LSTM implementation optimization techniques

Optimized sample implementation of an LSTM layer from scratch using NumPy:

```
import numpy as np

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

        # Initialize weights and biases
        self.W = np.random.randn(4 * hidden_size, input_size) * 0.01
        self.U = np.random.randn(4 * hidden_size, hidden_size) * 0.01
        self.b = np.zeros((4 * hidden_size, 1))

        # Initialize parameters for Adam optimizer
        self.m = np.zeros_like(self.W)
        self.v = np.zeros_like(self.W)
        self.t = 0

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

    def tanh(self, x):
        return np.tanh(x)

    def forward(self, x, h_prev, c_prev):
        seq_len, batch_size, _ = x.shape

        h = np.zeros((seq_len, batch_size, self.hidden_size))
        c = np.zeros((seq_len, batch_size, self.hidden_size))
        gates = np.zeros((seq_len, batch_size, 4 * self.hidden_size))

        for t in range(seq_len):
            # Concatenate input and previous hidden state
            xh = np.row_stack((x[t].T, h_prev.T)).T

            # Compute all gates at once
            gates[t] = np.dot(self.W, xh.T) + np.dot(self.U, h_prev.T) + self.b
            gates_split = np.split(gates[t], 4, axis=0)

            # Apply activations
            i = self.sigmoid(gates_split[0])
            f = self.sigmoid(gates_split[1])
            o = self.sigmoid(gates_split[2])
            g = self.tanh(gates_split[3])

            # Update cell state
            c[t] = f * c_prev + i * g

            # Update hidden state
            h[t] = o * self.tanh(c[t])

```

```python
            # Update previous states for next iteration
            h_prev = h[t]
            c_prev = c[t]

        self.cache = (x, h, c, gates)
        return h, c[-1]

    def backward(self, dh, dc):
        x, h, c, gates = self.cache
        seq_len, batch_size, _ = x.shape

        dx = np.zeros_like(x)
        dW = np.zeros_like(self.W)
        dU = np.zeros_like(self.U)
        db = np.zeros_like(self.b)

        dh_prev = np.zeros((batch_size, self.hidden_size))
        dc_prev = dc

        for t in reversed(range(seq_len)):
            # Get gates for current timestep
            gates_t = gates[t]
            gates_split = np.split(gates_t, 4, axis=0)
            i, f, o, g = [self.sigmoid(g) for g in gates_split[:3]] + [self.tanh(gates_split[3])]

            # Backprop through tanh in hidden state
            dtanh = dh_prev * o + dc_prev * (1 - self.tanh(c[t])**2)

            # Backprop through cell state
            dc_prev = dtanh * f
            di = dtanh * g * i * (1 - i)
            df = dtanh * c[t-1] * f * (1 - f)
            do = dh_prev * self.tanh(c[t]) * o * (1 - o)
            dg = dtanh * i * (1 - g**2)

            # Concatenate gradients
            dgate = np.concatenate((di, df, do, dg), axis=0)

            # Compute parameter gradients
            xh = np.row_stack((x[t].T, h[t-1].T)).T
            dW += np.dot(dgate, xh.T)
            dU += np.dot(dgate, h[t-1].T)
            db += dgate.sum(axis=1, keepdims=True)

            # Compute input gradient
            dx[t] = np.dot(self.W.T, dgate)[:self.input_size].T

            # Update hidden state gradient for previous timestep
            dh_prev = np.dot(self.U.T, dgate) + dh[t]

        # Clip gradients to prevent explosion
        for grad in [dW, dU, db, dx]:
            np.clip(grad, -5, 5, out=grad)

        return dx, dW, dU, db

    def optimize(self, dW, dU, db, learning_rate=0.001, beta1=0.9, beta2=0.999):
```

```python
        self.t += 1
        self.m = beta1 * self.m + (1 - beta1) * dW
        self.v = beta2 * self.v + (1 - beta2) * (dW**2)
        m_hat = self.m / (1 - beta1**self.t)
        v_hat = self.v / (1 - beta2**self.t)

        self.W -= learning_rate * m_hat / (np.sqrt(v_hat) + 1e-8)
        self.U -= learning_rate * dU
        self.b -= learning_rate * db

# Example usage:
input_size = 10
hidden_size = 20
seq_len = 5
batch_size = 32

lstm = LSTM(input_size, hidden_size)
x = np.random.randn(seq_len, batch_size, input_size)
h0 = np.zeros((batch_size, hidden_size))
c0 = np.zeros((batch_size, hidden_size))

h, c = lstm.forward(x, h0, c0)
dh = np.random.randn(seq_len, batch_size, hidden_size)
dx, dW, dU, db = lstm.backward(dh, np.zeros((batch_size, hidden_size)))
lstm.optimize(dW, dU, db)
```

Optimization Techniques Implemented:

1. **Vectorization:**
   - All matrix operations are vectorized using NumPy
   - Batch processing for multiple sequences in parallel
2. **Gradient Clipping:**
   - Gradients are clipped to the range [-5, 5] to prevent explosion
3. **Adam Optimizer:**
   - Adaptive moment estimation with bias correction
   - Maintains separate momentum (m) and variance (v) parameters
4. **Efficient Memory Management:**
   - Pre-allocates memory for hidden states and gates
   - Reuses intermediate values during forward/backward passes

Additional Optimization Techniques (not implemented here but recommended):

1. **Zoneout Regularization:**
   - Randomly preserve hidden units during training
2. **Peephole Connections:**
   - Allow gates to access cell state directly
3. **Layer Normalization:**
   - Normalize activations for more stable training
4. **CUDNN LSTM:**
   - GPU-accelerated implementation using cuDNN library

5. **Sparse Activation Pruning:**
   - Remove inactive neurons to reduce computation
6. **Quantization:**
   - Use lower precision (FP16/INT8) for parameters and activations
7. **Kernel Fusion:**
   - Combine multiple operations into single kernel launches
8. **Gradient Checkpointing:**
   - Trade computation for memory by recomputing activations during backward pass

Implementation effectively:

1. Initialize with proper weight scaling (Xavier/Glorot initialization)
2. Use batch normalization for input data
3. Implement proper learning rate scheduling
4. Use teacher forcing during training
5. Implement proper weight initialization strategies

This implementation provides a foundation that can be extended with these additional optimizations based on specific use case requirements.