# Object Classification System for RGB Images

**Course Project - COSC 6324**

## 

## Team

Venkata Santosh Valli Sunkarapalli

Valluri Shalini

Valluri Sai Prasanna

Veerla Sri Nikitha

## 1.Objective

\*Student 1: Data Preparation and Preprocessing\*

- \*Dataset Management\*

- Download the dataset and extract files from the provided zip.

- Ensure all necessary files are placed in the correct directory.

- \*Data Preprocessing\*

- Convert RGB images to grayscale using a weighted sum method.

- Store the processed grayscale images in a format suitable for the model.

- \*Dataset Splitting\*

- Divide the dataset into training and testing sets.

- Ensure the data is properly formatted and prepared for use in training.

\*Student 2: Model Development and Training\*

- \*Model Architecture\*

- Design and implement the Convolutional Neural Network (CNN) architecture.

- Define the layers and structure of the CNN, including convolutional, pooling, and fully connected layers.

- \*Loss Function and Optimizer\*

- Initialize the Cross Entropy Loss function for classification.

- Set up the Adam optimizer with a learning rate of 0.001 for training.

\*Student 3: Training Procedure and Hyperparameter Tuning\*

- \*Training Procedure\*

- Implement the training loop, including forward pass, loss computation, and backpropagation.

- Monitor the training progress and adjust hyperparameters as necessary.

- \*Hyperparameter Tuning\*

- Experiment with different hyperparameters to improve model performance.

- Document the impact of various hyperparameters on the training results.

\*Student 4: Evaluation and Results\*

- \*Model Evaluation\*

- Implement the testing function to evaluate the model on the test dataset.

- Compute and report classification accuracy and other evaluation metrics.

- \*Results Visualization\*

- Create visualizations for the classification results.

- Plot training loss over epochs and other relevant metrics.

- \*Documentation\*

- Write a detailed report or documentation outlining the project steps, results, and findings.

- Ensure all instructions and explanations are clear and comprehensive for users.

## 2. Background

### Introduction

Convolutional Neural Networks (CNNs) are a type of deep learning model tailored for processing grid-like data such as images. CNNs have dramatically transformed computer vision and image recognition by learning and extracting hierarchical representations from visual data. The architecture of CNNs includes convolutional layers, pooling layers, and fully connected layers. Each layer has a distinct role in feature extraction and classification:

* Convolutional Layers: Apply filters to images to detect specific features.
* Pooling Layers: Downsample feature maps, reducing spatial dimensions and controlling overfitting.
* Fully Connected Layers: Utilize extracted features to classify input images into categories.

CNNs excel at learning spatial hierarchies of patterns, enabling them to differentiate objects within images automatically without manual feature extraction.

### Project Goal

The goal is to build a system that classifies objects in RGB images using CNNs. The system will be trained on a dataset of RGB images, fine-tuned for optimal performance, and aimed at achieving high classification accuracy.

## 3. Project Description

### 3.1 Filters in CNNs

Filters, or kernels, are small matrices used in CNNs to extract features such as edges, textures, patterns, or shapes from images. These filters are convolved with input images to produce feature maps that highlight important patterns.

#### Role of Filters:

* Feature Extraction: Perform operations like edge detection, blurring, sharpening, and embossing.
* Translation Invariance: Recognize patterns regardless of their position by using shared weights.
* Dimensionality Reduction: Reduce image dimensions, making processing more feasible.
* Hierarchical Representation: Capture increasingly complex features across layers.

#### Filter Size and Stride:

* Filter Size: Larger filters capture complex patterns; smaller filters detect simpler features.
* Stride: Determines the step size of the filter movement. Larger strides reduce feature map size and computation, while smaller strides retain more spatial information.

#### Examples of Filters:

* Edge Detection: Sobel, Prewitt, and Roberts filters.
* Blur Filters: Gaussian blur filter.
* Sharpening Filters: Laplacian filter.
* Convolutional Filters in CNNs: Adapt and extract specific features during training.

### 3.2 Feature Vector Size Calculation

Assuming an input image size of 32x32 and max-pooling layers with a 2x2 window and stride of 2, the feature vector size can be calculated as follows:

1. Input Image: 32 × 32
2. After Max-Pooling Layer 1: 16 × 16 for 32 channels
3. After Max-Pooling Layer 2: 8 × 8 for 64 channels
4. After Max-Pooling Layer 3: 4 × 4 for 128 channels
5. After Max-Pooling Layer 4: 2 × 2 for 256 channels
6. After Max-Pooling Layer 5: 1 × 1 for 512 channels

Resulting feature vector size: 512 × 1 or 512.

### 3.3 Convolution and Pooling Layers

#### Convolutional Layers:

* Layer 1: Input 1, output 32, kernel 3x3
* Layer 2: Input 32, output 64, kernel 3x3
* Layer 3: Input 64, output 128, kernel 3x3
* Layer 4: Input 128, output 256, kernel 3x3
* Layer 5: Input 256, output 512, kernel 3x3

#### Pooling Layers:

* Max-Pooling Layer 1: 2x2 window, stride 2
* Max-Pooling Layer 2: 2x2 window, stride 2
* Max-Pooling Layer 3: 2x2 window, stride 2
* Max-Pooling Layer 4: 2x2 window, stride 2
* Max-Pooling Layer 5: 2x2 window, stride 2

These layers extract features at different abstraction levels, with convolutional layers creating feature maps and pooling layers downsampling them.

### 3.4 Fully Connected Layers

* Layer 1: Input size 512, output size 512
* Layer 2: Input size 512, output size 10 (number of classes)

The output from the last fully connected layer is fed into a softmax activation function to generate the probability distribution over classes, predicting the class label with the highest probability.

### 3.5 Loss Function

The system's loss function and the rationale behind its selection are critical but not provided in the original text.

### 3.6 Activation Functions

The Rectified Linear Unit (ReLU) activation function is chosen for its simplicity and effectiveness. ReLU is applied after each convolutional and linear layer in the network.

#### Reasons for Choosing ReLU:

* Non-linearity: Enables the network to learn and represent complex patterns.
* Computational Efficiency: Involves simpler mathematical operations, allowing faster convergence.
* Sparse Activation: Produces sparse activations by setting negative values to zero, helping prevent saturation.

### 3.7 Pooling Functions

Max pooling is used after each convolutional layer to downsample the feature maps.

#### Reasons for Choosing Max Pooling:

* Translation Invariance: Identifies key features regardless of position.
* Overfitting Reduction: Decreases spatial dimensions, preventing overfitting.
* Computational Efficiency: Simple operations make it faster compared to other techniques.
* Feature Invariance: Fosters invariance to small variations, aiding in capturing general patterns.

### 3.8 Training Stop Criteria

The training procedure stops based on the gradient norm. Training stops when the gradient norm of model parameters falls below a threshold, indicating convergence or sufficiently small gradients.

## 4. Experiments and Results

### 4.1 Learning Rate Determination

The learning rate (η) controls the step size during optimization. A value of 0.001 is a common starting point, often effective across different models and datasets, but may require experimentation for optimal results.

### 4.2 Training Duration

On CPU:

* Batch Size 1, 1 Epoch: ~10 minutes
* Batch Size 1, 10 Epochs: ~90 minutes

On GPU:

* Batch Size 1, 1 Epoch: ~20 seconds
* Batch Size 1, 10 Epochs: ~3 minutes

### 4.3 System Accuracy

The system's best accuracy is 9.77%.

## 5. Pros and Cons

### Pros:

* Deep Architecture: Allows learning complex hierarchical features.
* Max Pooling Layers: Aid in downsampling and retaining important features.
* ReLU Activation: Promotes faster convergence and mitigates the vanishing gradient problem.
* Adaptive Average Pooling: Handles variable-sized input images.
* Large Model Capacity: High capacity to learn intricate patterns.

### Cons:

* Complexity and Overfitting: High capacity may lead to overfitting on small datasets without regularization.
* Computational Demands: Requires significant resources for training and inference.
* Gradient Problems: Susceptible to vanishing or exploding gradients in deep networks.
* Limited Interpretability: Complexity makes understanding and interpreting decisions challenging.
* Hyperparameter Sensitivity: Performance depends heavily on hyperparameter tuning.

## 6. Further Improvements

* Data Augmentation: Increase training data diversity to improve generalization.
* Regularization: Implement techniques like dropout or batch normalization to reduce overfitting.
* Learning Rate Scheduling: Experiment with learning rate schedules for optimal training.
* Hyperparameter Tuning: Search for the best hyperparameters for optimal performance.
* Model Ensemble: Combine multiple predictions to improve performance.
* Transfer Learning: Utilize pre-trained models and fine-tune on the specific dataset.
* Gradient Clipping: Prevent gradient explosion or vanishing during training.
* Advanced Optimization Algorithms: Experiment with algorithms like Adam, RMSprop, or AdaGrad to improve performance.

# Appendix

**Training Hyperparameters**

* Loss Function: Cross Entropy Loss
* Optimizer: Adam
* Learning rate: 0.001
* Number of epochs: 10
* Batch size: 1 (for DataLoader)
* Gradient Norm Threshold Value: 1.0 x 10-4 (for early stopping based on gradient norm)

Model architecture:

class CNN(nn.Module):

def \_\_init\_\_(self):

super(CNN, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(1, 32, 3, padding=1)

self.relu1 = nn.ReLU()

self.maxpool1 = nn.MaxPool2d(2, 2)

self.conv2 = nn.Conv2d(32, 64, 3, padding=1)

self.relu2 = nn.ReLU()

self.maxpool2 = nn.MaxPool2d(2, 2)

self.conv3 = nn.Conv2d(64, 128, 3, padding=1)

self.relu3 = nn.ReLU()

self.maxpool3 = nn.MaxPool2d(2, 2)

self.conv4 = nn.Conv2d(128, 256, 3, padding=1)

self.relu4 = nn.ReLU()

self.maxpool4 = nn.MaxPool2d(2, 2)

self.conv5 = nn.Conv2d(256, 512, 3, padding=1)

self.relu5 = nn.ReLU()

self.maxpool5 = nn.MaxPool2d(2, 2)

self.avgpool = nn.AdaptiveAvgPool2d(1)

self.fc1 = nn.Linear(512, 512)

self.relu6 = nn.ReLU()

self.fc2 = nn.Linear(512, 10)

def forward(self, x):

x = self.conv1(x)

x = self.relu1(x)

x = self.maxpool1(x)

x = self.conv2(x)

x = self.relu2(x)

x = self.maxpool2(x)

x = self.conv3(x)

x = self.relu3(x)

x = self.maxpool3(x)

x = self.conv4(x)

x = self.relu4(x)

x = self.maxpool4(x)

x = self.conv5(x)

x = self.relu5(x)

x = self.maxpool5(x)

x = self.avgpool(x)

x = torch.flatten(x, 1)

x = x.squeeze()

x = self.fc1(x)

x = self.relu6(x)

x = self.fc2(x)

return x

Outputs and screenshots:





