Assignment 1 - 611 Image Filtering and Convolutional Neural Networks (CNNs)

Gaurav Patil - 012629189

February 22, 2025

1 Introduction

Image filtering and Convolutional Neural Networks (CNNs) are crucial components of image processing and computer vision. Image filtering enhances image quality, reduces noise, and extracts meaningful features, while CNNs are deep learning architectures designed for automated image classification and object detection. This report explores various filtering techniques and CNN architectures, including their performance on the CIFAR-10 dataset with different optimizers and hardware setups.

You can find the Assignment 2 repository at: GitHub - Assignment 2

2 Image Filtering

2.1 Techniques Implemented

Image filtering applies mathematical operations to pixel values to enhance image features. The following techniques were implemented:

- **Gaussian Filtering**: Smooths images and reduces noise while preserving edges.
- **Median Filtering**: Removes salt-and-pepper noise by replacing pixel values with the median of neighboring pixels.
- **Sobel Edge Detection**: Detects edges by computing gradients in the horizontal and vertical directions.
- **Laplacian Filter**: Enhances edge detection by computing the second derivative of image intensity.
- **Custom Kernel Filtering**: Applies user-defined convolution kernels for specific feature extraction.

2.2 Results and Observations

Applying these filters to sample images demonstrated their effectiveness. Gaussian and median filters successfully removed noise, while Sobel and Laplacian filters highlighted edges. Increasing kernel sizes in Gaussian filtering resulted in stronger blurring effects.

Noise significantly impacted edge detection, leading to false edges. Preprocessing with blurring before applying edge detection improved results. Advanced edge detection techniques, such as Laplacian of Gaussian (LoG) and the Second Derivative of Gaussian, were applied to both normal and blurred images using a 3x3 averaging kernel. The results showed that edges were less pronounced in blurred images, emphasizing the importance of preprocessing.

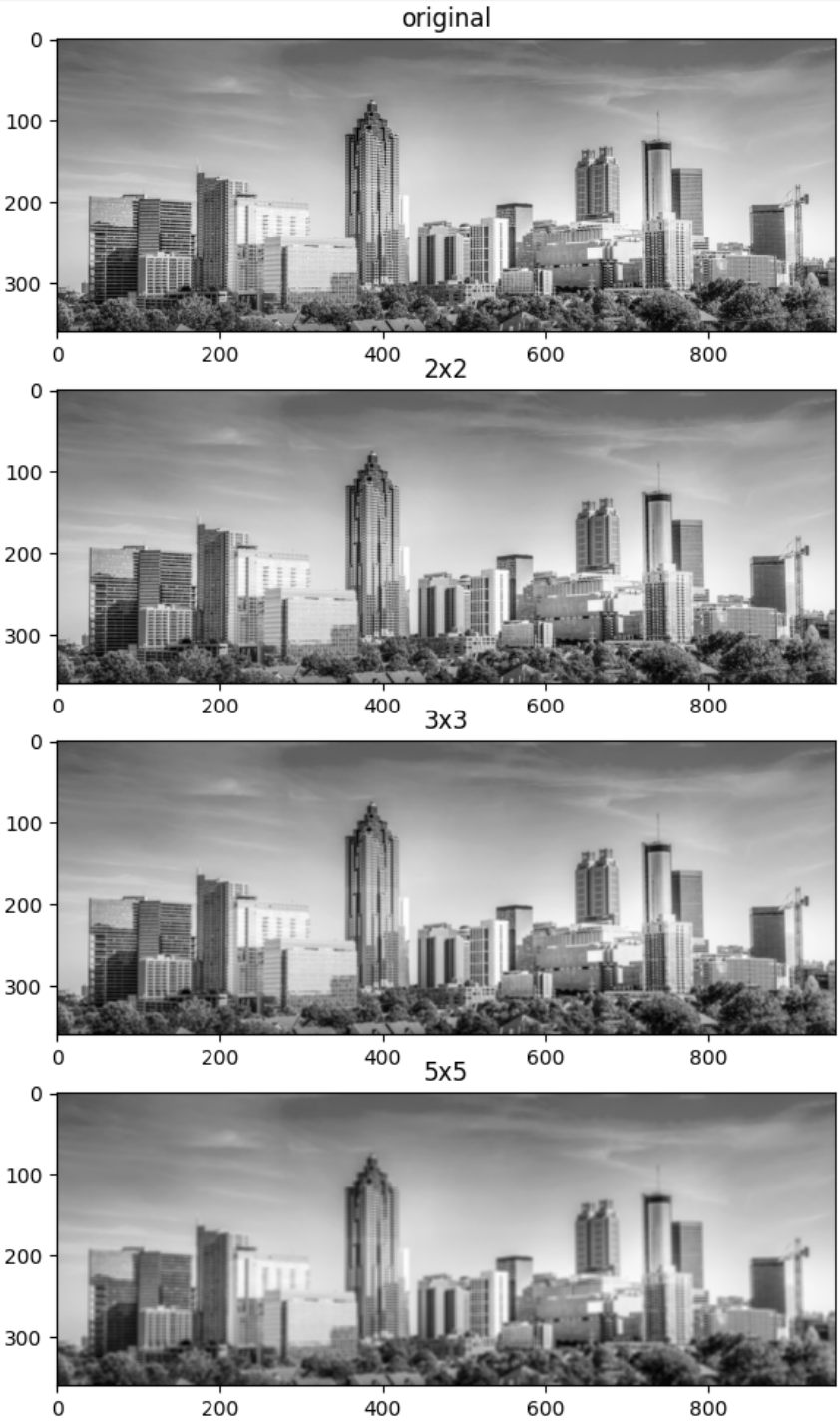

Figure 1: Effects of various kernel sizes on blurring.

```
[13]: # Create a custom kernel for blurring

# 2x2 kernel for averaging blurring
S2x2 = np.array([[ 1, 1],
                 [ 1, 1]])
S3x3 = np.array([[ 1, 1,1],
                 [ 1, 1,1],
                 [ 1, 1,1]])
S5x5 = np.array([[ 1, 1,1,1,1], [ 1, 1,1,1,1],[ 1, 1,1,1,1],
                 [ 1, 1,1,1,1],
                 [ 1, 1,1,1,1]])
fig = plt.figure(figsize=(48, 12))
fig.add_subplot(4,1,1)
plt.imshow(gray, cmap='gray')
plt.title('original')

# Filter the image using filter2D, which has inputs: (grayscale image, bit-depth, kernel)
blurred_image = cv2.filter2D(gray, -1, S2x2/4.0)
fig.add_subplot(4,1,2)
plt.imshow(blurred_image, cmap='gray')
plt.title('2x2')

# TODO: blur image using a 3x3 average
blurred_image3 = cv2.filter2D(gray, -1, S3x3/9.0)
fig.add_subplot(4,1,3)
plt.imshow(blurred_image3, cmap='gray')
plt.title('3x3')

# TODO: blur image using a 5x5 average
blurred_image5 = cv2.filter2D(gray, -1, S5x5/25.0)
fig.add_subplot(4,1,4)
plt.imshow(blurred_image5, cmap='gray')
plt.title('5x5')

plt.show()
```

Figure 2: Code changes for various kernel sizes on blurring.

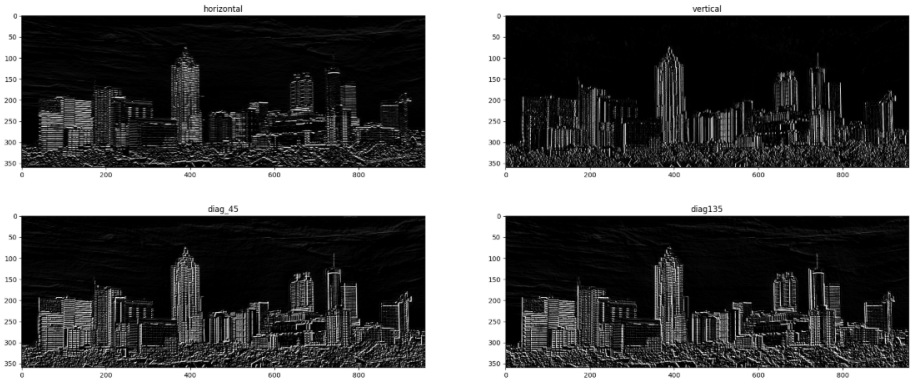

Figure 3: Output of existing kernels.

```python
# Define kernels
log_kernel = np.array([[ 0, 0, -1, 0, 0],
                       [ 0, -1, -2, -1, 0],
                       [-1, -2, 16, -2, -1],
                       [ 0, -1, -2, -1, 0],
                       [ 0, 0, -1, 0, 0]])
dog_kernel = np.array([[ 0, 0, -1, 0, 0],
                       [ 0, -1, -2, -1, 0],
                       [-1, -2, 16, -2, -1],
                       [ 0, -1, -2, -1, 0],
                       [ 0, 0, -1, 0, 0]])
sobel_x = np.array([[-1, 0, 1],
                    [-2, 0, 2],
                    [-1, 0, 1]])
sobel_y = np.array([[-1, -2, -1],
                    [ 0,  0,  0],
                    [ 1,  2,  1]])

# Create figure for plotting
fig = plt.figure(figsize=(24, 10))  # Increase figure size to accommodate all subplots

# Apply kernels using cv2.filter2D
filtered_image_log = cv2.filter2D(gray, -1, log_kernel)
fig.add_subplot(4,1, 1)
plt.imshow(filtered_image_log, cmap='gray')
plt.title('Laplacian of Gaussian')

filtered_image_dog = cv2.filter2D(gray, -1, dog_kernel)
fig.add_subplot(4,1, 2)
plt.imshow(filtered_image_dog, cmap='gray')
plt.title('Second Derivative of Gaussian')

filtered_image_sobel_x = cv2.filter2D(gray, -1, sobel_x)
fig.add_subplot(4,1, 3)
plt.imshow(filtered_image_sobel_x, cmap='gray')
plt.title('Sobel X')

filtered_image_sobel_y = cv2.filter2D(gray, -1, sobel_y)
fig.add_subplot(4,1, 4)
plt.imshow(filtered_image_sobel_y, cmap='gray')
plt.title('Sobel Y')
```

Figure 4: Other edge detection kernel Code.

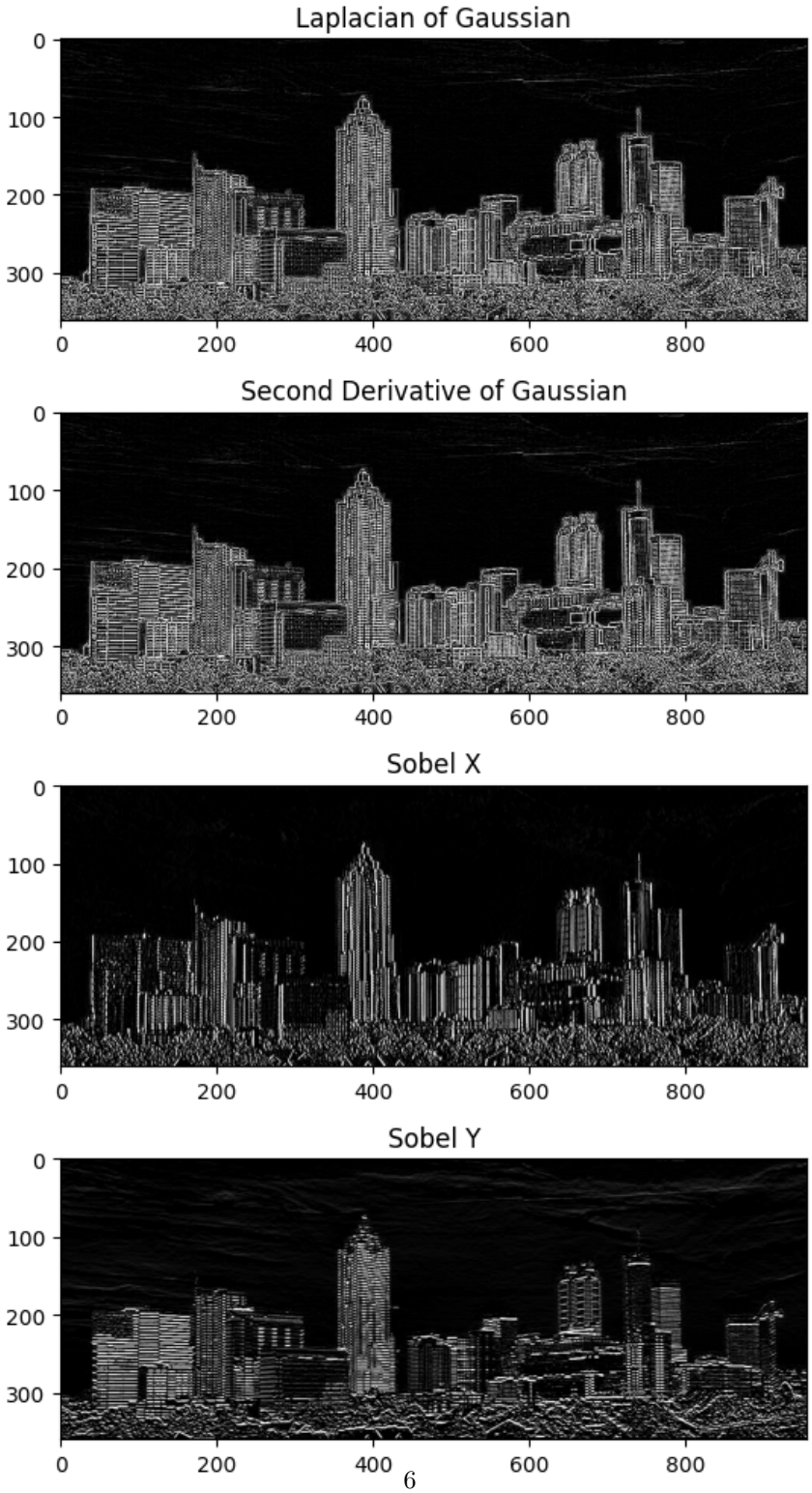

Figure 5: Other edge detection kernel on the base gray image.

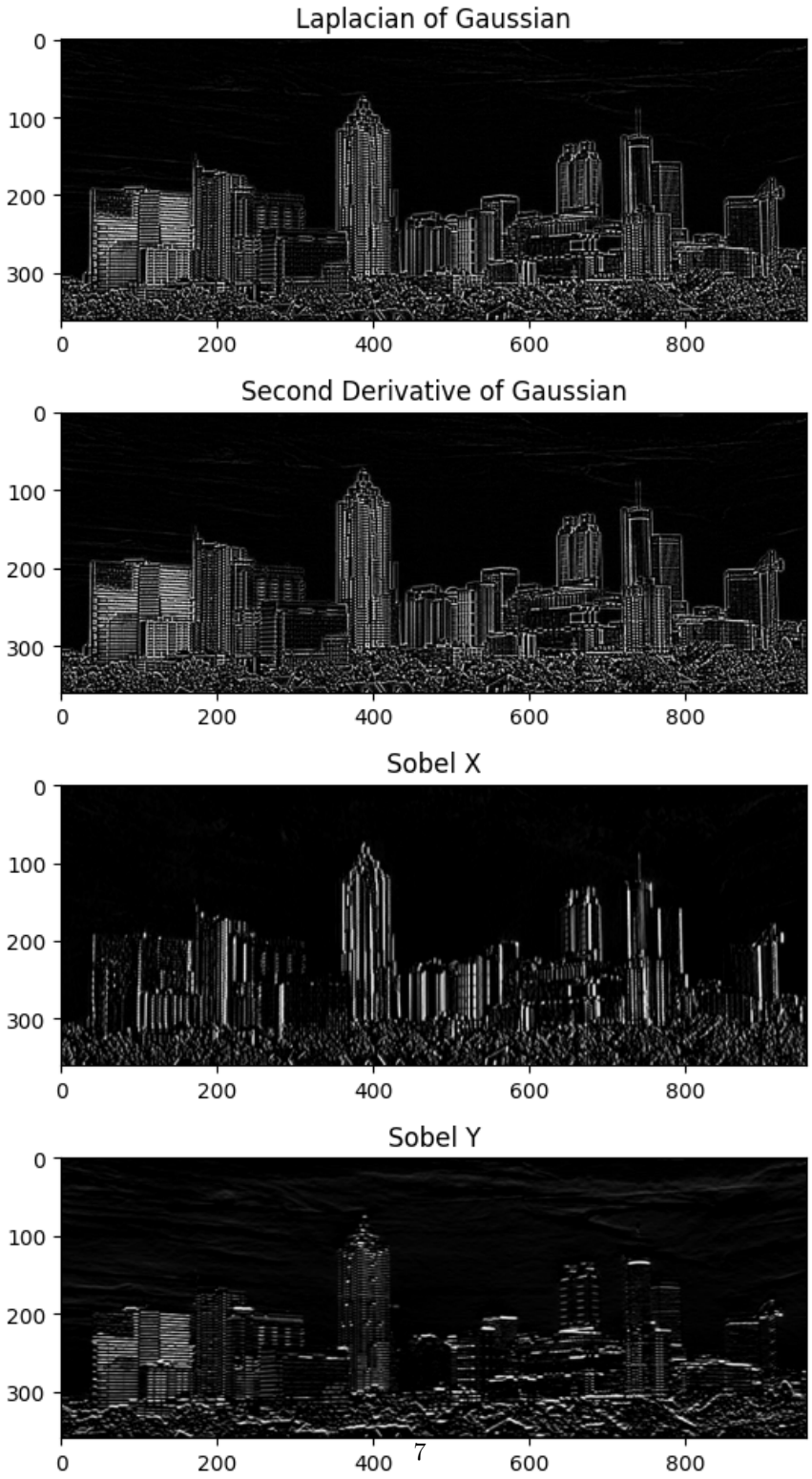

Figure 6: Other edge detection kernel on the Blurred 3x3 image.

3 Convolutional Neural Networks (CNNs)

3.1 Model Architecture and Dataset

Different CNN architectures were tested, modifying the number of convolutional layers, filter sizes, and network depth to evaluate their impact on classification accuracy. The CIFAR-10 dataset was used, consisting of 60,000 images across 10 classes, each with a resolution of 32x32 pixels. The dataset's small size limited the benefits of very deep architectures, making regularization essential to prevent overfitting.

For the chosen architecture, we achieved an accuracy of approximately 72%.

3.2 Optimization Strategies

Two optimizers were compared:

- **Adam Optimizer**: Adaptive learning rates with momentum for faster convergence and efficient handling of sparse gradients.
- **SGD with Momentum**: Uses a fixed learning rate with an additional momentum term to accelerate learning and avoid local minima.

Both optimizers achieved similar accuracy, but SGD proved computationally faster.

3.3 Training Performance and Overfitting

The training process revealed:

- The Adam-optimized model converged quickly but exhibited fluctuations in accuracy.
- The SGD-optimized model required more epochs but demonstrated a smoother and more stable learning curve.
- Overfitting was observed due to insufficient regularization, leading to unreliable generalization. Overfitted models were not saved.

3.4 Hardware Performance Analysis

Model training was tested on different hardware setups:

- **Laptop CPU**: Training was significantly slower due to limited computational power.
- **Google Colab GPU**: Training time improved drastically, demonstrating the advantage of dedicated hardware acceleration.
- **Jetson Orin**: Training was attempted, but JetPack 6.2 is not currently compatible with PyTorch CUDA. A downgrade or a future CUDA-compatible release would be required.

```python
# Define the CNN architecture
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        # Convolutional Layers (Extract features from the input image)
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, stride=1, padding=1)
        # Input: (3x32x32) -> Output: (32x32x32) [3 channels to 32 channels]

        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1)
        # Input: (32x32x32) -> Output: (64x32x32) [32 channels to 64 channels]

        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding=1)
        # Input: (64x32x32) -> Output: (128x32x32) [64 channels to 128 channels]

        # Max Pooling Layer (Reduces spatial dimensions by a factor of 2)
        self.pool = nn.MaxPool2d(2, 2)  # Halves the spatial size

        # Fully Connected Layers
        # After 3 layers of max pooling, the image size is reduced from (32x32) to (4x4)
        # The input to the first fully connected layer will be 128 * 4 * 4 = 2048 features
        self.fc1 = nn.Linear(128 * 4 * 4, 512)  # Flattened size after pooling
        self.fc2 = nn.Linear(512, 10)  # Output layer (for 10 classes)

        # Dropout Layer (To prevent overfitting)
        self.dropout = nn.Dropout(0.25)

    def forward(self, x):
        # Convolutional + ReLU + Pooling
        x = self.pool(F.relu(self.conv1(x)))  # (32x32x3) -> (32x32x32) -> (16x16x32)
        x = self.pool(F.relu(self.conv2(x)))  # (16x16x32) -> (16x16x64) -> (8x8x64)
        x = self.pool(F.relu(self.conv3(x)))  # (8x8x64) -> (8x8x128) -> (4x4x128)

        # Flatten the tensor before passing it to the fully connected layer
        x = x.view(-1, 128 * 4 * 4)  # Flattened size: (batch_size, 128*4*4)

        # Fully connected layers with ReLU activation
        x = F.relu(self.fc1(x))  # (batch_size, 512)
        x = self.dropout(x)  # Dropout for regularization
        x = self.fc2(x)  # Output layer, (batch_size, 10) -> 10 classes

        return x
```

Figure 7: CNN architecture code.

```python
import torch.optim as optim
import torch.nn as nn
import torch.nn.functional as F

# specify loss function (categorical cross-entropy)
criterion = nn.CrossEntropyLoss()

# specify optimizer
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)  # Original choice
#optim.Adam(model.parameters(), lr=0.001)
import torch.optim as optim

# Specify loss function (Categorical Cross-Entropy)
criterion = nn.CrossEntropyLoss()

# Optimizer Variations
#optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
#optimizer_adam = optim.Adam(model.parameters(), lr=0.001)
```

Figure 8: Optimizer code.

```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=2048, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=10, bias=True)
  (dropout): Dropout(p=0.25, inplace=False)
)
```

Figure 9: CNN architecture.

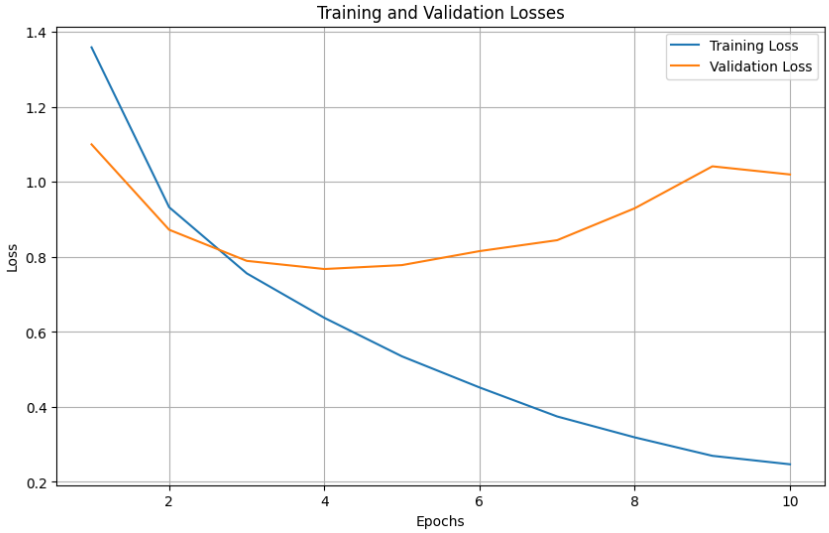

Figure 10: Graph of training loss and validation loss for Adam optimizer

```
Test Loss: 0.782643

Test Accuracy of airplane: 81% (817/1000)
Test Accuracy of automobile: 86% (860/1000)
Test Accuracy of  bird: 70% (706/1000)
Test Accuracy of   cat: 54% (542/1000)
Test Accuracy of  deer: 69% (694/1000)
Test Accuracy of   dog: 61% (610/1000)
Test Accuracy of  frog: 73% (733/1000)
Test Accuracy of horse: 66% (667/1000)
Test Accuracy of  ship: 86% (868/1000)
Test Accuracy of truck: 79% (790/1000)

Test Accuracy (Overall): 72% (7287/10000)
```

Figure 11: Accuracy for Adam Optimizer

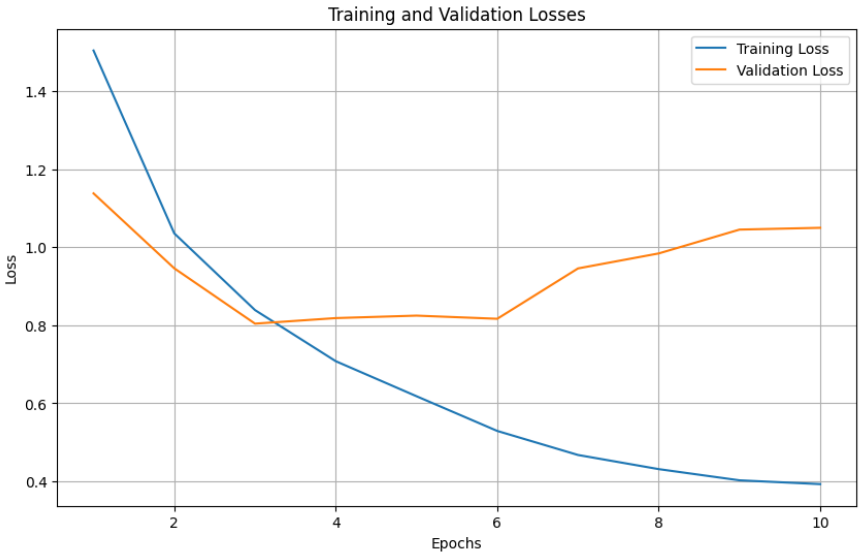

Figure 12: Graph of training loss and validation loss for SGD optimizer

```
Test Loss: 0.820112

Test Accuracy of airplane: 82% (828/1000)
Test Accuracy of automobile: 85% (852/1000)
Test Accuracy of  bird: 70% (701/1000)
Test Accuracy of   cat: 43% (437/1000)
Test Accuracy of  deer: 71% (713/1000)
Test Accuracy of   dog: 56% (568/1000)
Test Accuracy of  frog: 79% (791/1000)
Test Accuracy of horse: 75% (753/1000)
Test Accuracy of  ship: 78% (788/1000)
Test Accuracy of truck: 78% (780/1000)

Test Accuracy (Overall): 72% (7211/10000)
```

Figure 13: Accuracy for SGD Optimizer

4 Conclusion

Image filtering techniques play a vital role in preprocessing images by reducing noise and enhancing features, which is particularly beneficial for edge detection. We observed that increasing kernel size increases blurring, and preprocessing an image before edge detection improves accuracy. Advanced techniques like the Laplacian of Gaussian and Second Derivative of Gaussian demonstrated their effectiveness, especially when applied to preprocessed images.

In CNN training, the CIFAR-10 dataset's small image size limited the benefits of deeper architectures, emphasizing the need for regularization. Both Adam and SGD optimizers achieved similar accuracy, but SGD proved computationally more efficient.

Hardware acceleration played a crucial role in improving training speed, with GPUs significantly reducing training time compared to CPUs. The Jetson Orin's compatibility issues with PyTorch CUDA highlight challenges in deploying deep learning models on edge devices, requiring careful software version management.