Deep Learning - Musterlösung Übung 5

Generative Modelle und Fortgeschrittenes Deep Learning

Fachhochschule Südwestfalen

23. Oktober 2025

Hinweise zur Musterlösung

Diese Musterlösung bietet umfassende mathematische Herleitungen und praktische Implementierungen für generative Modelle und fortgeschrittene Deep Learning Techniken.

1 Autoencoders - Lösungen

1.1 Aufgabe 1.1: Autoencoder-Mathematik

Encoder-Decoder Architektur:

$$\mathbf{z} = f_{\text{enc}}(\mathbf{x}) = \sigma(\mathbf{W}_e\mathbf{x} + \mathbf{b}_e) \tag{1}$$

$$\hat{\mathbf{x}} = f_{\text{dec}}(\mathbf{z}) = \sigma(\mathbf{W}_d\mathbf{z} + \mathbf{b}_d) \tag{2}$$

$$L(\mathbf{x}, \hat{\mathbf{x}}) = \|\mathbf{x} - \hat{\mathbf{x}}\|^2 \tag{3}$$

Dimensionsanalyse: Für MNIST-Daten ($\mathbf{x} \in \mathbb{R}^{784}$) und latenten Raum ($\mathbf{z} \in \mathbb{R}^{32}$):

$$\mathbf{W}_e \in \mathbb{R}^{32\times 784}, \quad \mathbf{b}_e \in \mathbb{R}^{32} \tag{4}$$

$$\mathbf{W}_d \in \mathbb{R}^{784\times 32}, \quad \mathbf{b}_d \in \mathbb{R}^{784} \tag{5}$$

Gradientenberechnung:

Decoder-Gradienten:

$$\frac{\partial L}{\partial \mathbf{W}_d} = \frac{\partial L}{\partial \hat{\mathbf{x}}}\frac{\partial \hat{\mathbf{x}}}{\partial \mathbf{W}_d} \tag{6}$$

$$= -2(\mathbf{x} - \hat{\mathbf{x}}) \odot \sigma'(\mathbf{W}_d\mathbf{z} + \mathbf{b}_d)\mathbf{z}^T \tag{7}$$

Encoder-Gradienten (Backpropagation durch Decoder):

$$\frac{\partial L}{\partial \mathbf{z}} = \mathbf{W}_d^T\left[-2(\mathbf{x} - \hat{\mathbf{x}}) \odot \sigma'(\mathbf{W}_d\mathbf{z} + \mathbf{b}_d)\right] \tag{8}$$

$$\frac{\partial L}{\partial \mathbf{W}_e} = \frac{\partial L}{\partial \mathbf{z}} \odot \sigma'(\mathbf{W}_e\mathbf{x} + \mathbf{b}_e)\mathbf{x}^T \tag{9}$$

Kompressionsverhältnis:

$$\text{Compression Ratio} = \frac{\text{Original Size}}{\text{Compressed Size}} = \frac{784}{32} = 24.5 \tag{10}$$

Vergleich mit JPEG: JPEG erreicht typischerweise 10:1 bis 50:1, aber verlustbehaftet. Der Autoencoder lernt eine datenspezifische Kompression.

PCA-Vergleich: Linearer Autoencoder mit einer Hidden Layer ist äquivalent zu PCA, wenn:

- Keine Bias-Terms verwendet werden
- MSE Loss verwendet wird
- Globales Minimum erreicht wird

Beweis: Die optimalen Gewichte $\mathbf{W}_d$ entsprechen den ersten k Hauptkomponenten.

1.2 Aufgabe 1.2: Autoencoder-Implementierung

Standard Autoencoder:

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

class StandardAutoencoder:
    def __init__(self, input_dim, latent_dim):
        self.input_dim = input_dim
        self.latent_dim = latent_dim

        # Xavier initialization
        self.W_encoder = np.random.randn(latent_dim, input_dim) * np.
            sqrt(2.0 / input_dim)
        self.b_encoder = np.zeros((latent_dim, 1))

        self.W_decoder = np.random.randn(input_dim, latent_dim) * np.
            sqrt(2.0 / latent_dim)
        self.b_decoder = np.zeros((input_dim, 1))

        # For storing during forward pass
        self.cache = {}

    def sigmoid(self, x):
        return np.where(x >= 0,
                        1 / (1 + np.exp(-x)),
                        np.exp(x) / (1 + np.exp(x)))

    def sigmoid_derivative(self, x):
        s = self.sigmoid(x)
        return s * (1 - s)

    def encode(self, x):
        """Encode input to latent representation"""
        z_pre = self.W_encoder @ x + self.b_encoder
        z = self.sigmoid(z_pre)
        return z, z_pre

    def decode(self, z):
```

```
        """Decode latent representation to reconstruction"""
        x_pre = self.W_decoder @ z + self.b_decoder
        x_reconstructed = self.sigmoid(x_pre)
        return x_reconstructed, x_pre

    def forward(self, x):
        """Complete forward pass"""
        # Encoder
        z, z_pre = self.encode(x)

        # Decoder
        x_reconstructed, x_pre = self.decode(z)

        # Store for backpropagation
        self.cache = {
            'x': x,
            'z_pre': z_pre,
            'z': z,
            'x_pre': x_pre,
            'x_reconstructed': x_reconstructed
        }

        return x_reconstructed

    def backward(self, x_reconstructed, x_target):
        """Backpropagation"""
        # Loss gradient
        dLoss_dx_reconstructed = 2 * (x_reconstructed - x_target)

        # Decoder gradients
        dx_pre = dLoss_dx_reconstructed * self.sigmoid_derivative(
            self.cache['x_pre'])
        dW_decoder = dx_pre @ self.cache['z'].T
        db_decoder = np.sum(dx_pre, axis=1, keepdims=True)

        # Encoder gradients
        dz = self.W_decoder.T @ dx_pre
        dz_pre = dz * self.sigmoid_derivative(self.cache['z_pre'])
        dW_encoder = dz_pre @ self.cache['x'].T
        db_encoder = np.sum(dz_pre, axis=1, keepdims=True)

        return {
            'dW_encoder': dW_encoder,
            'db_encoder': db_encoder,
            'dW_decoder': dW_decoder,
            'db_decoder': db_decoder
        }

    def update_weights(self, gradients, learning_rate):
        """Update weights using gradients"""
        self.W_encoder -= learning_rate * gradients['dW_encoder']
```

```
        self.b_encoder -= learning_rate * gradients['db_encoder']
        self.W_decoder -= learning_rate * gradients['dW_decoder']
        self.b_decoder -= learning_rate * gradients['db_decoder']

    def train(self, X, epochs=1000, learning_rate=0.01, batch_size
        =32):
        """Training loop"""
        losses = []

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

            # Shuffle data
            indices = np.random.permutation(X.shape[1])
            X_shuffled = X[:, indices]

            # Mini-batch training
            for i in range(0, X.shape[1], batch_size):
                batch_X = X_shuffled[:, i:i+batch_size]

                # Forward pass
                x_reconstructed = self.forward(batch_X)

                # Compute loss
                loss = np.mean((batch_X - x_reconstructed)**2)
                epoch_loss += loss
                num_batches += 1

                # Backward pass
                gradients = self.backward(x_reconstructed, batch_X)

                # Update weights
                self.update_weights(gradients, learning_rate)

            avg_loss = epoch_loss / num_batches
            losses.append(avg_loss)

            if (epoch + 1) % 100 == 0:
                print(f"Epoch {epoch+1}/{epochs}, Loss: {avg_loss:.6f
                    }")

        return losses

    def reconstruct(self, x):
        """Reconstruct single input"""
        return self.forward(x.reshape(-1, 1)).flatten()

class DenoisingAutoencoder(StandardAutoencoder):
    def __init__(self, input_dim, latent_dim, noise_factor=0.3):
        super().__init__(input_dim, latent_dim)
```

```python
        self.noise_factor = noise_factor

    def add_noise(self, x):
        """Add Gaussian noise to input"""
        noise = np.random.normal(0, self.noise_factor, x.shape)
        noisy_x = x + noise
        return np.clip(noisy_x, 0, 1)  # Ensure values stay in [0,1]

    def train_denoising(self, X, epochs=1000, learning_rate=0.01, batch_size=32):
        """Training with noise"""
        losses = []

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

            # Shuffle data
            indices = np.random.permutation(X.shape[1])
            X_shuffled = X[:, indices]

            for i in range(0, X.shape[1], batch_size):
                batch_X = X_shuffled[:, i:i+batch_size]

                # Add noise to input
                noisy_X = self.add_noise(batch_X)

                # Forward pass with noisy input
                x_reconstructed = self.forward(noisy_X)

                # Loss computed against clean target
                loss = np.mean((batch_X - x_reconstructed)**2)
                epoch_loss += loss
                num_batches += 1

                # Backward pass
                gradients = self.backward(x_reconstructed, batch_X)
                self.update_weights(gradients, learning_rate)

            avg_loss = epoch_loss / num_batches
            losses.append(avg_loss)

            if (epoch + 1) % 100 == 0:
                print(f"Epoch {epoch+1}/{epochs}, Denoising Loss: {avg_loss:.6f}")

        return losses

# Test with synthetic data
def create_synthetic_mnist():
    """Create synthetic MNIST-like data"""
```

```python
    np.random.seed(42)

    # Create simple patterns
    data = []
    for _ in range(1000):
        # Create a 28x28 image with simple patterns
        img = np.zeros((28, 28))

        # Random rectangles, circles, lines
        if np.random.rand() < 0.33:
            # Rectangle
            x1, y1 = np.random.randint(5, 15, 2)
            x2, y2 = np.random.randint(x1+3, 25, 2)
            img[x1:x2, y1:y2] = 1
        elif np.random.rand() < 0.66:
            # Circle
            center = np.random.randint(8, 20, 2)
            radius = np.random.randint(3, 8)
            y, x = np.ogrid[:28, :28]
            mask = (x - center[0])**2 + (y - center[1])**2 <= radius**2
            img[mask] = 1
        else:
            # Line
            x1, y1 = np.random.randint(0, 28, 2)
            x2, y2 = np.random.randint(0, 28, 2)
            # Simple line drawing
            length = max(abs(x2-x1), abs(y2-y1))
            for t in np.linspace(0, 1, length):
                x = int(x1 + t*(x2-x1))
                y = int(y1 + t*(y2-y1))
                if 0 <= x < 28 and 0 <= y < 28:
                    img[x, y] = 1

        data.append(img.flatten())

    return np.array(data).T  # Shape: (784, 1000)

# Example usage
print("Erstelle synthetische Daten...")
X_synthetic = create_synthetic_mnist()

print("Trainiere Standard Autoencoder...")
autoencoder = StandardAutoencoder(input_dim=784, latent_dim=32)
losses_standard = autoencoder.train(X_synthetic, epochs=500, learning_rate=0.01)

print("Trainiere Denoising Autoencoder...")
denoising_ae = DenoisingAutoencoder(input_dim=784, latent_dim=32, noise_factor=0.3)
losses_denoising = denoising_ae.train_denoising(X_synthetic, epochs
```

```
=500, learning_rate=0.01)

# Visualization
plt.figure(figsize=(12, 4))

# Plot losses
plt.subplot(1, 3, 1)
plt.plot(losses_standard, label='Standard AE')
plt.plot(losses_denoising, label='Denoising AE')
plt.title('Training Losses')
plt.xlabel('Epoch')
plt.ylabel('MSE Loss')
plt.legend()
plt.grid(True)

# Original vs Reconstruction
test_idx = 0
original = X_synthetic[:, test_idx].reshape(28, 28)
reconstructed_std = autoencoder.reconstruct(X_synthetic[:, test_idx]).reshape(28, 28)

# Add noise for denoising test
noisy_input = denoising_ae.add_noise(X_synthetic[:, test_idx:test_idx+1])
reconstructed_denoising = denoising_ae.forward(noisy_input).reshape(28, 28)

plt.subplot(1, 3, 2)
plt.imshow(original, cmap='gray')
plt.title('Original')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(reconstructed_std, cmap='gray')
plt.title('Rekonstruktion')
plt.axis('off')

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

2 Variational Autoencoders (VAE) - Lösungen

2.1 Aufgabe 2.1: VAE-Mathematik

VAE-Formulierung:

$$q_\phi(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\boldsymbol{\mu}_\phi(\mathbf{x}), \boldsymbol{\sigma}^2_\phi(\mathbf{x})) \quad (11)$$

$$p_\theta(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\boldsymbol{\mu}_\theta(\mathbf{z}), \mathbf{I}) \quad (12)$$

$$\mathcal{L}(\mathbf{x}) = \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})] - D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}) \| p(\mathbf{z})) \quad (13)$$

KL-Divergenz geschlossen: Für $q(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}^2)$ und $p(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$:

$$D_{KL}(q(\mathbf{z}|\mathbf{x}) \| p(\mathbf{z})) = \frac{1}{2} \sum_{j=1}^{J} (\mu_j^2 + \sigma_j^2 - \log \sigma_j^2 - 1) \tag{14}$$

Reparametrisierung-Trick:

$$\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon} \tag{15}$$

$$\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \tag{16}$$

Dies macht den Sampling-Prozess differenzierbar.

2.2 Aufgabe 2.2: VAE-Implementierung

```
class VariationalAutoencoder:
    def __init__(self, input_dim, latent_dim):
        self.input_dim = input_dim
        self.latent_dim = latent_dim

        # Encoder network (outputs mu and log_var)
        self.W_enc_1 = np.random.randn(256, input_dim) * 0.01
        self.b_enc_1 = np.zeros((256, 1))

        self.W_mu = np.random.randn(latent_dim, 256) * 0.01
        self.b_mu = np.zeros((latent_dim, 1))

        self.W_logvar = np.random.randn(latent_dim, 256) * 0.01
        self.b_logvar = np.zeros((latent_dim, 1))

        # Decoder network
        self.W_dec_1 = np.random.randn(256, latent_dim) * 0.01
        self.b_dec_1 = np.zeros((256, 1))

        self.W_dec_2 = np.random.randn(input_dim, 256) * 0.01
        self.b_dec_2 = np.zeros((input_dim, 1))

    def relu(self, x):
        return np.maximum(0, x)

    def relu_derivative(self, x):
        return (x > 0).astype(float)

    def sigmoid(self, x):
        return np.where(x >= 0,
                        1 / (1 + np.exp(-x)),
                        np.exp(x) / (1 + np.exp(x)))

    def encode(self, x):
        """Encoder: x -> mu, log_var"""
        h1 = self.relu(self.W_enc_1 @ x + self.b_enc_1)
```

```
        mu = self.W_mu @ h1 + self.b_mu
        log_var = self.W_logvar @ h1 + self.b_logvar
        return mu, log_var, h1

    def reparameterize(self, mu, log_var):
        """Reparameterization trick"""
        std = np.exp(0.5 * log_var)
        eps = np.random.normal(0, 1, mu.shape)
        z = mu + std * eps
        return z, eps

    def decode(self, z):
        """Decoder: z -> x_reconstructed"""
        h1 = self.relu(self.W_dec_1 @ z + self.b_dec_1)
        x_reconstructed = self.sigmoid(self.W_dec_2 @ h1 + self.
           b_dec_2)
        return x_reconstructed, h1

    def forward(self, x):
        """Complete forward pass"""
        # Encode
        mu, log_var, h_enc = self.encode(x)

        # Sample
        z, eps = self.reparameterize(mu, log_var)

        # Decode
        x_reconstructed, h_dec = self.decode(z)

        # Store for backpropagation
        self.cache = {
            'x': x,
            'h_enc': h_enc,
            'mu': mu,
            'log_var': log_var,
            'z': z,
            'eps': eps,
            'h_dec': h_dec,
            'x_reconstructed': x_reconstructed
        }

        return x_reconstructed, mu, log_var

    def compute_loss(self, x, x_reconstructed, mu, log_var):
        """VAE loss = Reconstruction loss + KL divergence"""
        batch_size = x.shape[1]

        # Reconstruction loss (binary cross-entropy)
        reconstruction_loss = -np.sum(
            x * np.log(x_reconstructed + 1e-8) +
            (1 - x) * np.log(1 - x_reconstructed + 1e-8)
```

```python
        ) / batch_size

        # KL divergence
        kl_loss = -0.5 * np.sum(1 + log_var - mu**2 - np.exp(log_var)
            ) / batch_size

        total_loss = reconstruction_loss + kl_loss

        return total_loss, reconstruction_loss, kl_loss

    def generate(self, num_samples=1):
        """Generate new samples from prior"""
        z = np.random.normal(0, 1, (self.latent_dim, num_samples))
        generated, _ = self.decode(z)
        return generated

# Example VAE training (simplified)
def train_vae_example():
    # Create simple synthetic data
    X = np.random.rand(784, 100)  # 100 samples of 784-dim data

    vae = VariationalAutoencoder(input_dim=784, latent_dim=20)

    print("Training VAE...")
    for epoch in range(100):
        # Forward pass
        x_recon, mu, log_var = vae.forward(X)

        # Compute loss
        total_loss, recon_loss, kl_loss = vae.compute_loss(X, x_recon
            , mu, log_var)

        if (epoch + 1) % 20 == 0:
            print(f"Epoch {epoch+1}: Total Loss: {total_loss:.4f}, "
                  f"Recon: {recon_loss:.4f}, KL: {kl_loss:.4f}")

    # Generate new samples
    generated = vae.generate(5)
    print(f"Generated samples shape: {generated.shape}")

train_vae_example()
```

3 Generative Adversarial Networks (GANs) - Lösungen

3.1 Aufgabe 3.1: GAN-Theorie

Minimax-Spiel:

$$\min_G \max_D V(D,G) = \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p_z}[\log(1 - D(G(\mathbf{z})))] \tag{17}$$

Optimaler Discriminator: Für festen Generator G, der optimale Discriminator ist:

$$D^*(\mathbf{x}) = \frac{p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_g(\mathbf{x})} \tag{18}$$

Beweis: Zu maximieren:

$$V(G,D) = \int_{\mathbf{x}} p_{\text{data}}(\mathbf{x}) \log D(\mathbf{x}) + p_g(\mathbf{x}) \log(1 - D(\mathbf{x})) d\mathbf{x} \tag{19}$$

Ableitung nach $D(\mathbf{x})$ und Nullsetzen:

$$\frac{\partial}{\partial D(\mathbf{x})} V(G,D) = \frac{p_{\text{data}}(\mathbf{x})}{D(\mathbf{x})} - \frac{p_g(\mathbf{x})}{1 - D(\mathbf{x})} = 0 \tag{20}$$

Lösung: $D^*(\mathbf{x}) = \frac{p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x})+p_g(\mathbf{x})}$
Globales Optimum: Wenn $p_g = p_{\text{data}}$, dann $D^*(\mathbf{x}) = \frac{1}{2}$ und $V(G^*, D^*) = -\log 4$.

3.2 Aufgabe 3.2: GAN-Implementierung

```python
class SimpleGAN:
    def __init__(self, latent_dim=100, data_dim=784):
        self.latent_dim = latent_dim
        self.data_dim = data_dim

        # Generator weights
        self.G_W1 = np.random.randn(128, latent_dim) * 0.01
        self.G_b1 = np.zeros((128, 1))
        self.G_W2 = np.random.randn(data_dim, 128) * 0.01
        self.G_b2 = np.zeros((data_dim, 1))

        # Discriminator weights
        self.D_W1 = np.random.randn(128, data_dim) * 0.01
        self.D_b1 = np.zeros((128, 1))
        self.D_W2 = np.random.randn(1, 128) * 0.01
        self.D_b2 = np.zeros((1, 1))

    def leaky_relu(self, x, alpha=0.2):
        return np.where(x > 0, x, alpha * x)

    def leaky_relu_derivative(self, x, alpha=0.2):
        return np.where(x > 0, 1, alpha)
```

```python

    def sigmoid(self, x):
        return np.where(x >= 0,
                        1 / (1 + np.exp(-x)),
                        np.exp(x) / (1 + np.exp(x)))

    def sigmoid_derivative(self, x):
        s = self.sigmoid(x)
        return s * (1 - s)

    def generator(self, z):
        """Generator: z -> fake_data"""
        h1 = self.leaky_relu(self.G_W1 @ z + self.G_b1)
        output = self.sigmoid(self.G_W2 @ h1 + self.G_b2)
        return output, h1

    def discriminator(self, x):
        """Discriminator: x -> probability"""
        h1 = self.leaky_relu(self.D_W1 @ x + self.D_b1)
        output = self.sigmoid(self.D_W2 @ h1 + self.D_b2)
        return output, h1

    def train_discriminator(self, real_data, fake_data, learning_rate
       =0.0002):
        """Train discriminator for one step"""
        batch_size = real_data.shape[1]

        # Forward pass on real data
        real_output, real_h1 = self.discriminator(real_data)

        # Forward pass on fake data
        fake_output, fake_h1 = self.discriminator(fake_data)

        # Discriminator loss
        d_loss_real = -np.mean(np.log(real_output + 1e-8))
        d_loss_fake = -np.mean(np.log(1 - fake_output + 1e-8))
        d_loss = d_loss_real + d_loss_fake

        # Gradients for real data
        d_real_output = -1 / (real_output + 1e-8) / batch_size
        d_real_h1_pre = d_real_output * self.sigmoid_derivative(self.
           D_W2 @ real_h1 + self.D_b2)
        d_D_W2_real = d_real_h1_pre @ real_h1.T
        d_D_b2_real = np.sum(d_real_h1_pre, axis=1, keepdims=True)

        d_real_h1 = self.D_W2.T @ d_real_h1_pre
        d_real_h1_pre_2 = d_real_h1 * self.leaky_relu_derivative(self
           .D_W1 @ real_data + self.D_b1)
        d_D_W1_real = d_real_h1_pre_2 @ real_data.T
        d_D_b1_real = np.sum(d_real_h1_pre_2, axis=1, keepdims=True)

```

```python
        # Gradients for fake data
        d_fake_output = 1 / (1 - fake_output + 1e-8) / batch_size
        d_fake_h1_pre = d_fake_output * self.sigmoid_derivative(self.
            D_W2 @ fake_h1 + self.D_b2)
        d_D_W2_fake = d_fake_h1_pre @ fake_h1.T
        d_D_b2_fake = np.sum(d_fake_h1_pre, axis=1, keepdims=True)

        d_fake_h1 = self.D_W2.T @ d_fake_h1_pre
        d_fake_h1_pre_2 = d_fake_h1 * self.leaky_relu_derivative(self
            .D_W1 @ fake_data + self.D_b1)
        d_D_W1_fake = d_fake_h1_pre_2 @ fake_data.T
        d_D_b1_fake = np.sum(d_fake_h1_pre_2, axis=1, keepdims=True)

        # Update discriminator weights
        self.D_W2 -= learning_rate * (d_D_W2_real + d_D_W2_fake)
        self.D_b2 -= learning_rate * (d_D_b2_real + d_D_b2_fake)
        self.D_W1 -= learning_rate * (d_D_W1_real + d_D_W1_fake)
        self.D_b1 -= learning_rate * (d_D_b1_real + d_D_b1_fake)

        return d_loss

    def train_generator(self, z, learning_rate=0.0002):
        """Train generator for one step"""
        batch_size = z.shape[1]

        # Generate fake data
        fake_data, g_h1 = self.generator(z)

        # Pass through discriminator
        d_output, d_h1 = self.discriminator(fake_data)

        # Generator loss (wants discriminator to output 1)
        g_loss = -np.mean(np.log(d_output + 1e-8))

        # Backpropagate through discriminator (frozen weights)
        d_d_output = -1 / (d_output + 1e-8) / batch_size
        d_d_h1_pre = d_d_output * self.sigmoid_derivative(self.D_W2 @
             d_h1 + self.D_b2)
        d_d_h1 = self.D_W2.T @ d_d_h1_pre
        d_fake_data = self.D_W1.T @ (d_d_h1 * self.
            leaky_relu_derivative(self.D_W1 @ fake_data + self.D_b1))

        # Backpropagate through generator
        d_g_output = d_fake_data * self.sigmoid_derivative(self.G_W2
            @ g_h1 + self.G_b2)
        d_G_W2 = d_g_output @ g_h1.T
        d_G_b2 = np.sum(d_g_output, axis=1, keepdims=True)

        d_g_h1 = self.G_W2.T @ d_g_output
        d_g_h1_pre = d_g_h1 * self.leaky_relu_derivative(self.G_W1 @
            z + self.G_b1)
```

```python
        d_G_W1 = d_g_h1_pre @ z.T
        d_G_b1 = np.sum(d_g_h1_pre, axis=1, keepdims=True)

        # Update generator weights
        self.G_W2 -= learning_rate * d_G_W2
        self.G_b2 -= learning_rate * d_G_b2
        self.G_W1 -= learning_rate * d_G_W1
        self.G_b1 -= learning_rate * d_G_b1

        return g_loss

    def generate_samples(self, num_samples):
        """Generate samples from random noise"""
        z = np.random.normal(0, 1, (self.latent_dim, num_samples))
        generated, _ = self.generator(z)
        return generated

# Example training
def train_gan_example():
    # Synthetic real data
    real_data = np.random.rand(784, 1000)

    gan = SimpleGAN(latent_dim=100, data_dim=784)

    epochs = 1000
    batch_size = 64

    print("Training GAN...")
    for epoch in range(epochs):
        # Random batch of real data
        idx = np.random.randint(0, real_data.shape[1], batch_size)
        real_batch = real_data[:, idx]

        # Generate fake data
        z = np.random.normal(0, 1, (gan.latent_dim, batch_size))
        fake_batch, _ = gan.generator(z)

        # Train discriminator
        d_loss = gan.train_discriminator(real_batch, fake_batch)

        # Train generator
        z = np.random.normal(0, 1, (gan.latent_dim, batch_size))
        g_loss = gan.train_generator(z)

        if (epoch + 1) % 100 == 0:
            print(f"Epoch {epoch+1}: D_loss: {d_loss:.4f}, G_loss: {g_loss:.4f}")

    # Generate samples
    samples = gan.generate_samples(10)
```

```
    print(f"Generated {samples.shape[1]} samples of dimension {
        samples.shape[0]}")

train_gan_example()
```

4 Vertiefende Aufgaben - Lösungen

4.1 Aufgabe 4.1: Data Augmentation

```
class DataAugmentation:
    def __init__(self):
        pass

    def horizontal_flip(self, image):
        """Horizontal flip"""
        return np.fliplr(image)

    def vertical_flip(self, image):
        """Vertical flip"""
        return np.flipud(image)

    def rotation(self, image, angle):
        """Simple rotation (simplified implementation)"""
        # In practice, use scipy.ndimage.rotate or cv2.rotate
        # This is a placeholder
        return image

    def gaussian_noise(self, image, mean=0, std=0.1):
        """Add Gaussian noise"""
        noise = np.random.normal(mean, std, image.shape)
        noisy_image = image + noise
        return np.clip(noisy_image, 0, 1)

    def brightness_adjustment(self, image, factor):
        """Adjust brightness"""
        bright_image = image * factor
        return np.clip(bright_image, 0, 1)

    def contrast_adjustment(self, image, factor):
        """Adjust contrast"""
        mean = np.mean(image)
        contrast_image = (image - mean) * factor + mean
        return np.clip(contrast_image, 0, 1)

    def random_crop(self, image, crop_size):
        """Random crop"""
        h, w = image.shape
        ch, cw = crop_size

        if h < ch or w < cw:
```

```
            return image

        x = np.random.randint(0, h - ch + 1)
        y = np.random.randint(0, w - cw + 1)

        return image[x:x+ch, y:y+cw]

    def augment_batch(self, images, augmentation_prob=0.5):
        """Apply random augmentations to a batch"""
        augmented = []

        for img in images:
            # Reshape if needed
            if img.ndim == 1:
                img = img.reshape(28, 28)  # Assume MNIST-like

            # Apply random augmentations
            if np.random.rand() < augmentation_prob:
                # Choose random augmentation
                aug_type = np.random.choice(['flip', 'noise', '
                    brightness', 'contrast'])

                if aug_type == 'flip':
                    if np.random.rand() < 0.5:
                        img = self.horizontal_flip(img)
                    else:
                        img = self.vertical_flip(img)
                elif aug_type == 'noise':
                    img = self.gaussian_noise(img, std=0.1)
                elif aug_type == 'brightness':
                    factor = np.random.uniform(0.8, 1.2)
                    img = self.brightness_adjustment(img, factor)
                elif aug_type == 'contrast':
                    factor = np.random.uniform(0.8, 1.2)
                    img = self.contrast_adjustment(img, factor)

            augmented.append(img.flatten())

        return np.array(augmented)

# Example usage
augmenter = DataAugmentation()
sample_images = np.random.rand(10, 784)  # 10 samples
augmented = augmenter.augment_batch(sample_images)
print(f"Augmented {len(augmented)} images")
```

4.2 Aufgabe 4.2: Gradient Clipping

```
class GradientClipper:
    def __init__(self, max_norm=5.0):
```

```
        self.max_norm = max_norm

    def clip_gradients(self, gradients):
        """Clip gradients by global norm"""
        # Calculate global norm
        total_norm = 0
        for grad in gradients.values():
            if isinstance(grad, np.ndarray):
                total_norm += np.sum(grad**2)

        total_norm = np.sqrt(total_norm)

        # Clip if necessary
        if total_norm > self.max_norm:
            clip_ratio = self.max_norm / total_norm
            clipped_gradients = {}
            for key, grad in gradients.items():
                if isinstance(grad, np.ndarray):
                    clipped_gradients[key] = grad * clip_ratio
                else:
                    clipped_gradients[key] = grad
            return clipped_gradients, total_norm

        return gradients, total_norm

    def clip_gradients_by_value(self, gradients, min_val=-1.0,
       max_val=1.0):
        """Clip gradients by value"""
        clipped_gradients = {}
        for key, grad in gradients.items():
            if isinstance(grad, np.ndarray):
                clipped_gradients[key] = np.clip(grad, min_val,
                   max_val)
            else:
                clipped_gradients[key] = grad
        return clipped_gradients

# Example usage with RNN
class RNNWithClipping:
    def __init__(self, input_size, hidden_size, output_size):
        self.W_xh = np.random.randn(hidden_size, input_size) * 0.01
        self.W_hh = np.random.randn(hidden_size, hidden_size) * 0.01
        self.W_hy = np.random.randn(output_size, hidden_size) * 0.01
        self.b_h = np.zeros((hidden_size, 1))
        self.b_y = np.zeros((output_size, 1))

        self.clipper = GradientClipper(max_norm=5.0)

    def forward(self, inputs):
        """Forward pass through RNN"""
        h = np.zeros((self.W_hh.shape[0], 1))
```

```python
        outputs = []
        self.cache = {'inputs': inputs, 'hiddens': [h.copy()]}

        for x in inputs:
            x = x.reshape(-1, 1)
            h = np.tanh(self.W_xh @ x + self.W_hh @ h + self.b_h)
            y = self.W_hy @ h + self.b_y
            outputs.append(y)
            self.cache['hiddens'].append(h.copy())

        return outputs

    def backward(self, doutputs):
        """Backward pass with gradient computation"""
        # Simplified backward pass
        gradients = {
            'W_xh': np.zeros_like(self.W_xh),
            'W_hh': np.zeros_like(self.W_hh),
            'W_hy': np.zeros_like(self.W_hy),
            'b_h': np.zeros_like(self.b_h),
            'b_y': np.zeros_like(self.b_y)
        }

        # Accumulate gradients (simplified)
        for i, dout in enumerate(doutputs):
            gradients['W_hy'] += dout @ self.cache['hiddens'][i+1].T
            gradients['b_y'] += dout
            # ... (weitere Gradienten-Berechnungen)

        # Clip gradients
        clipped_gradients, grad_norm = self.clipper.clip_gradients(
           gradients)

        return clipped_gradients, grad_norm

    def update_weights(self, gradients, learning_rate):
        """Update weights with clipped gradients"""
        self.W_xh -= learning_rate * gradients['W_xh']
        self.W_hh -= learning_rate * gradients['W_hh']
        self.W_hy -= learning_rate * gradients['W_hy']
        self.b_h -= learning_rate * gradients['b_h']
        self.b_y -= learning_rate * gradients['b_y']

print("Gradient Clipping implementiert")
```

Zusammenfassung und Best Practices

Generative Modelle - Vergleich

- **Autoencoders:** Deterministische Kompression, gut für Dimensionsreduktion

- **VAEs:** Probabilistische Generierung, interpretierbare latente Räume
- **GANs:** Hochqualitative Samples, aber Training instabil

Training-Stabilität

- **Data Augmentation:** Erhöht Generalisierung und Robustheit
- **Gradient Clipping:** Verhindert explodierenden Gradienten
- **Proper Initialization:** Xavier/He für stabile Aktivierungen
- **Learning Rate Scheduling:** Adaptive Anpassung während Training

Praktische Tipps

- **VAE Training:** Balance zwischen Rekonstruktion und KL-Loss
- **GAN Training:** Discriminator nicht zu stark trainieren
- **Monitoring:** Visualisierung von generierten Samples
- **Evaluation:** FID, IS für quantitative Bewertung