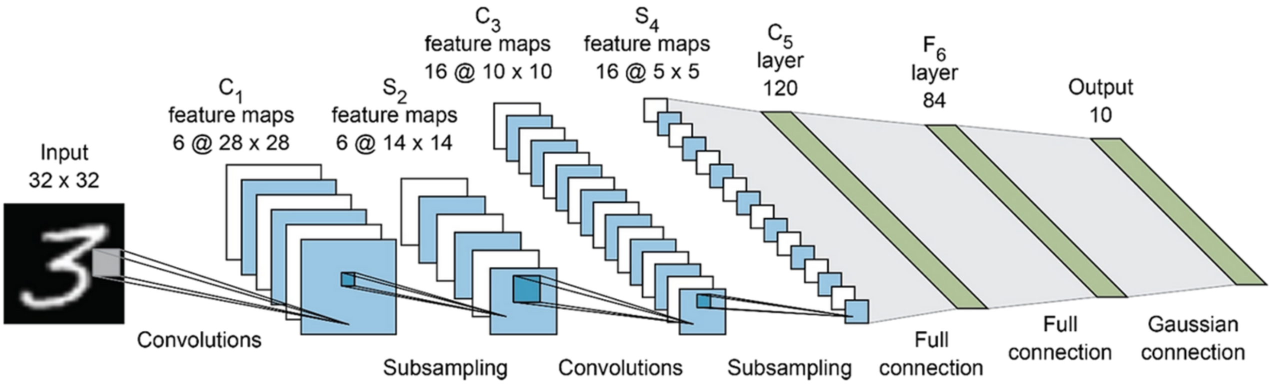
Explanation of LeNet CNN Forward and Backward Pass

LeNet Architecture Overview

LeNet is a classic convolution neural network architecture designed for handwritten digit recognition. It consists of:

1. Convolution layers for feature extraction
2. Activation functions (ReLU) for non-linearity
3. Pooling layers for dimensionality reduction
4. Fully connected layers for classification



Layer-by-Layer Explanation

1. Convolution Layer

Intuitive Idea:
The convolution layer is designed to automatically detect local patterns in the input data, such as edges, textures, or more complex shapes in images. By applying multiple filters, the network can learn hierarchical representations of the input.

Forward Pass:
The convolution operation involves sliding filters over the input to produce feature maps:

$$O_{i,j}^{(c)} = \sum_{m=0}^{f-1} \sum_{n=0}^{f-1} I_{i+m,j+n}^{(k)} \cdot K_{m,n}^{(c,k)} + b^{(c)}$$

where:

- I is the input feature map
- K is the convolution kernel
- b is the bias term
- f is the filter size
- c is the output channel index
- k is the input channel index

Backward Pass:
The gradients are computed using the chain rule:

$$\frac{\partial L}{\partial K^{(c,k)}} = I^{(k)} * \text{rot180}\left(\frac{\partial L}{\partial O^{(c)}}\right)$$

$$\frac{\partial L}{\partial I^{(k)}} = \sum_c \text{rot180}(K^{(c,k)}) * \frac{\partial L}{\partial O^{(c)}}$$

$$\frac{\partial L}{\partial b^{(c)}} = \sum_{i,j} \frac{\partial L}{\partial O_{i,j}^{(c)}}$$

2. Batch Normalization Layer

Intuitive Idea:
Batch normalization aims to stabilize and accelerate the training process by normalizing the inputs to each layer, reducing internal covariate shift. It also provides a regularizing effect.

Forward Pass:
Batch normalization normalizes the input, scales, and shifts it:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$y_i = \gamma \hat{x}_i + \beta$$

Backward Pass:
The gradients are computed as:

$$\frac{\partial L}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial L}{\partial y_i}$$

$$\frac{\partial L}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial L}{\partial y_i}\hat{x}_i$$

$$\frac{\partial L}{\partial \hat{x}_i} = \gamma \frac{\partial L}{\partial y_i}$$

$$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial \hat{x}_i} \cdot \frac{1}{\sqrt{\sigma_B^2 + \epsilon}}\left(1 - \frac{1}{m} - \frac{(x_i - \mu_B)^2}{m(\sigma_B^2 + \epsilon)}\right)$$

Explanation of Batch Normalization Layer

Batch normalization is a technique designed to improve the training of neural networks by normalizing the inputs to each layer. This normalization helps to:

- Reduce internal covariate shift (changes in the distribution of layer inputs during training)
- Accelerate training by allowing higher learning rates
- Provide some regularization effect
- Make the network more robust to initialization

Forward Pass Formulas

1. Compute Batch Mean

The mean of the inputs for each feature map across the batch is calculated:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$$

where:

- m is the number of samples in the batch
- x_i represents the input for each sample in the batch

2. Compute Batch Variance

The variance of the inputs for each feature map across the batch is calculated:

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

This measures how much the inputs vary from the mean.

3. Normalize Inputs

The inputs are normalized using the computed mean and variance:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

where:

- ϵ is a small constant (e.g., 10^{-5}) added for numerical stability

4. Scale and Shift

The normalized inputs are scaled and shifted using learnable parameters:

$$y_i = \gamma \hat{x}_i + \beta$$

where:

- γ (scale parameter) and β (shift parameter) are learned during training
- These parameters allow the network to recover the original representation if needed

5. Running Mean and Variance (for Inference)

During training, we maintain running estimates of the mean and variance to use during inference:

$$\mu_{\text{running}} = \text{momentum} \cdot \mu_{\text{running}} + (1 - \text{momentum}) \cdot \mu_B$$

$$\sigma_{\text{running}}^2 = \text{momentum} \cdot \sigma_{\text{running}}^2 + (1 - \text{momentum}) \cdot \sigma_B^2$$

where:

- momentum is typically a value close to 1 (e.g., 0.9)

Backward Pass Formulas

1. Gradient with respect to Beta (β)

The gradient for the shift parameter β is simply the sum of the gradients from the output:

$$\frac{\partial L}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial L}{\partial y_i}$$

2. Gradient with respect to Gamma (γ)

The gradient for the scale parameter γ involves the normalized inputs:

$$\frac{\partial L}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial L}{\partial y_i} \hat{x}_i$$

3. Gradient with respect to Normalized Inputs ($\hat{x}$)

The gradient propagates through the scaling operation:

$$\frac{\partial L}{\partial \hat{x}_i} = \gamma \frac{\partial L}{\partial y_i}$$

4. Gradient with respect to Variance (σ_B^2)

The gradient with respect to the variance involves the deviation of each input from the mean:

$$\frac{\partial L}{\partial \sigma_B^2} = \sum_{i=1}^{m} \frac{\partial L}{\partial \hat{x}_i} \cdot \frac{-1}{2}(x_i - \mu_B)(\sigma_B^2 + \epsilon)^{-3/2}$$

5. Gradient with respect to Mean (μ_B)

The gradient with respect to the mean has two components:

$$\frac{\partial L}{\partial \mu_B} = \sum_{i=1}^{m} \frac{\partial L}{\partial \hat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_B^2 + \epsilon}} + \frac{\partial L}{\partial \sigma_B^2} \cdot \frac{-2}{m} \sum_{j=1}^{m} (x_j - \mu_B)$$

6. Gradient with respect to Inputs (x_i)

The final gradient with respect to the inputs combines the gradients from all previous steps:

$$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial \hat{x}_i} \cdot \frac{1}{\sqrt{\sigma_B^2 + \epsilon}} + \frac{\partial L}{\partial \sigma_B^2} \cdot \frac{2}{m}(x_i - \mu_B) + \frac{\partial L}{\partial \mu_B} \cdot \frac{1}{m}$$

Implementation Notes

- During training, we use the batch mean and variance for normalization.
- During inference (testing), we use the running mean and variance estimates.
- The parameters γ and β are updated during back propagation like other network weights.
- Batch normalization is typically applied after convolutional or fully connected layers but before activation functions.

3. ReLU Activation

Intuitive Idea:
The Rectified Linear Unit (ReLU) activation function introduces non-linearity to the network while being computationally efficient. It helps the network learn complex patterns by allowing it to model non-linear relationships.

Forward Pass:
The ReLU activation function is applied element-wise:

$$f(x) = \max(0, x)$$

Backward Pass:
The gradient is computed as:

$$\frac{\partial L}{\partial x} = \begin{cases} \frac{\partial L}{\partial y} & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

4. Max Pooling Layer

Intuitive Idea:
Max pooling reduces the spatial dimensions of the feature maps, making the network more computationally efficient and helping to prevent overfitting. It also provides a form of translation invariance.

Forward Pass:
Max pooling takes the maximum value within each pooling window:

$$O_{i,j} = \max_{m,n \in \text{pooling window}} I_{i+m,j+n}$$

Backward Pass:
The gradient is propagated only to the positions of the maximum values:

$$\frac{\partial L}{\partial I_{i,j}} = \begin{cases} \frac{\partial L}{\partial O_{k,l}} & \text{if } I_{i,j} = \max \text{ in window} \\ 0 & \text{otherwise} \end{cases}$$

5. Fully Connected Layer

Intuitive Idea:
Fully connected layers take the high-level features extracted by the convolutional layers and combine them to make predictions. They perform classification based on the learned features.

Forward Pass:
The fully connected layer performs matrix multiplication:

$$y = Wx + b$$

Backward Pass:
The gradients are computed using matrix operations:

$$\frac{\partial L}{\partial W} = x \cdot \frac{\partial L}{\partial y}^T$$

$$\frac{\partial L}{\partial b} = \sum_{i=1}^{m} \frac{\partial L}{\partial y_i}$$

$$\frac{\partial L}{\partial x} = W^T \cdot \frac{\partial L}{\partial y}$$

Complete Forward and Backward Pass for LeNet

Forward Pass:

1. Input image passes through the first convolutional layer
2. Batch normalization is applied to the output
3. ReLU activation introduces non-linearity
4. Max pooling reduces spatial dimensions
5. The process repeats for subsequent layers
6. Flattened output is passed through fully connected layers
7. Softmax activation produces probability distribution for classification

Backward Pass:

1. Gradients start from the output layer (usually computed using cross-entropy loss)
2. Gradients propagate backward through fully connected layers
3. Max pooling layer propagates gradients only to max positions
4. ReLU layer passes gradients where activations were positive
5. Batch normalization layer updates its parameters (gamma and beta)
6. Convolution layers update their filters and biases
7. Gradients continue propagating backward through all layers

---

Dropout Layer

Intuitive Idea

The dropout layer is a regularization technique used in neural networks to prevent overfitting. The core idea is to randomly "drop out" a certain percentage of neurons during the training phase. This forces the network to learn more robust features and prevents it from relying too much on any single neuron, thereby improving generalization.

Forward Pass

During the forward pass, each neuron has a probability `p` of being dropped. The remaining neurons' outputs are scaled by a factor of `1/(1-p)` to maintain the expected sum of outputs. This scaling ensures that the magnitudes of the activations remain consistent across training and inference phases.

Formula:
For a neuron's output `a`, the forward pass with dropout can be represented as:

```
a' = (a * mask) / (1 - p)
```

where `mask` is a binary tensor of the same shape as `a`, with elements set to 1 with probability `(1-p)` and 0 with probability `p`.

Backward Pass

During the backward pass, gradients are only propagated through the neurons that were not dropped out during the forward pass. The gradients for the dropped neurons are zeroed out. This means that only the active neurons (those not dropped) contribute to the weight updates.

Formula:
If `delta` represents the gradient of the loss with respect to the output of the dropout layer, then the gradient with respect to the input of the dropout layer is:

```
delta_input = delta * mask / (1 - p)
```

This ensures that the gradients are scaled appropriately and only propagate through the active neurons.

Summary

- **Intuitive Idea**: Randomly deactivate neurons to prevent over-reliance on any single neuron.
- **Forward Pass**: Apply dropout mask and scale outputs.
- **Backward Pass**: Propagate gradients only through active neurons and scale appropriately.

Dropout is particularly effective in fully connected layers and is often used in conjunction with other regularization techniques like L2 regularization.

The UML class diagram of the LeNet-5 implementation:

```
+-----------------------------------------------------------+
|                        <<abstract>>                        |
|                          Layer                            |
+-----------------------------------------------------------+
```

```
| + forward(x: np.ndarray) : np.ndarray                      |
| + backward(grad: np.ndarray) : np.ndarray                  |
| + update(learning_rate: float) : void                      |
+------------------------------------------------------------+
       ^
       |
       |
+------+-------+---------------------+----------------------+
|              |                     |                      |
|              |                     |                      |
|  +-----------+----------+  +-------+--------+  +----------+-------+
|  |       Conv2D         |  |    MaxPool2D   |  |      Dense      |
|  +----------------------+  +----------------+  +-----------------+
|  | - in_channels: int   |  | - pool_size: int|  | - input_dim: int|
|  | - out_channels: int  |  | - stride: int   |  | - output_dim:int|
|  | - kernel_size: int   |  +----------------+  +-----------------+
|  | - stride: int        |  | + forward()     |  | + forward()     |
|  | - padding: int       |  | + backward()    |  | + backward()    |
|  | - weights: np.ndarray|  +----------------+  +-----------------+
|  | - biases: np.ndarray |
|  | - cache: tuple       |
|  +----------------------+
|  | + im2col()           |
|  | + col2im()           |
|  | + forward()          |
|  | + backward()         |
|  | + update()           |
|  +----------------------+
|
|  +----------------------+  +-----------------+  +----------------+
|  |    BatchNorm2D       |  |     Dropout     |  |     ReLU       |
|  +----------------------+  +-----------------+  +----------------+
|  | - gamma: np.ndarray  |  | - p: float      |  | - cache: np.ndarray|
|  | - beta: np.ndarray   |  | - mask: np.ndarray| +----------------+
|  | - eps: float         |  +-----------------+  | + forward()     |
|  | - momentum: float    |  | + forward()     |  | + backward()    |
|  | - running_mean: np.ndarray| + backward()    |  +----------------+
|  | - running_var: np.ndarray |
|  | - cache: tuple       |  +-----------------+  +----------------+
|  +----------------------+                    |     Softmax     |
|  | + forward()          |                    +----------------+
|  | + backward()         |                    | - cache: np.ndarray|
|  | + update()           |                    +----------------+
|  +----------------------+                    | + forward()     |
|                                             | + backward()    |
|  +----------------------+                    +----------------+
|  |      Flatten         |
|  +----------------------+
|  | - input_shape: tuple |
|  +----------------------+
|  | + forward()          |
|  | + backward()         |
|  +----------------------+
|
+------------------------------------------------------------+
|                          LeNet5                            |
+------------------------------------------------------------+
| - layers: List[Layer]                                      |
```

```
+----------------------------------------------------------+
| + forward(x: np.ndarray, training: bool) : np.ndarray     |
| + backward(grad: np.ndarray) : void                       |
| + update(learning_rate: float) : void                     |
| + predict(x: np.ndarray) : np.ndarray                     |
| + loss(y_pred, y_true) : float                            |
| + accuracy(y_pred, y_true) : float                        |
+----------------------------------------------------------+
```

Key Points

Each concrete layer implements:

- `forward()`: Computes the forward pass
- `backward()`: Computes gradients during back propagation
- `update()`: Updates parameters (for layers with learnable weights)
- All layers maintain their own parameters and cache variables needed for back propagation.

The `LeNet5` class orchestrates:

- Full forward/backward passes through all layers
- Parameter updates
- Prediction and evaluation methods

LeNet Implementation from Scratch

A complete implementation of LeNet-5 using NumPy, featuring:

- Convolution layers with im2col optimization
- Max pooling layers
- Fully connected layers
- Batch normalization
- Dropout
- ReLU activation
- Softmax output

```
import numpy as np
from typing import Tuple, List, Optional

class Layer:
    def forward(self, x: np.ndarray) -> np.ndarray:
```

```python
        raise NotImplementedError

    def backward(self, grad: np.ndarray) -> np.ndarray:
        raise NotImplementedError

    def update(self, learning_rate: float) -> None:
        pass

import numpy as np
from numpy.lib.stride_tricks import as_strided
from typing import Tuple, List, Optional

class Layer:
    def forward(self, x: np.ndarray) -> np.ndarray:
        raise NotImplementedError

    def backward(self, grad: np.ndarray) -> np.ndarray:
        raise NotImplementedError

    def update(self, learning_rate: float) -> None:
        pass

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

        # He initialization
        scale = np.sqrt(2.0 / (in_channels * kernel_size * kernel_size))
        self.weights = np.random.randn(out_channels, in_channels, kernel_size, kernel_size) * scale
        self.biases = np.zeros(out_channels)

        self.cache = None

    def im2col(self, x: np.ndarray) -> np.ndarray:
        """Optimized im2col using as_strided"""
        N, C, H, W = x.shape
        K = self.kernel_size
        stride = self.stride
        pad = self.padding

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

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

        # Shape of the output array
        shape = (N, C, K, K, H_out, W_out)
```

```
        # Strides of the input array (bytes to move in each dimension)
        strides = (x_padded.strides[0], x_padded.strides[1],
                   x_padded.strides[2], x_padded.strides[3],
                   stride * x_padded.strides[2], stride * x_padded.strides[3])

        # Create strided view
        strided = as_strided(x_padded, shape=shape, strides=strides)

        # Reshape to column format
        cols = strided.transpose(0, 4, 5, 1, 2, 3).reshape(N * H_out * W_out, -1)

        return cols.T  # Transpose to match standard im2col output

    def col2im(self, cols: np.ndarray, x_shape: Tuple[int, int, int, int]) -> np.ndarray:
        """Inverse of im2col using numpy operations"""
        N, C, H, W = x_shape
        K = self.kernel_size
        stride = self.stride
        pad = self.padding

        H_padded = H + 2 * pad
        W_padded = W + 2 * pad
        x_padded = np.zeros((N, C, H_padded, W_padded), dtype=cols.dtype)

        H_out = (H + 2 * pad - K) // stride + 1
        W_out = (W + 2 * pad - K) // stride + 1

        # Reshape columns back to image patches
        cols_reshaped = cols.T.reshape(N, H_out, W_out, C, K, K)
        cols_reshaped = cols_reshaped.transpose(0, 3, 4, 5, 1, 2)  # N, C, K, K, H_out, W_out

        # Add patches back to image
        for i in range(K):
            for j in range(K):
                x_padded[:, :, i:i+H_out*stride:stride, j:j+W_out*stride:stride] += cols_reshaped[:, :, i, j, :, :]

        if pad == 0:
            return x_padded
        return x_padded[:, :, pad:-pad, pad:-pad]

    def forward(self, x: np.ndarray) -> np.ndarray:
        N, C, H, W = x.shape
        K = self.kernel_size
        stride = self.stride
        pad = self.padding

        H_out = (H + 2 * pad - K) // stride + 1
        W_out = (W + 2 * pad - K) // stride + 1

        # Convert input to columns using optimized im2col
        x_cols = self.im2col(x)

        # Reshape weights
        w_cols = self.weights.reshape(self.out_channels, -1)

```

```python
        # Perform matrix multiplication
        out = w_cols @ x_cols + self.biases.reshape(-1, 1)

        # Reshape output
        out = out.reshape(self.out_channels, H_out, W_out, N)
        out = out.transpose(3, 0, 1, 2)

        # Cache for backward pass
        self.cache = (x, x_cols)

        return out

    def backward(self, grad: np.ndarray) -> np.ndarray:
        x, x_cols = self.cache
        N, C, H, W = x.shape
        K = self.kernel_size

        # Reshape gradient (outc, oh, hw, n) => (outc, oh*ow*n)
        grad_reshaped = grad.transpose(1, 2, 3, 0).reshape(self.out_channels, -1)

        # Calculate weight gradients
        #(outc, oh*ow*n) @ (oh*ow*n, in_C*k*k)
        dw = grad_reshaped @ x_cols.T
        # (outc, in_c, k, k)
        dw = dw.reshape(self.weights.shape)

        # Calculate bias gradients db(outc, 1)
        db = np.sum(grad, axis=(0, 2, 3))

        # Calculate input gradients, w_cols(out_c, in_c*k*k)
        w_cols = self.weights.reshape(self.out_channels, -1)
        dx_cols = w_cols.T @ grad_reshaped
        dx = self.col2im(dx_cols, x.shape)

        # Store gradients
        self.dw = dw
        self.db = db

        return dx

    def update(self, learning_rate: float) -> None:
        self.weights -= learning_rate * self.dw
        self.biases -= learning_rate * self.db

#
# The other classes (MaxPool2D, Flatten, Dense, BatchNorm2D, Dropout, ReLU, Softmax, LeNet5)
#
class MaxPool2D(Layer):
    def __init__(self, pool_size: int = 2, stride: int = 2):
        self.pool_size = pool_size
        self.stride = stride
        self.cache = None

    def forward(self, x: np.ndarray) -> np.ndarray:
        N, C, H, W = x.shape
        pool_size = self.pool_size
        stride = self.stride
```

```python

        H_out = (H - pool_size) // stride + 1
        W_out = (W - pool_size) // stride + 1

        # Reshape input for vectorized max operation
        x_reshaped = x.reshape(N, C, H // pool_size, pool_size,
                               W // pool_size, pool_size)
        out = x_reshaped.max(axis=3).max(axis=4)

        # Create mask for backward pass
        x_reshaped = x.reshape(N, C, H_out, stride, W_out, stride)
        mask = (x_reshaped == out[:, :, :, np.newaxis, :, np.newaxis])

        self.cache = (x.shape, mask)
        return out

    def backward(self, grad: np.ndarray) -> np.ndarray:
        input_shape, mask = self.cache
        N, C, H, W = input_shape
        pool_size = self.pool_size

        # Upsample gradient
        grad_upsampled = np.repeat(np.repeat(grad, pool_size, axis=2), pool_size, axis=3)

        # Apply mask
        dx = grad_upsampled * mask.reshape(input_shape)
        return dx

class Flatten(Layer):
    def __init__(self):
        self.input_shape = None

    def forward(self, x: np.ndarray) -> np.ndarray:
        self.input_shape = x.shape
        return x.reshape(x.shape[0], -1)

    def backward(self, grad: np.ndarray) -> np.ndarray:
        return grad.reshape(self.input_shape)

class Dense(Layer):
    def __init__(self, input_dim: int, output_dim: int):
        # He initialization
        scale = np.sqrt(2.0 / input_dim)
        self.weights = np.random.randn(input_dim, output_dim) * scale
        self.biases = np.zeros(output_dim)
        self.cache = None

    def forward(self, x: np.ndarray) -> np.ndarray:
        self.cache = x
        return x @ self.weights + self.biases

    def backward(self, grad: np.ndarray) -> np.ndarray:
        x = self.cache
        self.dw = x.T @ grad
        self.db = np.sum(grad, axis=0)
        return grad @ self.weights.T

    def update(self, learning_rate: float) -> None:
```

```python
        self.weights -= learning_rate * self.dw
        self.biases -= learning_rate * self.db

class BatchNorm2D(Layer):
    def __init__(self, num_features: int, eps: float = 1e-5, momentum: float = 0.1):
        self.gamma = np.ones(num_features)
        self.beta = np.zeros(num_features)
        self.eps = eps
        self.momentum = momentum
        self.running_mean = np.zeros(num_features)
        self.running_var = np.ones(num_features)
        self.cache = None

    def forward(self, x: np.ndarray, training: bool = True) -> np.ndarray:
        N, C, H, W = x.shape

        if training:
            # Calculate mean and variance over batch and spatial dimensions
            mean = np.mean(x, axis=(0, 2, 3), keepdims=True)
            var = np.var(x, axis=(0, 2, 3), keepdims=True)

            # Update running statistics
            self.running_mean = self.momentum * mean.squeeze() + (1 - self.momentum) * self.running_mean
            self.running_var = self.momentum * var.squeeze() + (1 - self.momentum) * self.running_var

            # Normalize
            x_norm = (x - mean) / np.sqrt(var + self.eps)
        else:
            # Use running statistics during inference
            x_norm = (x - self.running_mean.reshape(1, C, 1, 1)) / np.sqrt(self.running_var.reshape(1, C, 1, 1) + self.eps)

        # Scale and shift
        out = self.gamma.reshape(1, C, 1, 1) * x_norm + self.beta.reshape(1, C, 1, 1)

        if training:
            self.cache = (x, mean, var, x_norm)

        return out

    def backward(self, grad: np.ndarray) -> np.ndarray:
        x, mean, var, x_norm = self.cache
        N, C, H, W = x.shape

        # Calculate gradients
        dbeta = np.sum(grad, axis=(0, 2, 3))
        dgamma = np.sum(grad * x_norm, axis=(0, 2, 3))

        # Intermediate gradients
        dx_norm = grad * self.gamma.reshape(1, C, 1, 1)
        dvar = np.sum(dx_norm * (x - mean) * -0.5 * (var + self.eps) ** (-1.5), axis=(0, 2, 3), keepdims=True)
        dmean = np.sum(dx_norm * -1 / np.sqrt(var + self.eps), axis=(0, 2, 3), keepdims=True) + \
                dvar * np.mean(-2 * (x - mean), axis=(0, 2, 3), keepdims=True))

```

```python
        # Final gradient
        dx = dx_norm / np.sqrt(var + self.eps) + \
             dvar * 2 * (x - mean) / (N * H * W) + \
             dmean / (N * H * W)

        self.dgamma = dgamma
        self.dbeta = dbeta

        return dx

    def update(self, learning_rate: float) -> None:
        self.gamma -= learning_rate * self.dgamma
        self.beta -= learning_rate * self.dbeta

# inverted drop-out implementation
class Dropout(Layer):
    def __init__(self, p: float = 0.5):
        self.p = p
        self.mask = None

    def forward(self, x: np.ndarray, training: bool = True) -> np.ndarray:
        if training:
            self.mask = (np.random.rand(*x.shape) > self.p) / (1 - self.p)
            return x * self.mask
        return x

    def backward(self, grad: np.ndarray) -> np.ndarray:
        return grad * self.mask

class ReLU(Layer):
    def __init__(self):
        self.cache = None

    def forward(self, x: np.ndarray) -> np.ndarray:
        self.cache = x
        return np.maximum(0, x)

    def backward(self, grad: np.ndarray) -> np.ndarray:
        x = self.cache
        return grad * (x > 0)

class Softmax(Layer):
    def __init__(self):
        self.cache = None

    def forward(self, x: np.ndarray) -> np.ndarray:
        # Numerically stable softmax
        exps = np.exp(x - np.max(x, axis=1, keepdims=True))
        out = exps / np.sum(exps, axis=1, keepdims=True)
        self.cache = out
        return out

    def backward(self, grad: np.ndarray) -> np.ndarray:
        s = self.cache
        # Jacobian matrix: diag(s) - s.T @ s
        return s * (grad - (grad * s).sum(axis=1, keepdims=True))

class LeNet5:
```

```python
    def __init__(self, input_shape: Tuple[int, int, int], num_classes: int):
        C, H, W = input_shape

        self.layers = [
            Conv2D(in_channels=C, out_channels=6, kernel_size=5, stride=1, padding=2),
            BatchNorm2D(num_features=6),
            ReLU(),
            MaxPool2D(pool_size=2, stride=2),

            Conv2D(in_channels=6, out_channels=16, kernel_size=5, stride=1, padding=0),
            BatchNorm2D(num_features=16),
            ReLU(),
            MaxPool2D(pool_size=2, stride=2),

            Flatten(),

            Dense(input_dim=16*5*5, output_dim=120),
            BatchNorm2D(num_features=120),
            ReLU(),
            Dropout(p=0.5),

            Dense(input_dim=120, output_dim=84),
            BatchNorm2D(num_features=84),
            ReLU(),
            Dropout(p=0.5),

            Dense(input_dim=84, output_dim=num_classes),
            Softmax()
        ]

    def forward(self, x: np.ndarray, training: bool = True) -> np.ndarray:
        for layer in self.layers:
            if isinstance(layer, (BatchNorm2D, Dropout)):
                x = layer.forward(x, training=training)
            else:
                x = layer.forward(x)
        return x

    def backward(self, grad: np.ndarray) -> None:
        for layer in reversed(self.layers):
            grad = layer.backward(grad)

    def update(self, learning_rate: float) -> None:
        for layer in self.layers:
            if hasattr(layer, 'update'):
                layer.update(learning_rate)

    def predict(self, x: np.ndarray) -> np.ndarray:
        return np.argmax(self.forward(x, training=False), axis=1)

    def loss(self, y_pred: np.ndarray, y_true: np.ndarray) -> float:
        # Cross-entropy loss
        m = y_true.shape[0]
        log_likelihood = -np.log(y_pred[range(m), y_true])
        loss = np.sum(log_likelihood) / m
        return loss

    def accuracy(self, y_pred: np.ndarray, y_true: np.ndarray) -> float:
```

```python
        return np.mean(y_pred == y_true)

# Example usage:
if __name__ == "__main__":
    # Create a dummy dataset
    np.random.seed(42)
    X_train = np.random.randn(100, 1, 28, 28)  # 100 samples, 1 channel, 28x28
    y_train = np.random.randint(0, 10, size=100)  # 10 classes

    # Initialize LeNet-5
    lenet = LeNet5(input_shape=(1, 28, 28), num_classes=10)

    # Training loop (simplified)
    learning_rate = 0.01
    epochs = 5
    batch_size = 10

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

        for i in range(0, len(X_train), batch_size):
            X_batch = X_train[i:i+batch_size]
            y_batch = y_train[i:i+batch_size]

            # Forward pass
            y_pred = lenet.forward(X_batch)

            # Calculate loss
            loss = lenet.loss(y_pred, y_batch)
            epoch_loss += loss * len(X_batch)

            # Calculate accuracy
            preds = np.argmax(y_pred, axis=1)
            correct += np.sum(preds == y_batch)

            # Backward pass
            grad = y_pred.copy()
            grad[range(len(y_batch)), y_batch] -= 1
            grad /= len(y_batch)
            lenet.backward(grad)

            # Update weights
            lenet.update(learning_rate)

        epoch_loss /= len(X_train)
        accuracy = correct / len(X_train)

        print(f"Epoch {epoch+1}/{epochs}, Loss: {epoch_loss:.4f}, Accuracy: {accuracy:.4f}")
```

Key Features of This Implementation:

1. **Convolutional Layers with im2col**:
   - Uses the im2col algorithm for efficient convolution operations

- Implements both forward and backward passes
- Supports padding and stride

2. **Batch Normalization**:
   - Implements batch norm for both convolutional and fully connected layers
   - Maintains running statistics for inference
   - Properly handles the scale (gamma) and shift (beta) parameters
3. **Dropout**:
   - Implements inverted dropout
   - Only active during training
   - Scales activations during training to maintain expected values
4. **Complete LeNet-5 Architecture**:
   - Two convolutional layers with ReLU and max pooling
   - Three fully connected layers
   - Batch normalization and dropout for regularization
   - Softmax output for classification
5. **Training Framework**:
   - Includes forward and backward passes
   - Weight updates with learning rate
   - Cross-entropy loss calculation
   - Accuracy measurement

This implementation provides a complete, from-scratch version of LeNet-5 with modern improvements like batch normalization and dropout, while maintaining the original architecture's essence.
