**1. Project Overview and Findings**

This project focuses on classifying images from the CIFAR-10 dataset using a deep learning model. The primary goal was to develop an accurate image classifier and apply it to an unlabeled dataset for generating predictions.

Key findings from the project include:

The trained model achieved a high classification accuracy of 94%, demonstrating the effectiveness of the chosen architecture and training process.

Standard preprocessing techniques, such as normalization using CIFAR-10 mean and standard deviation, significantly improved the model’s performance.

The model was successfully deployed on an unlabeled test set, and predictions were stored in a CSV file for further evaluation.

**2 Methodology**

**1. Model Design**

Our approach focuses on designing a lightweight ResNet variant with competitive classification performance while maintaining a low parameter count (<5M) to improve computational efficiency. The key modifications in our architecture include:

• **Lightweight Residual Blocks**: We use a modified residual block structure where each block consists of two 3 \times 3 convolutional layers with batch normalization and ReLU activation. To maintain the shortcut connection when the number of channels changes, we introduce a 1 \times 1 convolution.

• **Three-Stage Hierarchical Design**: The network consists of three main stages, each reducing the spatial dimensions while increasing the number of channels (64 → 128 → 256). This ensures a balance between feature extraction capability and computational efficiency.

• **Global Average Pooling**: Instead of using fully connected layers with large weight matrices, we apply global average pooling before the final classification layer, significantly reducing the number of trainable parameters.

The overall architecture consists of an initial convolutional layer followed by three stacked residual blocks and a final classification layer. The detailed structure is shown below:

| **Layer Type** | **Output Size** | **Configuration** |
| --- | --- | --- |
| Conv1 | 32×32×64 | 3 x 3, stride 1 |
| Residual Block 1 | 32×32×64 | 3 x 3, 3 x 3 ×2 |
| Residual Block 2 | 16×16×128 | 3 x 3, 3 x 3 ×2, stride 2 |
| Residual Block 3 | 8×8×256 | 3 x 3, 3 x 3 ×2, stride 2 |
| Global Avg Pool | 1×1×256 | Adaptive Pooling |
| Fully Connected | 10 | Linear Layer |

This structure ensures that our model retains the benefits of residual learning while keeping the number of parameters low.

**2. Training Process**

To improve the generalization ability of our model, we incorporate various data augmentation techniques and carefully tune the training strategy.

**Data Augmentation**

ance model generalization and robustness, we apply several data augmentation techniques during training. **Random horizontal flipping** is used to introduce spatial variability, ensuring that the model does not overfit to specific orientations. **Random cropping with padding** helps simulate natural variations in object positioning by shifting objects within the frame. **Color jittering** modifies brightness, contrast, and saturation to improve the model’s resilience to different lighting conditions. **Random rotation** is applied to introduce orientation invariance, allowing the model to recognize objects regardless of minor rotational differences. Finally, **random erasing (Cutout)** is employed to encourage the network to focus on discriminative features rather than specific pixel locations, thereby improving its robustness to occlusions.

**Training Configuration**

We use the **CIFAR-10** dataset for training and set the **batch size** to 128. For optimization, we choose **Stochastic Gradient Descent (SGD) with momentum 0.9**, which is known to provide better generalization performance compared to Adam in image classification tasks. The initial **learning rate** is set to **0.01**, and we employ a **cosine annealing learning rate schedule** to ensure a smooth and gradual decrease in learning rate over time, helping the model converge effectively. Additionally, we apply **weight decay (5 × 10⁻⁴)** as a form of regularization to mitigate overfitting. The model is trained for **200 epochs** to ensure sufficient learning while balancing computational efficiency.

**Training Monitoring**

To systematically track training progress, we implement multiple monitoring mechanisms. We print **real-time batch loss** every **50 iterations** to provide immediate feedback on optimization behavior. To analyze model performance over time, we **compute and store per-epoch training loss and test accuracy**. Furthermore, the model is evaluated **every 5 epochs**, with **phase summaries printed every 10 epochs** to highlight performance trends. At the end of training, we **save the best-performing model** using PyTorch’s torch.save() function, ensuring that the optimal model state is preserved for further evaluation or deployment.

**3. Pros and Cons of Architectural Choices**

Our chosen architecture strikes a balance between accuracy and computational efficiency. One of its primary advantages is its **lower computational cost**, as it requires fewer parameters and operations compared to deeper ResNet variants. This makes it well-suited for deployment in resource-constrained environments. Additionally, the model exhibits **improved generalization** due to the effective use of data augmentation techniques, such as random cropping, color jittering, and cutout, which help mitigate overfitting. Moreover, the architecture enables **faster training and inference**, reducing both training time and memory requirements while maintaining competitive performance.

However, our approach also has limitations. The **limited feature extraction capacity** of a shallower network can hinder its ability to capture intricate patterns in highly complex datasets. This constraint may lead to suboptimal performance on tasks requiring deeper hierarchical representations. Furthermore, the **risk of underfitting** remains a concern, as a lightweight design might struggle to learn high-level abstract features as effectively as deeper architectures.

**4. Lessons Learned**

Throughout the model design and training process, we derived several key insights that contributed to optimizing performance and efficiency. First, **data augmentation plays a crucial role in generalization**. Techniques such as cutout and color jitter significantly enhanced the model’s robustness by simulating real-world variations in input data. Second, **optimizer selection impacts training stability**. We found that stochastic gradient descent (SGD) with momentum led to more stable convergence and better final accuracy compared to Adam, which often exhibited higher variance in performance.

Additionally, we observed that **architectural trade-offs are critical in balancing efficiency and accuracy**. While reducing the number of parameters enhances computational efficiency, excessive simplifications can degrade performance, particularly on complex datasets. Lastly, **learning rate scheduling proved to be essential for effective training**. The adoption of a cosine annealing schedule allowed the model to avoid premature convergence, enabling a smoother optimization trajectory and ultimately leading to better generalization.

Our final model achieves strong performance while maintaining computational efficiency, making it well-suited for deployment in environments with limited resources.

**3. Results**

Final Test Accuracy: 94%

Model Architecture:

A deep CNN-based model with convolutional layers, batch normalization, and ReLU activations.

Number of Parameters: ~5 million (varies depending on architecture details).

Output Format: The predicted labels were successfully saved in submission.csv with the following format:

python-repl

ID,Label

0,3

1,5

2,2

...

This file was stored in /kaggle/working/ and verified using os.listdir() to ensure successful export.

This project successfully demonstrated the use of deep learning for image classification, achieving a high accuracy rate and effective predictions on an unlabeled dataset. Future improvements could include model ensembling, fine-tuning with transfer learning, or augmenting the training data for better generalization.