Deep Learning - Musterlösung Übung 4

Recurrent Neural Networks und LSTM

Fachhochschule Südwestfalen

23. Oktober 2025

Hinweise zur Musterlösung

Diese Musterlösung bietet umfassende mathematische Herleitungen und praktische Implementierungen für RNNs und LSTMs.

1 RNN-Grundlagen - Lösungen

1.1 Aufgabe 1.1: Vanilla RNN Forward Pass

RNN-Gleichungen:

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h) \tag{1}$$
$$y_t = W_{hy}h_t + b_y \tag{2}$$

Gegeben:

$$W_{xh} = \begin{pmatrix} 0.5 & 0.3 \\ -0.2 & 0.4 \end{pmatrix}, \quad W_{hh} = \begin{pmatrix} 0.1 & -0.3 \\ 0.6 & 0.2 \end{pmatrix} \tag{3}$$
$$W_{hy} = \begin{pmatrix} 0.7 & -0.1 \end{pmatrix}, \quad b_h = \begin{pmatrix} 0.1 \\ -0.2 \end{pmatrix}, \quad b_y = 0.3 \tag{4}$$

Sequenz: $x_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, $x_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$, mit $h_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$

Zeitschritt t=1:

$$h_1 = \tanh\left(W_{hh}h_0 + W_{xh}x_1 + b_h\right) \tag{5}$$
$$= \tanh\left(\begin{pmatrix} 0.1 & -0.3 \\ 0.6 & 0.2 \end{pmatrix}\begin{pmatrix} 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0.5 & 0.3 \\ -0.2 & 0.4 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0.1 \\ -0.2 \end{pmatrix}\right) \tag{6}$$
$$= \tanh\left(\begin{pmatrix} 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0.5 \\ -0.2 \end{pmatrix} + \begin{pmatrix} 0.1 \\ -0.2 \end{pmatrix}\right) \tag{7}$$
$$= \tanh\left(\begin{pmatrix} 0.6 \\ -0.4 \end{pmatrix}\right) = \begin{pmatrix} 0.537 \\ -0.380 \end{pmatrix} \tag{8}$$

$$
\begin{aligned}
y_1 &= W_{hy}h_1 + b_y && (9)\\
&= \begin{pmatrix}0.7 & -0.1\end{pmatrix}\begin{pmatrix}0.537\\-0.380\end{pmatrix} + 0.3 && (10)\\
&= 0.7 \cdot 0.537 + (-0.1)\cdot(-0.380) + 0.3 && (11)\\
&= 0.376 + 0.038 + 0.3 = 0.714 && (12)
\end{aligned}
$$

Zeitschritt t=2:

$$
\begin{aligned}
h_2 &= \tanh\left(W_{hh}h_1 + W_{xh}x_2 + b_h\right) && (13)\\
&= \tanh\left(\begin{pmatrix}0.1 & -0.3\\0.6 & 0.2\end{pmatrix}\begin{pmatrix}0.537\\-0.380\end{pmatrix} + \begin{pmatrix}0.5 & 0.3\\-0.2 & 0.4\end{pmatrix}\begin{pmatrix}0\\1\end{pmatrix} + \begin{pmatrix}0.1\\-0.2\end{pmatrix}\right) && (14)\\
&= \tanh\left(\begin{pmatrix}0.168\\0.246\end{pmatrix} + \begin{pmatrix}0.3\\0.4\end{pmatrix} + \begin{pmatrix}0.1\\-0.2\end{pmatrix}\right) && (15)\\
&= \tanh\left(\begin{pmatrix}0.568\\0.446\end{pmatrix}\right) = \begin{pmatrix}0.514\\0.418\end{pmatrix} && (16)
\end{aligned}
$$

$$
\begin{aligned}
y_2 &= W_{hy}h_2 + b_y && (17)\\
&= 0.7 \cdot 0.514 + (-0.1)\cdot 0.418 + 0.3 && (18)\\
&= 0.360 - 0.042 + 0.3 = 0.618 && (19)
\end{aligned}
$$

$\boxed{\text{Outputs: } y_1 = 0.714, \quad y_2 = 0.618}$

1.2 Aufgabe 1.2: Backpropagation Through Time

BPTT-Algorithmus:

Für eine Sequenz der Länge T mit Loss $L = \sum_{t=1}^{T} L_t$:

Output-Gradienten:

$$
\begin{aligned}
\frac{\partial L_t}{\partial y_t} &= \text{loss-spezifisch} && (20)\\
\frac{\partial L_t}{\partial W_{hy}} &= \frac{\partial L_t}{\partial y_t} h_t^T && (21)\\
\frac{\partial L_t}{\partial b_y} &= \frac{\partial L_t}{\partial y_t} && (22)
\end{aligned}
$$

Hidden State Gradienten:

$$
\begin{aligned}
\frac{\partial L_t}{\partial h_t} &= W_{hy}^T \frac{\partial L_t}{\partial y_t} + \frac{\partial L_{t+1}}{\partial h_t} \quad (\text{für } t < T) && (23)\\
\frac{\partial L_T}{\partial h_T} &= W_{hy}^T \frac{\partial L_T}{\partial y_T} \quad (\text{für } t = T) && (24)
\end{aligned}
$$

Rekursive Beziehung:

$$
\begin{aligned}
\frac{\partial L_{t+1}}{\partial h_t} &= \frac{\partial L_{t+1}}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial h_t} && (25)\\
&= \frac{\partial L_{t+1}}{\partial h_{t+1}} W_{hh}^T \text{diag}(1 - h_{t+1}^2) && (26)
\end{aligned}
$$

Parameter-Gradienten:

$$\frac{\partial L}{\partial W_{hh}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial h_t} \text{diag}(1 - h_t^2) h_{t-1}^T \tag{27}$$

$$\frac{\partial L}{\partial W_{xh}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial h_t} \text{diag}(1 - h_t^2) x_t^T \tag{28}$$

$$\frac{\partial L}{\partial b_h} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial h_t} \text{diag}(1 - h_t^2) \tag{29}$$

2 LSTM-Implementierung - Musterlösung

2.1 Aufgabe 2.1: LSTM Forward Pass

LSTM-Gleichungen:

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \quad \text{(Forget Gate)} \tag{30}$$

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \quad \text{(Input Gate)} \tag{31}$$

$$\tilde{C}_t = \tanh(W_C[h_{t-1}, x_t] + b_C) \quad \text{(Candidate Values)} \tag{32}$$

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \quad \text{(Cell State)} \tag{33}$$

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \quad \text{(Output Gate)} \tag{34}$$

$$h_t = o_t \odot \tanh(C_t) \quad \text{(Hidden State)} \tag{35}$$

Implementierung:

```python
import numpy as np

class LSTMCell:
    def __init__(self, input_size, hidden_size):
        self.input_size = input_size
        self.hidden_size = hidden_size

        # Initialize weights (Xavier initialization)
        std = np.sqrt(2.0 / (input_size + hidden_size))

        # Combined weight matrices for efficiency
        self.W_f = np.random.randn(hidden_size, input_size +
            hidden_size) * std
        self.b_f = np.zeros((hidden_size, 1))

        self.W_i = np.random.randn(hidden_size, input_size +
            hidden_size) * std
        self.b_i = np.zeros((hidden_size, 1))

        self.W_C = np.random.randn(hidden_size, input_size +
            hidden_size) * std
        self.b_C = np.zeros((hidden_size, 1))

        self.W_o = np.random.randn(hidden_size, input_size +
            hidden_size) * std
```

```
        self.b_o = np.zeros((hidden_size, 1))

        # For backpropagation
        self.cache = {}

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

    def forward(self, x, h_prev, C_prev):
        """LSTM forward pass"""
        # Concatenate input and previous hidden state
        concat = np.vstack([h_prev, x])

        # Forget gate
        f = self.sigmoid(self.W_f @ concat + self.b_f)

        # Input gate
        i = self.sigmoid(self.W_i @ concat + self.b_i)

        # Candidate values
        C_tilde = np.tanh(self.W_C @ concat + self.b_C)

        # Cell state
        C = f * C_prev + i * C_tilde

        # Output gate
        o = self.sigmoid(self.W_o @ concat + self.b_o)

        # Hidden state
        h = o * np.tanh(C)

        # Cache for backward pass
        self.cache = {
            'x': x, 'h_prev': h_prev, 'C_prev': C_prev,
            'concat': concat, 'f': f, 'i': i, 'C_tilde': C_tilde,
            'C': C, 'o': o, 'h': h
        }

        return h, C

    def backward(self, dh, dC):
        """LSTM backward pass"""
        cache = self.cache

        # Output gate gradients
        do = dh * np.tanh(cache['C'])
        dC += dh * cache['o'] * (1 - np.tanh(cache['C'])**2)

```

```
        # Cell state gradients
        dC_tilde = dC * cache['i']
        di = dC * cache['C_tilde']
        df = dC * cache['C_prev']
        dC_prev = dC * cache['f']

        # Gate gradients (before activation)
        do_raw = do * cache['o'] * (1 - cache['o'])
        di_raw = di * cache['i'] * (1 - cache['i'])
        df_raw = df * cache['f'] * (1 - cache['f'])
        dC_tilde_raw = dC_tilde * (1 - cache['C_tilde']**2)

        # Weight gradients
        dW_o = do_raw @ cache['concat'].T
        db_o = do_raw

        dW_i = di_raw @ cache['concat'].T
        db_i = di_raw

        dW_f = df_raw @ cache['concat'].T
        db_f = df_raw

        dW_C = dC_tilde_raw @ cache['concat'].T
        db_C = dC_tilde_raw

        # Input gradients
        dconcat = (self.W_o.T @ do_raw + self.W_i.T @ di_raw +
                   self.W_f.T @ df_raw + self.W_C.T @ dC_tilde_raw)

        dh_prev = dconcat[:self.hidden_size]
        dx = dconcat[self.hidden_size:]

        # Store gradients
        self.dW_o, self.db_o = dW_o, db_o
        self.dW_i, self.db_i = dW_i, db_i
        self.dW_f, self.db_f = dW_f, db_f
        self.dW_C, self.db_C = dW_C, db_C

        return dx, dh_prev, dC_prev

class LSTM:
    def __init__(self, input_size, hidden_size, output_size,
       num_layers=1):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.num_layers = num_layers

        # LSTM layers
        self.lstm_layers = []
        for i in range(num_layers):
```

```
            layer_input_size = input_size if i == 0 else hidden_size
            self.lstm_layers.append(LSTMCell(layer_input_size, hidden_size))

        # Output layer
        self.W_out = np.random.randn(output_size, hidden_size) * 0.1
        self.b_out = np.zeros((output_size, 1))

        # For storing states
        self.hidden_states = []
        self.cell_states = []

    def forward(self, inputs):
        """Forward pass through LSTM"""
        batch_size = inputs.shape[1] if len(inputs.shape) > 1 else 1
        seq_length = len(inputs)

        # Initialize states
        h = [np.zeros((self.hidden_size, batch_size)) for _ in range(self.num_layers)]
        C = [np.zeros((self.hidden_size, batch_size)) for _ in range(self.num_layers)]

        self.hidden_states = []
        self.cell_states = []
        outputs = []

        # Process sequence
        for t in range(seq_length):
            x = inputs[t].reshape(-1, 1) if inputs[t].ndim == 1 else inputs[t]

            # Forward through LSTM layers
            for layer in range(self.num_layers):
                h[layer], C[layer] = self.lstm_layers[layer].forward(x, h[layer], C[layer])
                x = h[layer]  # Output becomes input for next layer

            # Store states
            self.hidden_states.append([h_layer.copy() for h_layer in h])
            self.cell_states.append([C_layer.copy() for C_layer in C])

            # Output layer
            output = self.W_out @ h[-1] + self.b_out
            outputs.append(output)

        return outputs

    def backward(self, doutputs):
```

```
        """Backward pass through LSTM"""
        seq_length = len(doutputs)

        # Initialize gradients
        dh = [np.zeros_like(self.hidden_states[0][layer]) for layer in range(self.num_layers)]
        dC = [np.zeros_like(self.cell_states[0][layer]) for layer in range(self.num_layers)]

        # Output layer gradients
        dW_out = np.zeros_like(self.W_out)
        db_out = np.zeros_like(self.b_out)

        # Backward through time
        for t in reversed(range(seq_length)):
            # Output layer gradients
            dout = doutputs[t]
            dW_out += dout @ self.hidden_states[t][-1].T
            db_out += dout

            # LSTM layer gradients
            dh[-1] += self.W_out.T @ dout

            # Backward through LSTM layers
            for layer in reversed(range(self.num_layers)):
                if t == 0:
                    h_prev = np.zeros_like(self.hidden_states[t][layer])
                    C_prev = np.zeros_like(self.cell_states[t][layer])
                else:
                    h_prev = self.hidden_states[t-1][layer]
                    C_prev = self.cell_states[t-1][layer]

                # Backward through LSTM cell
                dx, dh_prev, dC_prev = self.lstm_layers[layer].backward(dh[layer], dC[layer])

                if layer > 0:
                    dh[layer-1] = dx

                if t > 0:
                    dh[layer] = dh_prev
                    dC[layer] = dC_prev
                else:
                    dh[layer] = np.zeros_like(dh[layer])
                    dC[layer] = np.zeros_like(dC[layer])

        # Store output layer gradients
        self.dW_out = dW_out
        self.db_out = db_out
```

```python

        return dh, dC
```

2.2 Aufgabe 2.2: Numerisches Beispiel

Vereinfachtes LSTM mit kleinen Dimensionen:

```python
# Test LSTM with simple sequence
def test_lstm():
    # Simple sequence: [1, 0], [0, 1], [1, 1]
    inputs = [np.array([[1], [0]]), np.array([[0], [1]]), np.array
        ([[1], [1]])]

    # Create LSTM
    lstm = LSTM(input_size=2, hidden_size=3, output_size=1)

    # Forward pass
    outputs = lstm.forward(inputs)

    print("LSTM Outputs:")
    for t, output in enumerate(outputs):
        print(f"t={t}: {output.flatten()}")

    # Simple loss (mean squared error with target = 1)
    loss = 0
    doutputs = []
    for output in outputs:
        target = np.array([[1]])  # Simple target
        loss += 0.5 * np.sum((output - target)**2)
        doutput = output - target
        doutputs.append(doutput)

    print(f"Loss: {loss}")

    # Backward pass
    lstm.backward(doutputs)

    return lstm, outputs, loss

# Run test
lstm, outputs, loss = test_lstm()
```

3 Sequenz-Modellierung - Lösungen

3.1 Aufgabe 3.1: Sprachmodellierung

Character-Level Language Model:

```python
class CharRNN:
    def __init__(self, vocab_size, hidden_size=100, seq_length=25):
```

```
        self.vocab_size = vocab_size
        self.hidden_size = hidden_size
        self.seq_length = seq_length

        # LSTM for sequence modeling
        self.lstm = LSTM(vocab_size, hidden_size, vocab_size)

        # Character to index mapping
        self.char_to_idx = {}
        self.idx_to_char = {}

    def prepare_data(self, text):
        """Prepare character-level data"""
        chars = list(set(text))
        self.char_to_idx = {ch: i for i, ch in enumerate(chars)}
        self.idx_to_char = {i: ch for i, ch in enumerate(chars)}
        self.vocab_size = len(chars)

        # Convert text to indices
        data = [self.char_to_idx[ch] for ch in text]
        return data

    def create_sequences(self, data):
        """Create input-target pairs"""
        inputs, targets = [], []

        for i in range(0, len(data) - self.seq_length, self.seq_length):
            input_seq = data[i:i + self.seq_length]
            target_seq = data[i + 1:i + self.seq_length + 1]

            # One-hot encoding
            input_onehot = np.zeros((self.seq_length, self.vocab_size))
            target_onehot = np.zeros((self.seq_length, self.vocab_size))

            for t, (inp, tar) in enumerate(zip(input_seq, target_seq)):
                input_onehot[t, inp] = 1
                target_onehot[t, tar] = 1

            inputs.append(input_onehot)
            targets.append(target_onehot)

        return inputs, targets

    def train_step(self, input_seq, target_seq, learning_rate=0.01):
        """Single training step"""
        # Forward pass
        outputs = self.lstm.forward(input_seq)
```

```python

        # Compute loss (cross-entropy)
        loss = 0
        doutputs = []

        for t, (output, target) in enumerate(zip(outputs, target_seq)):
            # Softmax
            exp_output = np.exp(output - np.max(output))
            probs = exp_output / np.sum(exp_output)

            # Cross-entropy loss
            loss += -np.sum(target * np.log(probs + 1e-8))

            # Gradient
            doutput = probs - target
            doutputs.append(doutput)

        # Backward pass
        self.lstm.backward(doutputs)

        # Update weights
        self.update_weights(learning_rate)

        return loss / len(outputs)

    def update_weights(self, learning_rate):
        """Update LSTM weights"""
        # Update LSTM layers
        for layer in self.lstm.lstm_layers:
            layer.W_f -= learning_rate * layer.dW_f
            layer.b_f -= learning_rate * layer.db_f
            layer.W_i -= learning_rate * layer.dW_i
            layer.b_i -= learning_rate * layer.db_i
            layer.W_C -= learning_rate * layer.dW_C
            layer.b_C -= learning_rate * layer.db_C
            layer.W_o -= learning_rate * layer.dW_o
            layer.b_o -= learning_rate * layer.db_o

        # Update output layer
        self.lstm.W_out -= learning_rate * self.lstm.dW_out
        self.lstm.b_out -= learning_rate * self.lstm.db_out

    def generate_text(self, seed_char, length=100, temperature=1.0):
        """Generate text starting from seed character"""
        generated = [seed_char]

        # Initialize hidden and cell states
        h = np.zeros((self.hidden_size, 1))
        C = np.zeros((self.hidden_size, 1))

```

```python
        for _ in range(length):
            # Prepare input
            char_idx = self.char_to_idx[generated[-1]]
            x = np.zeros((self.vocab_size, 1))
            x[char_idx, 0] = 1

            # Forward pass
            h, C = self.lstm.lstm_layers[0].forward(x, h, C)
            output = self.lstm.W_out @ h + self.lstm.b_out

            # Apply temperature
            output = output / temperature

            # Softmax sampling
            exp_output = np.exp(output - np.max(output))
            probs = exp_output / np.sum(exp_output)

            # Sample next character
            next_idx = np.random.choice(self.vocab_size, p=probs.flatten())
            next_char = self.idx_to_char[next_idx]
            generated.append(next_char)

        return ''.join(generated)

# Example usage
text = "hello world this is a simple example for character level language modeling"
char_rnn = CharRNN(vocab_size=0, seq_length=10)

# Prepare data
data = char_rnn.prepare_data(text)
inputs, targets = char_rnn.create_sequences(data)

print(f"Vocabulary size: {char_rnn.vocab_size}")
print(f"Number of sequences: {len(inputs)}")

# Train for a few steps
for epoch in range(10):
    total_loss = 0
    for inp, tar in zip(inputs, targets):
        loss = char_rnn.train_step(inp, tar, learning_rate=0.1)
        total_loss += loss

    avg_loss = total_loss / len(inputs)
    print(f"Epoch {epoch+1}, Average Loss: {avg_loss:.4f}")

# Generate text
generated = char_rnn.generate_text('h', length=50)
print(f"Generated text: {generated}")
```

3.2 Aufgabe 3.2: Zeitreihenvorhersage

LSTM für Zeitreihen:

```python
class TimeSeriesLSTM:
    def __init__(self, input_size=1, hidden_size=50, output_size=1,
        num_layers=2):
        self.lstm = LSTM(input_size, hidden_size, output_size,
            num_layers)
        self.scaler_X = None
        self.scaler_y = None

    def create_sequences(self, data, seq_length, forecast_horizon=1):
        """Create sequences for time series prediction"""
        X, y = [], []

        for i in range(len(data) - seq_length - forecast_horizon + 1)
            :
            sequence = data[i:i + seq_length]
            target = data[i + seq_length:i + seq_length +
                forecast_horizon]
            X.append(sequence)
            y.append(target)

        return np.array(X), np.array(y)

    def normalize_data(self, X, y):
        """Normalize input and output data"""
        # Simple min-max normalization
        X_min, X_max = X.min(), X.max()
        y_min, y_max = y.min(), y.max()

        X_norm = (X - X_min) / (X_max - X_min)
        y_norm = (y - y_min) / (y_max - y_min)

        self.scaler_X = (X_min, X_max)
        self.scaler_y = (y_min, y_max)

        return X_norm, y_norm

    def train(self, X, y, epochs=100, learning_rate=0.01):
        """Train the time series model"""
        losses = []

        for epoch in range(epochs):
            epoch_loss = 0

            for i in range(len(X)):
                # Prepare sequence
                sequence = X[i].reshape(-1, 1, 1)  # (seq_len, batch,
                    features)
                target = y[i].reshape(-1, 1)
```

```python

                # Forward pass
                outputs = self.lstm.forward(sequence)

                # Only use last output for prediction
                prediction = outputs[-1]

                # Mean squared error loss
                loss = 0.5 * np.sum((prediction - target)**2)
                epoch_loss += loss

                # Backward pass
                doutputs = [np.zeros_like(out) for out in outputs]
                doutputs[-1] = prediction - target

                self.lstm.backward(doutputs)

                # Update weights
                self.update_weights(learning_rate)

            avg_loss = epoch_loss / len(X)
            losses.append(avg_loss)

            if (epoch + 1) % 10 == 0:
                print(f"Epoch {epoch+1}/{epochs}, Loss: {avg_loss:.6f}")

        return losses

    def update_weights(self, learning_rate):
        """Update model weights"""
        for layer in self.lstm.lstm_layers:
            layer.W_f -= learning_rate * layer.dW_f
            layer.b_f -= learning_rate * layer.db_f
            layer.W_i -= learning_rate * layer.dW_i
            layer.b_i -= learning_rate * layer.db_i
            layer.W_C -= learning_rate * layer.dW_C
            layer.b_C -= learning_rate * layer.db_C
            layer.W_o -= learning_rate * layer.dW_o
            layer.b_o -= learning_rate * layer.db_o

        self.lstm.W_out -= learning_rate * self.lstm.dW_out
        self.lstm.b_out -= learning_rate * self.lstm.db_out

    def predict(self, sequence):
        """Make prediction for a sequence"""
        sequence = sequence.reshape(-1, 1, 1)
        outputs = self.lstm.forward(sequence)
        return outputs[-1].flatten()

    def denormalize(self, normalized_value, is_target=True):
```

```
        """Denormalize predicted values"""
        if is_target:
            min_val, max_val = self.scaler_y
        else:
            min_val, max_val = self.scaler_X

        return normalized_value * (max_val - min_val) + min_val

# Generate synthetic time series data
def generate_sine_wave(length=1000, frequency=0.02, noise=0.1):
    t = np.arange(length)
    signal = np.sin(2 * np.pi * frequency * t) + noise * np.random.
        randn(length)
    return signal

# Example usage
data = generate_sine_wave(500)
seq_length = 20

# Create sequences
ts_lstm = TimeSeriesLSTM(input_size=1, hidden_size=30)
X, y = ts_lstm.create_sequences(data, seq_length)

# Normalize data
X_norm, y_norm = ts_lstm.normalize_data(X, y)

# Split into train/test
split_idx = int(0.8 * len(X_norm))
X_train, X_test = X_norm[:split_idx], X_norm[split_idx:]
y_train, y_test = y_norm[:split_idx], y_norm[split_idx:]

# Train model
print("Training Time Series LSTM...")
losses = ts_lstm.train(X_train, y_train, epochs=50, learning_rate
    =0.01)

# Test predictions
predictions = []
for i in range(len(X_test)):
    pred = ts_lstm.predict(X_test[i])
    predictions.append(pred[0])

# Denormalize predictions
predictions_denorm = [ts_lstm.denormalize(pred) for pred in
    predictions]
targets_denorm = [ts_lstm.denormalize(y_test[i][0]) for i in range(
    len(y_test))]

# Calculate RMSE
rmse = np.sqrt(np.mean((np.array(predictions_denorm) - np.array(
    targets_denorm))**2))
```

```python
print(f"Test RMSE: {rmse:.4f}")
```

4 Vertiefende Fragen - Lösungen

4.1 Aufgabe 4.1: Gradient-Probleme

Vanishing Gradient in RNNs:

Das Vanishing Gradient Problem tritt auf, wenn Gradienten durch viele Zeitschritte propagiert werden:

$$\frac{\partial L}{\partial h_1} = \frac{\partial L}{\partial h_T} \prod_{t=2}^{T} \frac{\partial h_t}{\partial h_{t-1}} \tag{36}$$

Für RNNs mit Tanh-Aktivierung:

$$\frac{\partial h_t}{\partial h_{t-1}} = W_{hh}^T \text{diag}(\frac{\partial \tanh(z_t)}{\partial z_t}) = W_{hh}^T \text{diag}(1 - h_t^2) \tag{37}$$

Da $|1 - h_t^2| \leq 1$ und typischerweise $\|W_{hh}\| < 1$ für Stabilität, wird das Produkt exponentiell klein:

$$\left\| \prod_{t=2}^{T} \frac{\partial h_t}{\partial h_{t-1}} \right\| \leq \|W_{hh}\|^{T-1} \to 0 \text{ für } T \to \infty \tag{38}$$

LSTM-Lösung:

LSTMs lösen dies durch: 1. **Cell State Highway:** $C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$ 2. **Additive Updates:** Vermeidung wiederholter Multiplikationen 3. **Forget Gate Control:** Selektive Informationserhaltung

$$\frac{\partial C_t}{\partial C_{t-1}} = f_t \quad \text{(keine Matrixmultiplikation)} \tag{39}$$

4.2 Aufgabe 4.2: Attention Mechanism

Einfacher Attention-Mechanismus:

```python
class SimpleAttention:
    def __init__(self, hidden_size):
        self.hidden_size = hidden_size
        self.W_a = np.random.randn(hidden_size, hidden_size) * 0.1
        self.v_a = np.random.randn(hidden_size, 1) * 0.1

    def forward(self, encoder_outputs, decoder_hidden):
        """
        encoder_outputs: (seq_len, hidden_size)
        decoder_hidden: (hidden_size, 1)
        """
```

```
        seq_len = encoder_outputs.shape[0]

        # Compute attention scores
        scores = np.zeros(seq_len)
        for i, h_enc in enumerate(encoder_outputs):
            h_enc = h_enc.reshape(-1, 1)

            # Additive attention
            energy = np.tanh(self.W_a @ (h_enc + decoder_hidden))
            score = self.v_a.T @ energy
            scores[i] = score.item()

        # Softmax to get attention weights
        exp_scores = np.exp(scores - np.max(scores))
        attention_weights = exp_scores / np.sum(exp_scores)

        # Compute context vector
        context = np.zeros((self.hidden_size, 1))
        for i, weight in enumerate(attention_weights):
            context += weight * encoder_outputs[i].reshape(-1, 1)

        return context, attention_weights

# Example usage
attention = SimpleAttention(hidden_size=4)

# Sample encoder outputs and decoder hidden state
encoder_outputs = np.random.randn(5, 4)  # 5 time steps, 4 hidden
    units
decoder_hidden = np.random.randn(4, 1)

context, weights = attention.forward(encoder_outputs, decoder_hidden)

print("Attention weights:", weights)
print("Context shape:", context.shape)
```

Attention-Mathematik:

$$e_{t,i} = v_a^T \tanh(W_a h_t + U_a s_i) \tag{40}$$

$$\alpha_{t,i} = \frac{\exp(e_{t,i})}{\sum_{j=1}^{T} \exp(e_{t,j})} \tag{41}$$

$$c_t = \sum_{i=1}^{T} \alpha_{t,i} h_i \tag{42}$$

Zusammenfassung und Praktische Tipps

RNN/LSTM Best Practices

- **Gradient Clipping:** $\|\nabla\| > \theta \Rightarrow \nabla = \theta \frac{\nabla}{\|\nabla\|}$

- **Proper Initialization:** Xavier für Gates, Zero für Biases
- **Learning Rate Scheduling:** Reduce on plateau
- **Dropout:** Zwischen LSTM-Schichten, nicht innerhalb

Sequenz-Modellierung Strategien

- **Teacher Forcing:** Training mit Ground Truth
- **Curriculum Learning:** Einfache → komplexe Sequenzen
- **Beam Search:** Bessere Inferenz für Generierung
- **Attention:** Für lange Sequenzen unerlässlich