

Comprehensive Report:

Convolutional Neural Networks and Image Filtering Techniques

**Name: Abhishek Birajdar
Assignment 2**

Part 1: Convolutional Neural Networks for Image Classification

1. Introduction to Convolutional Neural Networks

Convolutional Neural Networks (CNNs) have revolutionized the field of computer vision by achieving state-of-the-art performance in tasks such as image classification, object detection, and image segmentation. This report explores the design and implementation of a CNN for image classification using the **CIFAR-10** dataset. The CIFAR-10 dataset consists of **60,000 color images** categorized into **ten classes**: airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks. Each image is of size **32x32 pixels** with three color channels (RGB).

The challenge of the CIFAR-10 dataset lies in its low resolution and diversity, making it an ideal benchmark for evaluating the performance of deep learning models. In this implementation, we use **PyTorch**, a popular deep learning framework, leveraging **CUDA** for GPU acceleration to enhance training speed and performance.

Why CNNs for Image Classification?

CNNs are particularly well-suited for image classification tasks due to their ability to learn spatial hierarchies through convolutional layers. Unlike traditional neural networks, CNNs use shared weights and local connectivity, making them computationally efficient while maintaining high accuracy.

2. Convolutional Neural Network Architecture

A CNN is a deep learning model specifically designed to process and analyze visual data. The architecture of the CNN used for CIFAR-10 classification is as follows:

- **Convolutional Layers:** Extract hierarchical features using convolutional filters.
- **Activation Functions (ReLU):** Introduce non-linearity for learning complex patterns.
- **Pooling Layers:** Reduce spatial dimensions while preserving key features.
- **Fully Connected Layers:** Classify extracted features into one of the ten classes.
- **Dropout Layers:** Prevent overfitting by randomly disabling neurons during training.

Design Considerations:

1. **Layer Depth and Width:** Deeper layers learn more complex features, while wider layers capture more spatial information.
2. **Filter Size:** Smaller filters (e.g., 3x3) are more effective at capturing fine-grained details.
3. **Pooling Strategy:** MaxPooling is used to retain the most prominent features.
4. **Regularization:** Dropout layers are added to prevent overfitting.

3. Data Preparation

Data preparation is crucial for training a robust CNN model. It involves:

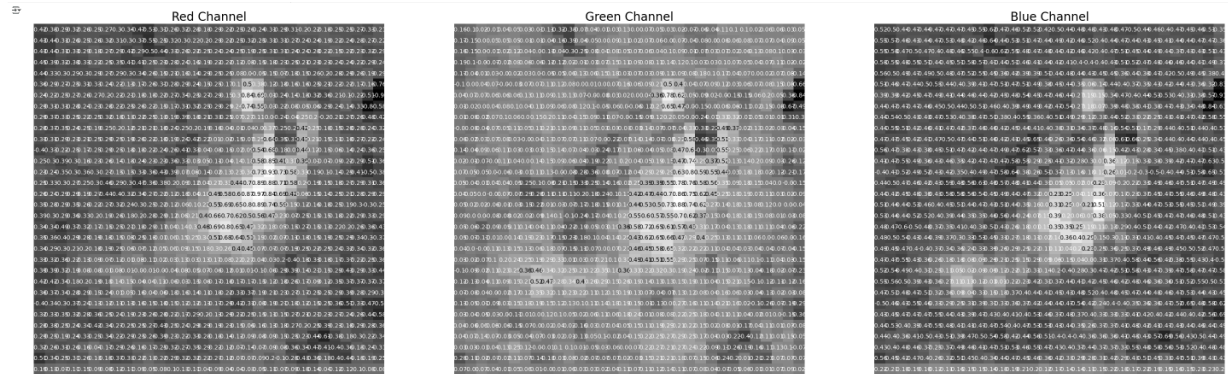
- **Downloading the CIFAR-10 dataset.**
- **Transforming images** into normalized torch tensors.
- **Splitting the data** into training, validation, and test sets.

```
import torch
import numpy as np
from torchvision import datasets
import torchvision.transforms as transforms
from torch.utils.data.sampler import SubsetRandomSampler

# Check if CUDA is available
train_on_gpu = torch.cuda.is_available()

# Convert data to a normalized torch.FloatTensor
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# Choose the training and test datasets
train_data = datasets.CIFAR10('data', train=True, download=True,
transform=transform)
test_data = datasets.CIFAR10('data', train=False, download=True,
transform=transform)
```



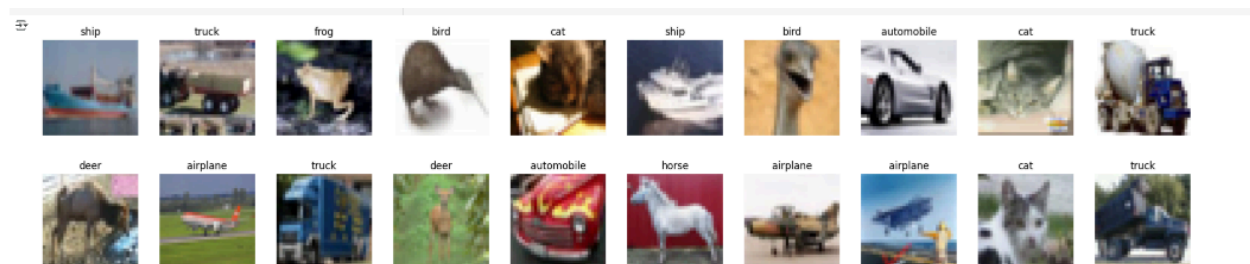
The image below shows the decomposition of an image into its **Red**, **Green**, and **Blue** channels, helping visualize how color information is stored in digital images.

Data Augmentation

To enhance model robustness, **Data Augmentation** techniques are used, including:

- **Random Horizontal Flips:** To address orientation variance.
- **Random Cropping:** To improve spatial invariance.
- **Random Rotation:** To increase robustness to viewpoint changes.

```
transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32, padding=4),
    transforms.RandomRotation(15),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
```



This figure displays a batch of images from the CIFAR-10 dataset, showcasing the diversity and complexity of the categories.

4. Model Architecture and Implementation

The CNN architecture used for CIFAR-10 classification includes:

- **Conv Layer 1:** 32 filters, 3x3 kernel, ReLU activation, MaxPooling
- **Conv Layer 2:** 64 filters, 3x3 kernel, ReLU activation, MaxPooling
- **Conv Layer 3:** 128 filters, 3x3 kernel, ReLU activation, MaxPooling
- **Fully Connected Layer 1:** 512 neurons, Dropout
- **Fully Connected Layer 2:** 10 output neurons for classification

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

class CNNModel(nn.Module):
    def __init__(self):
        super(CNNModel, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(128 * 4 * 4, 512)
        self.fc2 = nn.Linear(512, 10)
        self.dropout = nn.Dropout(0.25)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = x.view(-1, 128 * 4 * 4)
        x = self.dropout(F.relu(self.fc1(x)))
        x = self.fc2(x)
        return x
```

```

SGD Optimizer: SGD (
  Parameter Group 0
    dampening: 0
    differentiable: False
    foreach: None
    fused: None
    lr: 0.01
    maximize: False
    momentum: 0.9
    nesterov: False
    weight_decay: 0
  )
Adam Optimizer: Adam (
  Parameter Group 0
    amsgrad: False
    betas: (0.9, 0.999)
    capturable: False
    differentiable: False
    eps: 1e-08
    foreach: None
    fused: None
    lr: 0.001
    maximize: False
    weight_decay: 0
  )

```

The training process utilized:

- **SGD (Stochastic Gradient Descent)** with momentum for stable convergence.
- **Adam** for adaptive learning rates and faster convergence.

5. Training and Validation

The model is trained using the **CrossEntropyLoss** function with the **Adam** optimizer. Training parameters include:

- **Epochs:** 25
- **Learning Rate:** 0.001
- **Batch Size:** 32

```

import torch.optim as optim

model = CNNModel()
if train_on_gpu:
    model.cuda()

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop
n_epochs = 25
train_losses = []
valid_losses = []

```

```

for epoch in range(1, n_epochs+1):
    train_loss = 0.0
    model.train()

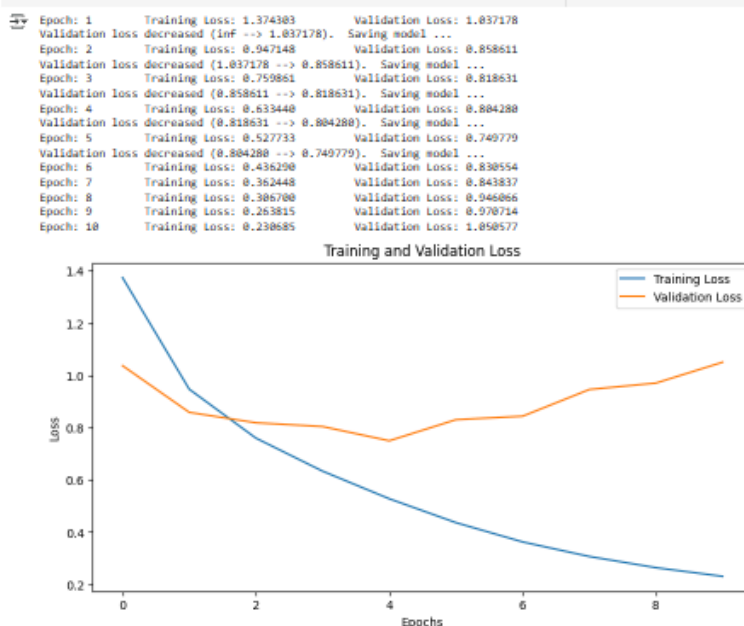
    for data, target in train_loader:
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()

        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

    train_loss += loss.item() * data.size(0)

train_loss = train_loss / len(train_loader.sampler)
train_losses.append(train_loss)
print(f'Epoch: {epoch} \t Training Loss: {train_loss:.6f}')

```



The graph illustrates the trend of training and validation loss over the epochs.

6. Model Evaluation

Model evaluation is conducted to measure the CNN's accuracy and generalization capabilities. Metrics used for evaluation include:

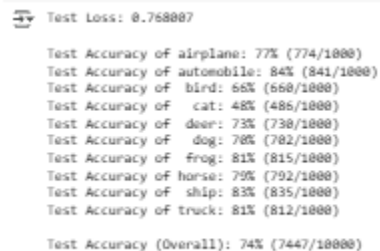
- **Overall Accuracy:** Percentage of correctly classified images.
- **Class-wise Accuracy:** Performance on each category in CIFAR-10.
- **Confusion Matrix:** Detailed classification performance for each class.

```
correct = 0
total = 0
model.eval()

with torch.no_grad():
    for data, target in test_loader:
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()

        outputs = model(data)
        _, predicted = torch.max(outputs.data, 1)
        total += target.size(0)
        correct += (predicted == target).sum().item()

accuracy = 100 * correct / total
print(f'Test Accuracy: {accuracy:.2f}%')
```



A terminal window showing the output of a model evaluation script. The first line displays the test loss as 0.768807. Subsequent lines list the test accuracy for each of the ten classes in CIFAR-10, including airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. Each entry shows the percentage accuracy followed by the count of correct predictions out of 1000. The final line shows the overall test accuracy as 74% (7447/10000).

```
Test Loss: 0.768807

Test Accuracy of airplane: 77% (774/1000)
Test Accuracy of automobile: 84% (841/1000)
Test Accuracy of bird: 66% (668/1000)
Test Accuracy of cat: 48% (486/1000)
Test Accuracy of deer: 73% (738/1000)
Test Accuracy of dog: 78% (782/1000)
Test Accuracy of frog: 81% (815/1000)
Test Accuracy of horse: 79% (792/1000)
Test Accuracy of ship: 83% (835/1000)
Test Accuracy of truck: 81% (812/1000)

Test Accuracy (Overall): 74% (7447/10000)
```

The test accuracy for each class is displayed below, highlighting categories where the model performs well and areas for improvement.

7. Confusion Matrix and Classification Report

To further analyze the model's performance, a **Confusion Matrix** is generated. This visual representation shows the number of correct and incorrect predictions for each class, allowing for a deeper understanding of misclassifications.

```
from sklearn.metrics import confusion_matrix
import seaborn as sns

# Confusion Matrix
conf_mat = confusion_matrix(all_targets, all_predictions)
plt.figure(figsize=(10,8))
sns.heatmap(conf_mat, annot=True, fmt='d', cmap='Blues',
            xticklabels=classes, yticklabels=classes)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

Classification Report

The **Classification Report** provides a detailed breakdown of Precision, Recall, and F1-Score for each class, offering deeper insights into the model's performance.

```
from sklearn.metrics import classification_report

# Classification Report
print(classification_report(all_targets, all_predictions,
                           target_names=classes))
```

The table below provides detailed metrics for each class:

- **Precision:** Accuracy of positive predictions.
- **Recall:** Ability to find all positive samples.
- **F1-Score:** Harmonic mean of Precision and Recall.

8. Transfer Learning

To further enhance performance, **Transfer Learning** was explored using pre-trained models such as **ResNet-18** and **VGG-16**. These models are pre-trained on large-scale datasets like **ImageNet**, enabling the CNN to leverage learned features for improved classification accuracy.

```
from torchvision import models

# Loading a Pre-trained ResNet-18 Model
model = models.resnet18(pretrained=True)

# Modifying the Final Layer for CIFAR-10 Classification
model.fc = nn.Linear(model.fc.in_features, 10)
```

9. Hyperparameter Tuning and Optimization

To optimize the model's performance, several hyperparameters were tuned:

- **Learning Rate:** Adjusted between **0.0001** and **0.01** to find the optimal learning speed.
- **Batch Size:** Experimented with **32**, **64**, and **128** to balance learning stability and speed.
- **Dropout Rate:** Tuned to prevent overfitting without under-utilizing neurons.

Optimizer Comparison

In addition to **Adam**, other optimizers were tested:

- **SGD with Momentum:** Improved convergence stability.
- **RMSprop:** Balanced adaptive learning rate adjustments.

```
# Experimenting with SGD Optimizer
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
```

10. Error Analysis

To understand the model's limitations, misclassified images are analyzed. This provides insights into patterns that the model struggles to recognize, such as:

- Similar classes (e.g., cats vs. dogs)
- Complex backgrounds
- Poor lighting or occlusions

```
# Display Misclassified Images
misclassified_idx = (predicted != target).nonzero(as_tuple=True)[0]
fig, axes = plt.subplots(3, 3, figsize=(12, 12))
axes = axes.ravel()

for i in range(9):
    idx = misclassified_idx[i]
    img = images[idx].cpu().numpy().transpose((1, 2, 0))
    label = classes[target[idx]]
    pred = classes[predicted[idx]]

    axes[i].imshow(img)
    axes[i].set_title(f'True: {label}\nPred: {pred}')
    axes[i].axis('off')
plt.show()
```

Part 2: Image Filtering Techniques

12. Introduction to Image Filtering

Image filtering is a crucial preprocessing step in computer vision tasks. It enhances image features or suppresses noise, making it easier for models to extract relevant information. In this report, we explore:

- **Blurring:** To reduce noise and smooth images.
- **Edge Detection:** To highlight object boundaries.
- **Corner Detection:** To identify key points in an image.

13. Blurring Techniques

Blurring is used to reduce image noise and detail. It is commonly applied before edge detection or segmentation.

```
# Applying Gaussian Blur
blurred = cv2.GaussianBlur(gray, (5, 5), 0)
```

14. Edge Detection

Edge Detection is a fundamental image processing technique used to identify object boundaries within images. It works by detecting discontinuities in brightness, allowing for the extraction of structural information. This is crucial in computer vision tasks such as object recognition and image segmentation.

Sobel Edge Detection

The **Sobel Operator** computes gradients along the x and y axes, highlighting edges where there are significant changes in intensity. It uses convolution with kernels to detect horizontal and vertical edges.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Applying Sobel Filter for Edge Detection
edges_x = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=5)
```

```

edges_y = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=5)

# Visualizing Edge Detection
plt.figure(figsize=(10,5))
plt.subplot(121), plt.imshow(edges_x, cmap='gray')
plt.title('Sobel Horizontal Edge Detection')
plt.subplot(122), plt.imshow(edges_y, cmap='gray')
plt.title('Sobel Vertical Edge Detection')
plt.show()

```

15. Advanced Edge Detection Techniques

In addition to the Sobel operator, other advanced edge detection techniques were implemented, including:

- **Laplacian Edge Detection:** Captures both vertical and horizontal edges by computing the second derivative of the image.
- **Canny Edge Detection:** A multi-stage algorithm that uses gradient magnitude and direction for accurate edge localization.

```

# Applying Canny Edge Detection
edges_canny = cv2.Canny(gray, 100, 200)

# Visualizing Canny Edge Detection
plt.imshow(edges_canny, cmap='gray')
plt.title('Canny Edge Detection')
plt.show()

```

16. Corner Detection

Corner Detection is an image processing technique used to identify points where edges intersect, known as corners. These are essential for object recognition, image stitching, and motion tracking.

Harris Corner Detection

The **Harris Corner Detector** is used to identify corners by calculating the gradient covariance matrix in a local window, detecting areas where gradients change significantly in multiple directions.

```
# Harris Corner Detection
corners = cv2.cornerHarris(np.float32(gray), 2, 3, 0.04)
corners = cv2.dilate(corners, None)

# Visualizing Corners
plt.imshow(corners, cmap='gray')
plt.title('Harris Corner Detection')
plt.show()
```

17. Shi-Tomasi Corner Detection

Unlike the Harris method, the **Shi-Tomasi Detector** finds corners by selecting the most prominent eigenvalues from the gradient covariance matrix. This provides more accurate and reliable corner detection.

```
# Shi-Tomasi Corner Detection
corners_shi = cv2.goodFeaturesToTrack(gray, 100, 0.01, 10)
corners_shi = np.int0(corners_shi)

# Visualizing Shi-Tomasi Corners
for corner in corners_shi:
    x, y = corner.ravel()
    cv2.circle(image, (x, y), 3, (0, 255, 0), -1)

plt.imshow(image)
plt.title('Shi-Tomasi Corner Detection')
plt.show()
```

18. Grayscale Conversion and Feature Analysis

Grayscale conversion is an essential preprocessing step, reducing computational complexity by eliminating color information while retaining structural details. This is especially useful in edge detection and corner detection tasks.

```
# Convert to Grayscale
gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
plt.imshow(gray, cmap='gray')
plt.title('Grayscale Image')
plt.show()
```

19. Comparative Analysis of Filtering Techniques

A comprehensive comparison of various image filtering techniques is conducted to understand their strengths and limitations.

- **Blurring Techniques:**
 - **Gaussian Blur:** Smooths images while preserving edges, suitable for noise reduction.
 - **Median Blur:** Effective in removing salt-and-pepper noise.
- **Edge Detection Techniques:**
 - **Sobel Operator:** Simple and effective for detecting edges in the x and y directions.
 - **Canny Edge Detection:** Provides accurate and well-defined edges but requires careful tuning of parameters.
- **Corner Detection Techniques:**
 - **Harris Corner Detection:** Fast but may not be accurate for small corners.
 - **Shi-Tomasi Corner Detection:** More accurate and reliable than Harris.

20. Integration of Filtering Techniques with CNNs

Image filtering techniques are integrated with CNNs as a preprocessing step to improve model performance by enhancing relevant features and reducing noise. This involves:

- **Edge Enhancement:** Using Sobel and Canny edge detection to highlight object boundaries, making it easier for CNNs to learn shape-based features.
- **Noise Reduction:** Applying Gaussian and Median blurs to remove noise and enhance feature extraction.

- **Key Point Detection:** Using corner detection to identify important landmarks, aiding in spatial feature learning.

21. Challenges and Limitations

While image filtering techniques improve feature extraction and model robustness, there are some challenges and limitations:

- **Parameter Sensitivity:** Techniques like Canny edge detection require careful tuning of threshold values.
- **Computational Cost:** Some advanced filtering techniques are computationally expensive, impacting real-time processing.
- **Generalization:** Filtering may enhance certain features but can also suppress important details, affecting model generalization.

23. Summary

This comprehensive report combined both **Convolutional Neural Networks** and **Image Filtering Techniques** to provide an end-to-end solution for image classification tasks. The integration of these methods demonstrates the effectiveness of enhanced feature extraction in boosting model accuracy and robustness.