CSCI 611 - Applied Machine Learning

Convolutional Neural Network (CNN)

Vishruth Byaramadu Lokesh

Student ID - 011885654

Part 1 – Image filter

Methodology

1. Uploading and Converting Image to Grayscale

The first step will involve the loading of an image('building.jpg') and converting it into grayscale using OpenCV. The process of converting into grayscale reduces the complexity of the image processing by maintaining essential texture and edge-based information but lowers the computational complexity.

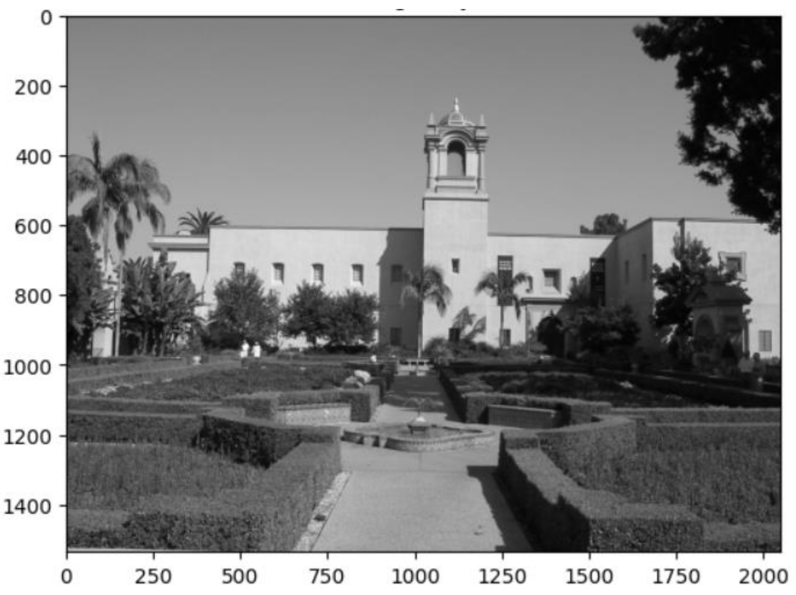


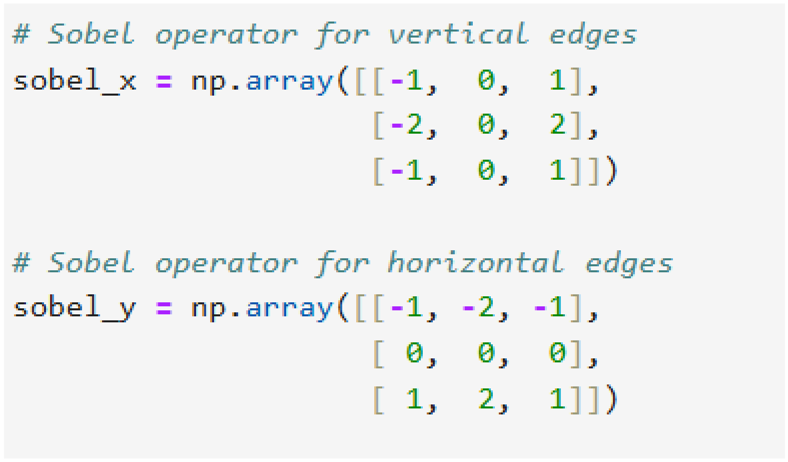
Fig 1: Chosen Image

2. Edge Detection Using Sobel Filters

In detecting vertical and horizontal edges in grayscale images, the Sobel operator is generally applied using the following Sobel kernels:

- Vertical edge detection Kernel (Kx)
- Horizontal Edge Detection Kernel (Ky)

The image convolves with these kernels in such a manner as to enhance the vertical and horizontal edges.



```python
# Sobel operator for vertical edges
sobel_x = np.array([[-1,  0,  1],
                    [-2,  0,  2],
                    [-1,  0,  1]])

# Sobel operator for horizontal edges
sobel_y = np.array([[-1, -2, -1],
                    [ 0,  0,  0],
                    [ 1,  2,  1]])
```

Fig 2: Sobel Operator

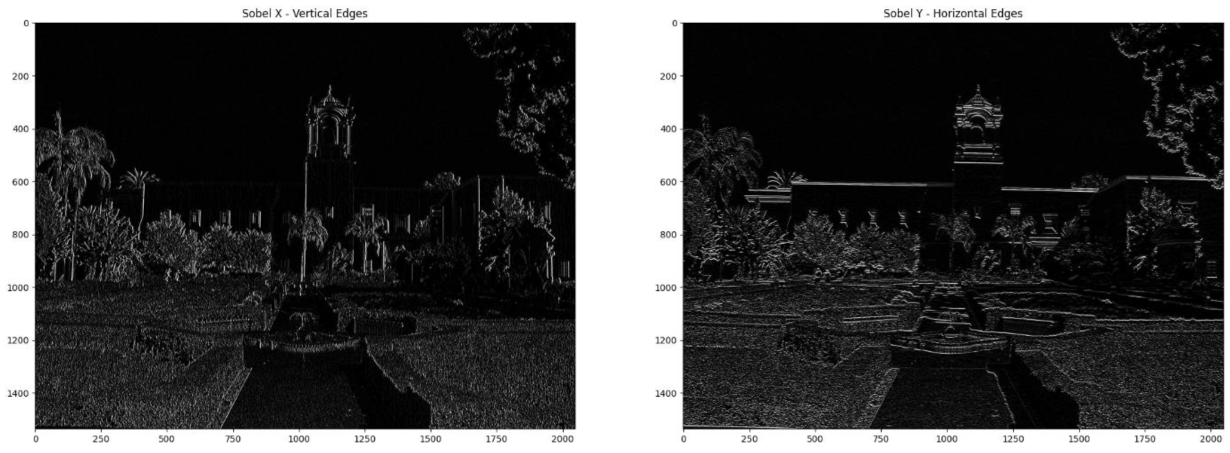


Fig 3: Vertical and horizontal edge detection after using sobel operator

3. Corner Detection

The corner detection is achieved using kernels specifically designed to detect different corner regions. Convolution of the grayscale image is done with each kernel, and the final response is set from summation of the square of the responses.

```python
corner_kernels = [
    np.array([[0, 1, -3],
              [0, 1, 1],
              [0, 0, 0]]),  # Top-left corner

    np.array([[-3, 1, 0],
              [1, 1, 0],
              [0, 0, 0]]),  # Top-right corner

    np.array([[0, 0, 0],
              [1, 1, 0],
              [-3, 1, 0]]),  # Bottom-left corner

    np.array([[0, 0, 0],
              [0, 1, 1],
              [0, 1, -3]])   # Bottom-right corner
 ]
```

Fig 4: Kernels operators that detect corners

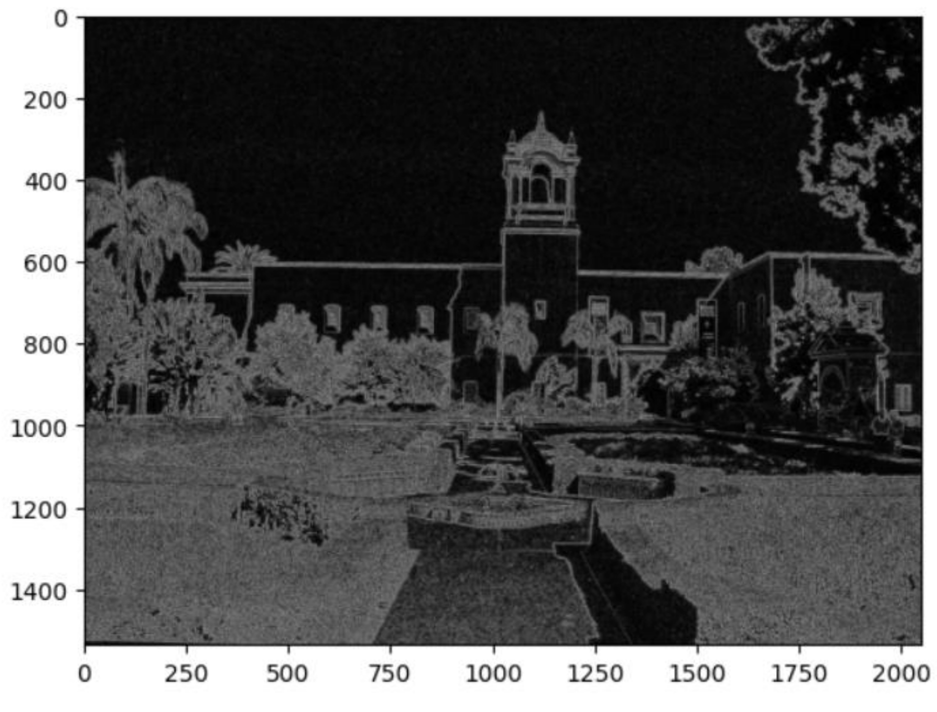
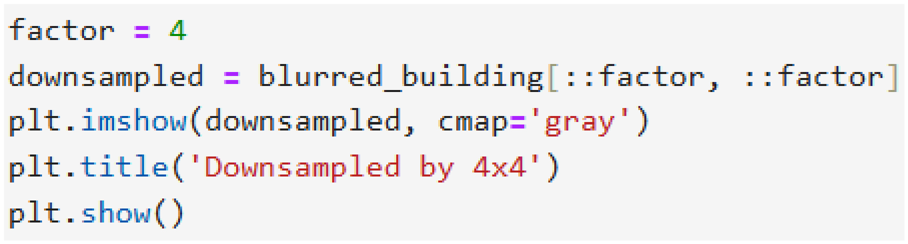


Fig 5: Result of Corner Detection

4. Scaling After Blurring

The first stage involves the blurring of the grayscale image using an averaging filter of 5 x 5, which aids in noise removal and smoothing. The image is then down sampled to smaller resolution, (4x4) to analyze the effects of scaling after blurring.



```
factor = 4
downsampled = blurred_building[::factor, ::factor]
plt.imshow(downsampled, cmap='gray')
plt.title('Downsampled by 4x4')
plt.show()
```

Fig 6: Code snippet for down sampling by a factor of 4

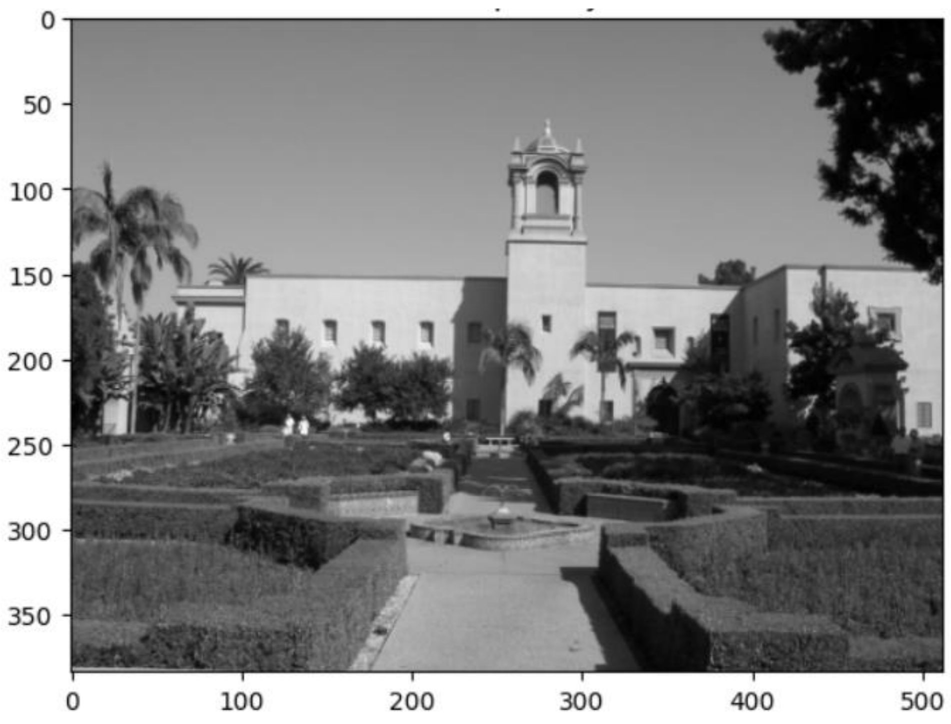
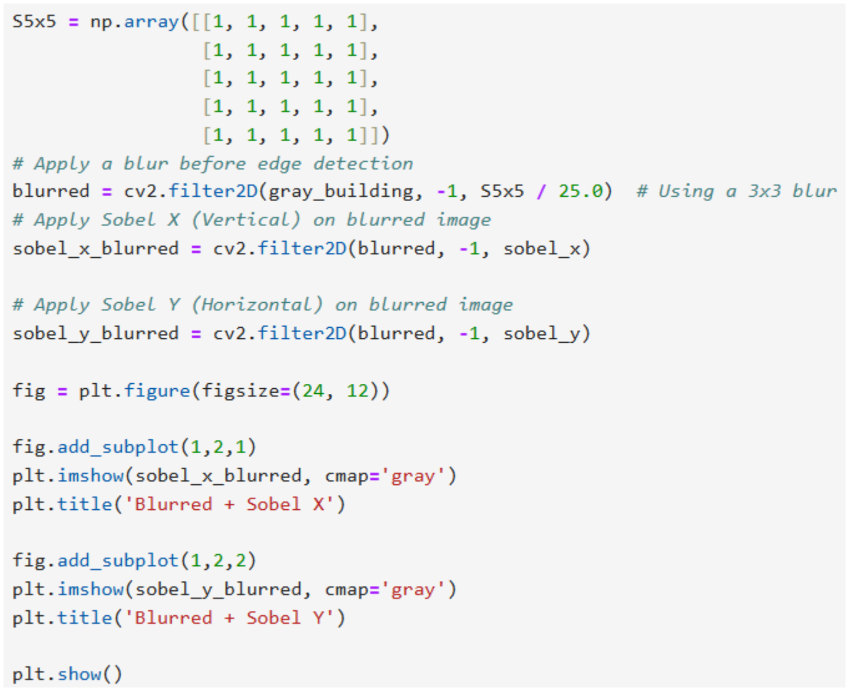


Fig 7: Downsampled Image by a 4x4

5. Edge Detection After Blurring

To assess the effect of blurring on edge detection, gray-scale images were blurred prior to the application of Sobel filters. The graphics produced allow for an evaluation of the variation between direct and pre-blurred edge detection. The edge detection of blurred images was able to show very good edge maps.



```
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]])
# Apply a blur before edge detection
blurred = cv2.filter2D(gray_building, -1, S5x5 / 25.0)  # Using a 3x3 blur
# Apply Sobel X (Vertical) on blurred image
sobel_x_blurred = cv2.filter2D(blurred, -1, sobel_x)

# Apply Sobel Y (Horizontal) on blurred image
sobel_y_blurred = cv2.filter2D(blurred, -1, sobel_y)

fig = plt.figure(figsize=(24, 12))

fig.add_subplot(1,2,1)
plt.imshow(sobel_x_blurred, cmap='gray')
plt.title('Blurred + Sobel X')

fig.add_subplot(1,2,2)
plt.imshow(sobel_y_blurred, cmap='gray')
plt.title('Blurred + Sobel Y')

plt.show()
```

Fig 8: Code snippet for edge detection after blurring

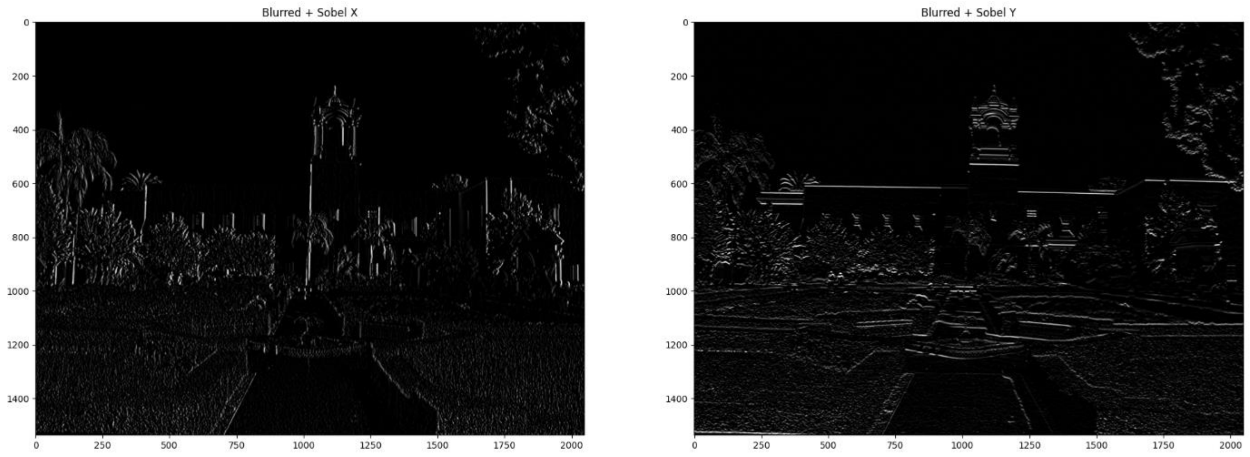


Fig 9: Result of edge detection after blurring

Part 2

Introduction

This report presents an overview of the implementation and findings of a Convolutional Neural Network (CNN) trained on CIFAR-10 data set using PyTorch. The model was developed for classifying images into one among ten categories: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. The training phase made use of CUDA for GPU acceleration wherever applicable. The report describes data preprocessing, model architecture, the training and evaluation techniques, and the key findings.

Dataset and Preprocessing

The CIFAR-10 dataset consists of 60,000 32x32 color images across 10 classes for training and validation. The following preprocessing steps were performed:

- Data was normalized with values range from -1 to 1 using transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
- Training data was split into 80% training and 20% validation using SubsetRandomSampler.
- Data was loaded in batches of 20 instances using DataLoader for efficient training.

Model Architecture

The CNN architecture comprises the following layers:

- Convolutional Layers: Three convolutional layers with ReLU activations.
    - Conv1: 3 input channels → 16 output channels (kernel size = 3, padding = 1)
    - Conv2: 16 input channels → 32 output channels (kernel size = 3, padding = 1)
    - Conv3: 32 input channels → 64 output channels (kernel size = 3, padding = 1)
- Max Pooling Layers: Applied after each convolutional layer with a 2x2 kernel.
- Fully Connected Layers:
    - Flattening layer to convert convolutional outputs to a vector.
    - FC1: 1024 input neurons → 512 neurons with ReLU activation.
    - FC2: 512 input neurons → 10 output neurons (one for each class).
- Dropout Layers: Used with a probability of 0.25 to prevent overfitting.

CNN Architecture Visualization

The generated CNN architecture visualization is included below, showcasing the computational graph of the network:

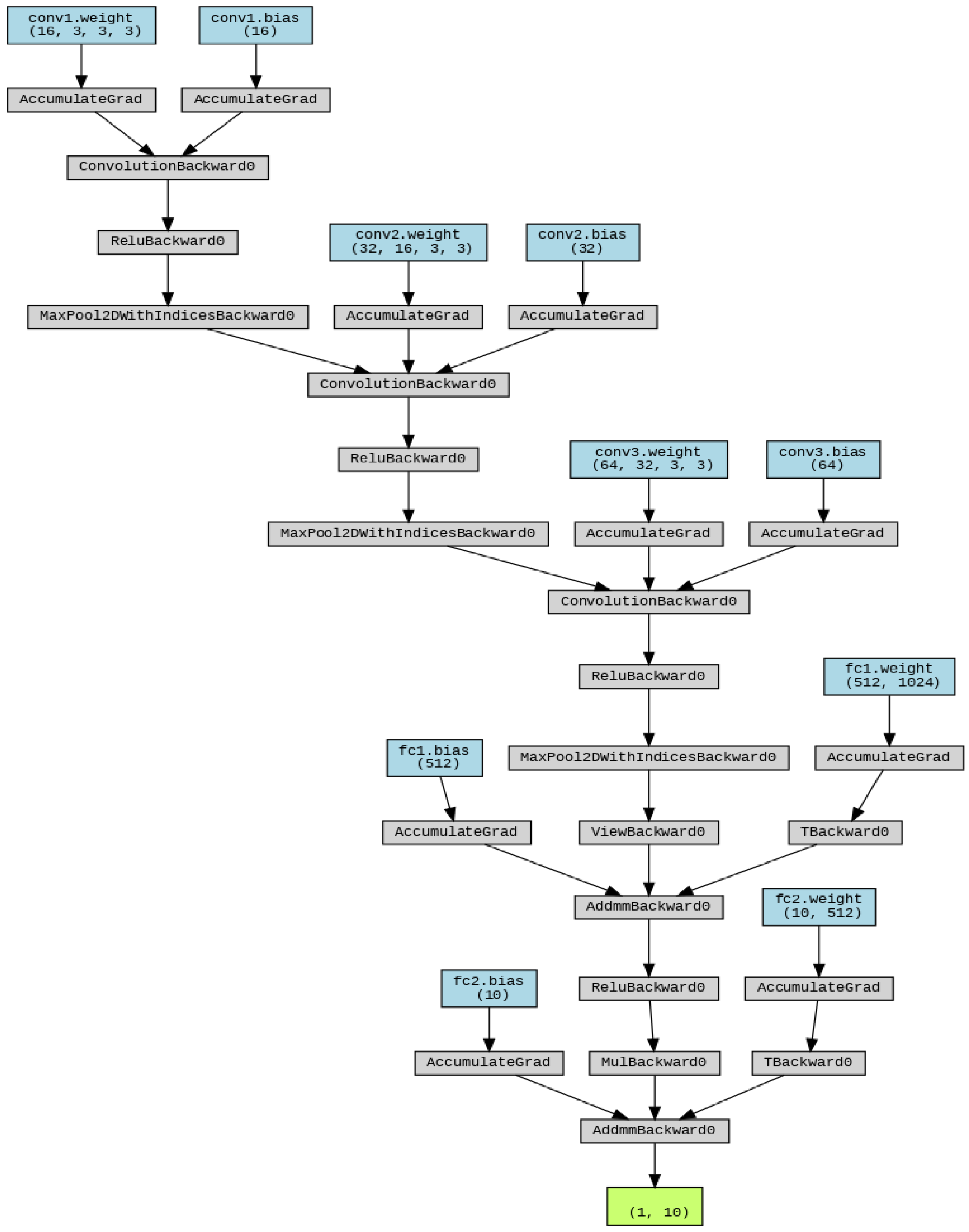
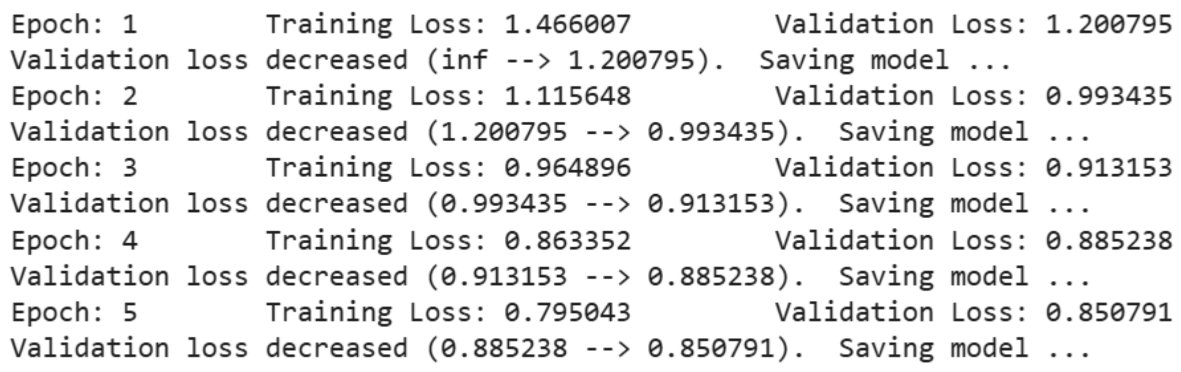


Fig 10: CNN Architecture

The visualization provides insight into how data flows through the network from convolution operations to activation functions to pooling layers to fully connected layers. The network begins with three convolutional layers, possessing max-pooling operations later. It progressed through the entire connected layers and ended with a classification softmax activation function.

Training Process

- Loss Function: Categorical Cross-Entropy Loss (nn.CrossEntropyLoss()).
- Optimizer: Adam optimizer with a learning rate of 0.001.
- Epochs: The model was trained for 5 epochs.
- Training Strategy:
    - During each epoch, the model iterated over training batches, performing forward and backward passes.
    - Training loss was accumulated and averaged for monitoring.
    - The model was validated on the validation set at the end of each epoch.
    - The best model was saved based on minimum validation loss.



```
Epoch: 1        Training Loss: 1.466007         Validation Loss: 1.200795
Validation loss decreased (inf --> 1.200795).  Saving model ...
Epoch: 2        Training Loss: 1.115648         Validation Loss: 0.993435
Validation loss decreased (1.200795 --> 0.993435).  Saving model ...
Epoch: 3        Training Loss: 0.964896         Validation Loss: 0.913153
Validation loss decreased (0.993435 --> 0.913153).  Saving model ...
Epoch: 4        Training Loss: 0.863352         Validation Loss: 0.885238
Validation loss decreased (0.913153 --> 0.885238).  Saving model ...
Epoch: 5        Training Loss: 0.795043         Validation Loss: 0.850791
Validation loss decreased (0.885238 --> 0.850791).  Saving model ...
```

Fig 11: Snippet of Training loss and validation loss after each Epoch

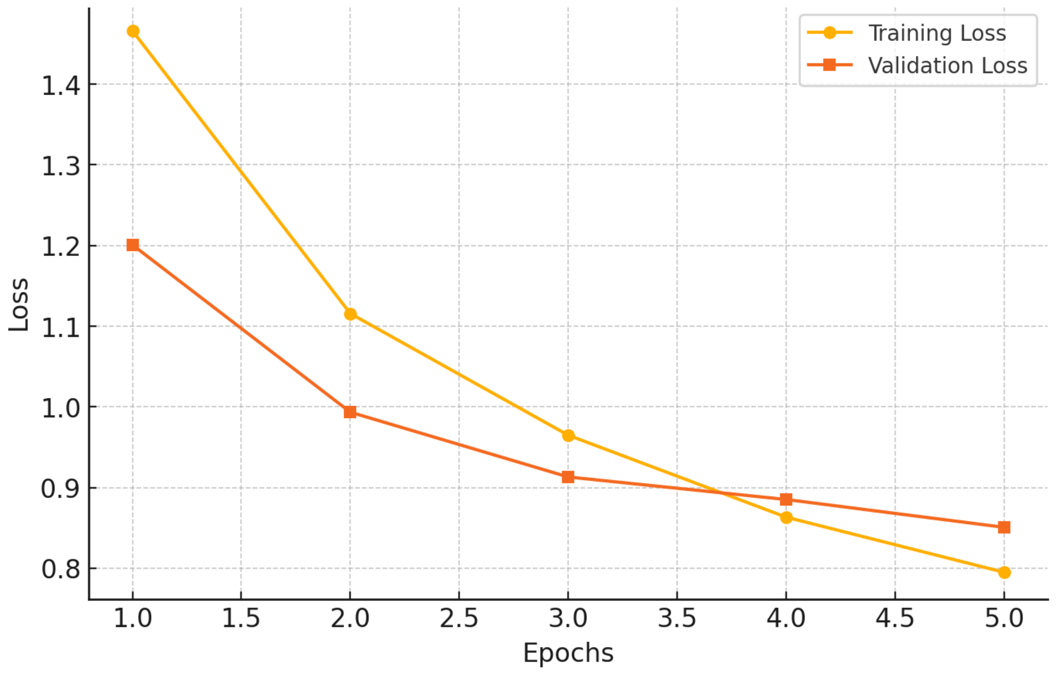


Fig 12: Plotting Training loss and validation loss over Epochs

Evaluation and Findings

The trained model was evaluated using the test dataset. The performance metrics obtained were as follows:

- Loss:
  - The final test loss was recorded after assessing the trained model on unseen data.
- Accuracy:
  - Individual class accuracies were used to further evaluate model performance across various categories.
  - The overall test accuracy was taken as the ratio of correctly classified images.
- Misclassifications:
  - Some misclassified examples were cast in visualization, pointing to confusion among the more alike categories (i.c., cat vs. dog, automobile vs. truck).

Visualizations

- A graphical representation of the CNN architecture was plotted.
- Sample images from the dataset were displayed along with their true and predicted labels.

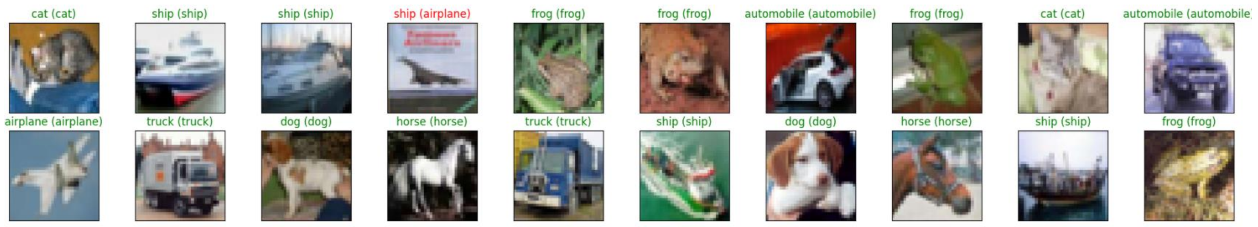
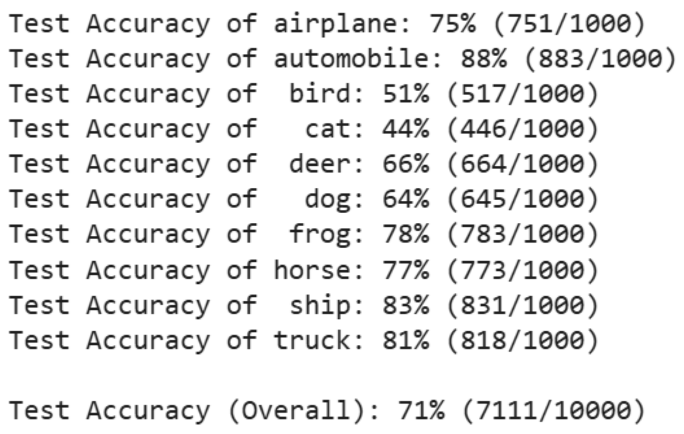


Fig 13: Sample images with true and predicted labels

- Misclassified images were highlighted in red, while correctly classified images were marked in green.
- The loss curves for training and validation were plotted to visualize the learning process.

Conclusion and Future Work

The CNN model demonstrated effective classification performance on the CIFAR-10 dataset. The overall accuracy is over 71 percent. However, there are potential areas for improvement:



```
Test Accuracy of airplane: 75% (751/1000)
Test Accuracy of automobile: 88% (883/1000)
Test Accuracy of  bird: 51% (517/1000)
Test Accuracy of   cat: 44% (446/1000)
Test Accuracy of  deer: 66% (664/1000)
Test Accuracy of   dog: 64% (645/1000)
Test Accuracy of  frog: 78% (783/1000)
Test Accuracy of horse: 77% (773/1000)
Test Accuracy of  ship: 83% (831/1000)
Test Accuracy of truck: 81% (818/1000)

Test Accuracy (Overall): 71% (7111/10000)
```

Fig 14: Snippet of total overall accuracy

- Hyperparameter Tuning: Experimenting with different learning rates, optimizers, and batch sizes could improve accuracy.
- Data Augmentation: Introducing transformations such as random cropping, rotation, and flipping could enhance generalization.
- Deeper Architectures: Using additional convolutional and fully connected layers could further refine feature extraction.
- Transfer Learning: Employing pre-trained networks such as ResNet or VGG could boost accuracy without extensive training.