3.3.1 Geometric Transformations

- Random rotation (±30°)
- Bidirectional flipping (horizontal and vertical axes)
- Variable scaling (90-110%)

3.3.2 Photometric Adjustments

- Brightness variation (±20%)
- Contrast modification (±15%)
- Hue and saturation adjustment (±10%)

3.4 Dataset Partitioning

The complete dataset was divided according to the following scheme:

Partition	Percentage	Image Count
Training	80%	70,295
Validation	20%	17,558
Testing	-	33

Table 2: Dataset partitioning scheme

The test dataset contains carefully selected representative samples not exposed during training or validation phases.

3.5 Preprocessing Methodology

Each image undergoes sequential preprocessing operations:

1. **Normalization**: Pixel values scaled to [0,1] range
2. **Standardization**: Mean subtraction and standard deviation normalization
3. **Tensor Conversion**: Images transformed into PyTorch tensor format

The preprocessing pipeline was implemented using PyTorch's transformation framework:

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])
```

4 Methodology

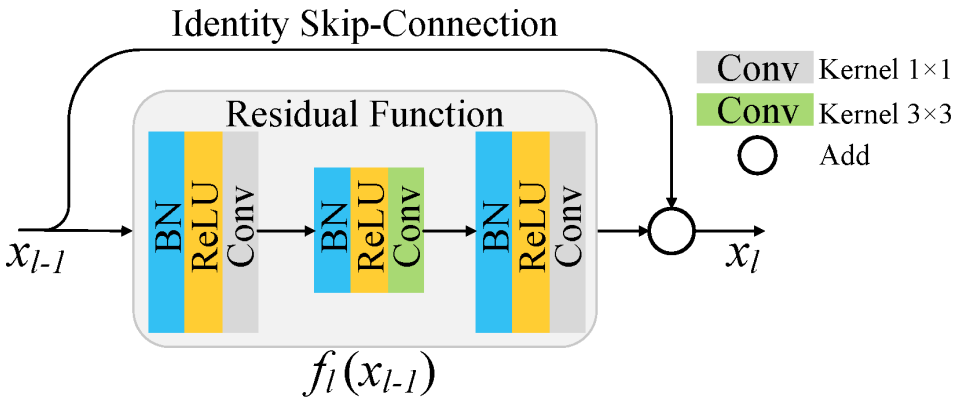

Figure 2: The modified ResNet-9 architecture

4.1 Modified ResNet-9 Architecture

4.1.1 Framework Overview

The customized ResNet-9 architecture consists of three principal components:

1. **Initial Feature Extraction**: Sequential convolutional layers
2. **Dual Residual Blocks**: Incorporating identity mappings
3. **Classification Component**: Final dense layer network

4.1.2 Architectural Specifications

The complete network architecture is defined as follows:

```
class ResNet9(ImageClassificationBase):
    def __init__(self, in_channels, num_diseases):
        super().__init__()

        # Initial Feature Extraction Block
        self.conv1 = ConvBlock(in_channels, 64)

        # Downsampling Block
        self.conv2 = ConvBlock(64, 128, pool=True)

        # First Residual Block
        self.res1 = nn.Sequential(ConvBlock(128, 128),
                                  ConvBlock(128, 128))

        # Intermediate Feature Extraction
        self.conv3 = ConvBlock(128, 256, pool=True)

        # Advanced Feature Extraction
```

```
        self.conv4 = ConvBlock(256, 512, pool=True)

        # Second Residual Block
        self.res2 = nn.Sequential(ConvBlock(512, 512),
                                  ConvBlock(512, 512))

        # Classification Component
        self.classifier = nn.Sequential(
            nn.MaxPool2d(4),
            nn.Flatten(),
            nn.Linear(512, num_diseases))
```

4.1.3 Residual Connection Implementation

The residual module implementation preserves identity mapping:

```
class ResidualBlock(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 3, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.conv2 = nn.Conv2d(3, 3, kernel_size=3, padding=1)
        self.relu2 = nn.ReLU()

    def forward(self, x):
        out = self.conv1(x)
        out = self.relu1(out)
        out = self.conv2(out)
        return self.relu2(out) + x  # Identity mapping
```

4.1.4 Convolutional Module Design

The fundamental building block of the architecture:

```
def ConvBlock(in_channels, out_channels, pool=False):
    layers = [
        nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
        nn.BatchNorm2d(out_channels),
        nn.ReLU(inplace=True)
    ]
    if pool:
        layers.append(nn.MaxPool2d(4))
    return nn.Sequential(*layers)
```

4.2 Training Methodology

4.2.1 Loss Function Selection

Cross-entropy loss was selected as the optimization criterion:

```
loss_fn = nn.CrossEntropyLoss()
```