, j] = max_val

                        # Store mask for backpropagation
                        mask_region = (region == max_val)
                        self.mask[b, c,
                                i*self.stride:i*self.stride+self.pool_size,
                                j*self.stride:j*self.stride+self.pool_size] = mask_region

        return output

    def backward(self, dout):
        batch_size, channels, out_height, out_width = dout.shape
        dx = np.zeros_like(self.mask)

        for b in range(batch_size):
```

```
            for c in range(channels):
                for i in range(out_height):
                    for j in range(out_width):
                        # Distribute gradient to max element
                        dx[b, c,
                           i*self.stride:i*self.stride+self.pool_size,
                           j*self.stride:j*self.stride+self.pool_size] += \
                           dout[b, c, i, j] * self.mask[b, c,
                                                       i*self.stride:i*self.stride+self.pool_size,
                                                       j*self.stride:j*self.stride+self.pool_size]

        return dx

class ReLU:
    def __init__(self):
        self.mask = None

    def forward(self, x):
        self.mask = x > 0
        return np.maximum(0, x)

    def backward(self, dout):
        return dout * self.mask

class Dense:
    def __init__(self, in_features, out_features):
        self.weights = np.random.randn(out_features, in_features) * np.sqrt(2.0 / in_features)
        self.biases = np.zeros(out_features)
        self.last_input = None
        self.dW = None
        self.db = None

    def forward(self, x):
        # Flatten input if needed
        if len(x.shape) > 2:
            x = x.reshape(x.shape[0], -1)

        self.last_input = x
        return x @ self.weights.T + self.biases

    def backward(self, dout):
        self.dW = dout.T @ self.last_input
```

```python
        self.db = np.sum(dout, axis=0)
        dx = dout @ self.weights

        # Reshape if needed
        if hasattr(self, 'input_shape'):
            dx = dx.reshape(self.input_shape)

        return dx

class SimpleCNN:
    def __init__(self):
        # Architecture: Conv -> ReLU -> MaxPool -> Conv -> ReLU -> MaxPool -> Dense -> Softmax
        self.conv1 = Conv2D(1, 6, 5, padding=2)  # 28x28x1 -> 28x28x6
        self.relu1 = ReLU()
        self.pool1 = MaxPool2D(2, 2)  # 28x28x6 -> 14x14x6

        self.conv2 = Conv2D(6, 16, 5)  # 14x14x6 -> 10x10x16
        self.relu2 = ReLU()
        self.pool2 = MaxPool2D(2, 2)  # 10x10x16 -> 5x5x16

        self.dense1 = Dense(5*5*16, 120)
        self.relu3 = ReLU()
        self.dense2 = Dense(120, 84)
        self.relu4 = ReLU()
        self.dense3 = Dense(84, 10)  # 10 classes

        self.layers = [self.conv1, self.relu1, self.pool1,
                       self.conv2, self.relu2, self.pool2,
                       self.dense1, self.relu3,
                       self.dense2, self.relu4, self.dense3]

    def forward(self, x):
        for layer in self.layers:
            x = layer.forward(x)
        return x

    def backward(self, dout):
        for layer in reversed(self.layers):
            dout = layer.backward(dout)
        return dout

    def softmax(self, x):
        exp_x = np.exp(x - np.max(x, axis=1, keepdims=True))
        return exp_x / np.sum(exp_x, axis=1, keepdims=True)

    def cross_entropy_loss(self, y_true, y_pred):
        y_pred = self.softmax(y_pred)
        batch_size = y_true.shape[0]
        log_likelihood = -np.log(y_pred[range(batch_size), y_true])
        return np.mean(log_likelihood)
```

```

    def predict(self, x):
        output = self.forward(x)
        return np.argmax(self.softmax(output), axis=1)
```

2.2 Aufgabe 2.2: Training und Evaluierung

```
# Training function
def train_cnn(model, X_train, y_train, X_val, y_val, epochs=10,
   batch_size=32, learning_rate=0.001):
    train_losses = []
    val_accuracies = []

    for epoch in range(epochs):
        epoch_loss = 0
        num_batches = 0

        # Shuffle training data
        indices = np.random.permutation(len(X_train))
        X_train_shuffled = X_train[indices]
        y_train_shuffled = y_train[indices]

        # Mini-batch training
        for i in range(0, len(X_train), batch_size):
            batch_X = X_train_shuffled[i:i+batch_size]
            batch_y = y_train_shuffled[i:i+batch_size]

            # Forward pass
            output = model.forward(batch_X)

            # Compute loss
            loss = model.cross_entropy_loss(batch_y, output)
            epoch_loss += loss
            num_batches += 1

            # Backward pass
            # Gradient of cross-entropy + softmax
            y_pred = model.softmax(output)
            dout = y_pred.copy()
            dout[range(len(batch_y)), batch_y] -= 1
            dout /= len(batch_y)

            model.backward(dout)

            # Update weights
            update_weights(model, learning_rate)

        # Average loss
        avg_loss = epoch_loss / num_batches
        train_losses.append(avg_loss)
```

```python

        # Validation accuracy
        val_pred = model.predict(X_val)
        val_acc = np.mean(val_pred == y_val)
        val_accuracies.append(val_acc)

        print(f"Epoch {epoch+1}/{epochs}, Loss: {avg_loss:.4f}, Val Acc: {val_acc:.4f}")

    return train_losses, val_accuracies

def update_weights(model, learning_rate):
    """Update all trainable parameters"""
    for layer in model.layers:
        if hasattr(layer, 'weights'):
            layer.weights -= learning_rate * layer.dW
            layer.biases -= learning_rate * layer.db

# Synthetic data for testing
def create_synthetic_data(n_samples=1000):
    """Create simple synthetic image data"""
    X = np.random.randn(n_samples, 1, 28, 28)
    y = np.random.randint(0, 10, n_samples)
    return X, y

# Test the model
X_train, y_train = create_synthetic_data(1000)
X_val, y_val = create_synthetic_data(200)

model = SimpleCNN()
train_losses, val_accs = train_cnn(model, X_train, y_train, X_val, y_val,
                                   epochs=5, batch_size=16, learning_rate=0.01)

# Plot results
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(train_losses)
plt.title('Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')

plt.subplot(1, 2, 2)
plt.plot(val_accs)
plt.title('Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.tight_layout()
plt.show()
```

3 Backpropagation in CNNs - Lösungen

3.1 Aufgabe 3.1: Convolution Backpropagation

Mathematische Herleitung:

Für eine Convolution-Schicht mit Output $Y = X * W + b$:

Gradient bezüglich Weights:

$$\frac{\partial L}{\partial W[f,c,u,v]} = \sum_{i,j} \frac{\partial L}{\partial Y[f,i,j]} \cdot X[c, i \cdot s + u, j \cdot s + v] \tag{25}$$

Gradient bezüglich Input:

$$\frac{\partial L}{\partial X[c,i,j]} = \sum_{f,u,v} \frac{\partial L}{\partial Y[f,i',j']} \cdot W[f,c,u,v] \tag{26}$$

wobei $i' = \frac{i-u}{s}$, $j' = \frac{j-v}{s}$ (wenn ganzzahlig und im gültigen Bereich).

Numerisches Beispiel:

Gegeben: - Input: X (1×1×3×3) - Kernel: W (1×1×2×2) - Output: Y (1×1×2×2) - Gradient: $\frac{\partial L}{\partial Y}$ (1×1×2×2)

$$\frac{\partial L}{\partial Y} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \tag{27}$$

Gradient bezüglich Kernel:

$$\frac{\partial L}{\partial W[0,0]} = X[0,0] \cdot 1 + X[0,1] \cdot 2 + X[1,0] \cdot 3 + X[1,1] \cdot 4 \tag{28}$$

$$\frac{\partial L}{\partial W[0,1]} = X[0,1] \cdot 1 + X[0,2] \cdot 2 + X[1,1] \cdot 3 + X[1,2] \cdot 4 \tag{29}$$

$$\frac{\partial L}{\partial W[1,0]} = X[1,0] \cdot 1 + X[1,1] \cdot 2 + X[2,0] \cdot 3 + X[2,1] \cdot 4 \tag{30}$$

$$\frac{\partial L}{\partial W[1,1]} = X[1,1] \cdot 1 + X[1,2] \cdot 2 + X[2,1] \cdot 3 + X[2,2] \cdot 4 \tag{31}$$

3.2 Aufgabe 3.2: Pooling Backpropagation

Max Pooling Gradient:

Das Gradient wird nur an die Position weitergegeben, die das Maximum hatte:

$$\frac{\partial L}{\partial X[i,j]} = \begin{cases} \frac{\partial L}{\partial Y[i',j']} & \text{wenn } X[i,j] = \max(\text{pooling region}) \\ 0 & \text{sonst} \end{cases} \tag{32}$$

Implementierung:

```python
def max_pool_backward_detailed(dout, x, pool_size=2, stride=2):
    """
    Detailed implementation of max pooling backward pass
    """
```

```
    batch_size, channels, height, width = x.shape
    out_height, out_width = dout.shape[2], dout.shape[3]

    dx = np.zeros_like(x)

    for b in range(batch_size):
        for c in range(channels):
            for i in range(out_height):
                for j in range(out_width):
                    # Get pooling region
                    h_start = i * stride
                    h_end = h_start + pool_size
                    w_start = j * stride
                    w_end = w_start + pool_size

                    region = x[b, c, h_start:h_end, w_start:w_end]

                    # Find position of maximum
                    max_pos = np.unravel_index(np.argmax(region),
                        region.shape)

                    # Pass gradient to max position
                    dx[b, c, h_start + max_pos[0], w_start + max_pos
                        [1]] += dout[b, c, i, j]

    return dx
```

4 Advanced Topics - Lösungen

4.1 Aufgabe 4.1: Data Augmentation

```
def augment_data(X, y):
    """Data augmentation for image classification"""
    augmented_X = []
    augmented_y = []

    for i in range(len(X)):
        image = X[i].copy()
        label = y[i]

        # Original image
        augmented_X.append(image)
        augmented_y.append(label)

        # Horizontal flip
        flipped = np.flip(image, axis=-1)
        augmented_X.append(flipped)
        augmented_y.append(label)

        # Rotation (simplified - small angle)
```

```python
        angle = np.random.uniform(-15, 15)
        rotated = rotate_image(image, angle)
        augmented_X.append(rotated)
        augmented_y.append(label)

        # Brightness adjustment
        bright_factor = np.random.uniform(0.8, 1.2)
        brightened = np.clip(image * bright_factor, 0, 1)
        augmented_X.append(brightened)
        augmented_y.append(label)

        # Noise addition
        noise = np.random.normal(0, 0.1, image.shape)
        noisy = np.clip(image + noise, 0, 1)
        augmented_X.append(noisy)
        augmented_y.append(label)

    return np.array(augmented_X), np.array(augmented_y)

def rotate_image(image, angle):
    """Simple rotation implementation"""
    # This would typically use scipy.ndimage.rotate
    # Here's a placeholder implementation
    return image  # Simplified for this example
```

4.2 Aufgabe 4.2: Transfer Learning

Konzeptuelle Erklärung:

Transfer Learning nutzt vortrainierte Modelle: 1. **Feature Extraction:** Frostere frühe Schichten, trainiere nur Klassifikator 2. **Fine-tuning:** Trainiere alle Schichten mit kleiner Learning Rate 3. **Progressive Unfreezing:** Schrittweise Freigabe von Schichten

Implementierung:

```python
class TransferCNN(SimpleCNN):
    def __init__(self, pretrained_features=None, num_classes=10):
        super().__init__()

        if pretrained_features:
            # Load pretrained convolutional layers
            self.conv1 = pretrained_features['conv1']
            self.conv2 = pretrained_features['conv2']

            # Freeze convolutional layers
            self.freeze_conv_layers()

        # Replace classifier
        self.dense3 = Dense(84, num_classes)

    def freeze_conv_layers(self):
        """Freeze convolutional layers for feature extraction"""
        self.conv1.trainable = False
```

```
        self.conv2.trainable = False

    def unfreeze_conv_layers(self):
        """Unfreeze for fine-tuning"""
        self.conv1.trainable = True
        self.conv2.trainable = True

    def fine_tune(self, X_train, y_train, epochs=5, learning_rate=0.0001):
        """Fine-tuning with small learning rate"""
        self.unfreeze_conv_layers()

        # Train with very small learning rate
        train_losses, _ = train_cnn(self, X_train, y_train, X_train, y_train,
                                   epochs=epochs, learning_rate=learning_rate)
        return train_losses
```

Zusammenfassung und Best Practices

CNN Design Principles

- **Receptive Field:** Schrittweise Vergrößerung durch mehrere kleine Kernel
- **Channel Progression:** Mehr Kanäle in tieferen Schichten
- **Pooling Strategy:** Max für Features, Average für globale Information
- **Activation Choice:** ReLU für Hidden Layers, Softmax für Klassifikation

Training Optimizations

- **Batch Normalization:** Stabilisiert Training
- **Dropout:** Regularisierung gegen Overfitting
- **Learning Rate Scheduling:** Adaptive Anpassung
- **Early Stopping:** Verhindert Overfitting