Noah Wiley
Ruby
CSCI 167
December 1, 2025

CIFAR-10 Image Classification Using Convolutional Neural Networks

Introduction:
The purpose of this project was to design, train, and evaluate deep learning models capable of performing image classification using the CIFAR-10 dataset. This project provided an opportunity to apply core concepts from the course that include CNNs, training strategies, and hyperparameters tuning to a real world dataset and investigate how architectural choices affect model performance. By iteratively modifying the model architecture and optimizing hyperparameters, the goal was to improve classification accuracy while gaining a deeper understanding of how different design decisions influence learning behavior and generalization.

The CIFAR-10 dataset was selected due to its widespread use as a benchmark in deep learning research, its appropriate size for iterative experimentation, and its sustainability for CNN-based approaches covered in the course. The project began with a baseline CNN model and progressed through multiple stages of refinement. Ultimately, the work culminated in a deeper CNN with batch normalization and progressively increasing filter sizes, achieving substantially higher performance than the initial baseline.

Dataset Overview:
The CIFAR-10 dataset contains 60,000 color images, each measuring 32x32 pixels with three RGB channels. These images are divided into 10 equally represented classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. The dataset includes 50,000 training images and 10,000 test images, with no class imbalance. The balance makes CIFAR-10 particularly well suited for analyzing model performance without the confounding effects of uneven label distribution.

The preprocessing pipeline consisted of two primary steps. First, images were converted to PyTorch tensors using `transforms.ToTensor()`, normalizing pixel values to the range [0,1]. Second, inputs were standardized using `transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))`, which shifts each color channel to have mean 0 and standard deviation 1, effectively mapping values to approximately [-1.-1]. This preprocessing accelerates convergence by stabilizing gradients and ensuring consistent input scaling.
Although CIFAR-100 was also considered, its larger number of classes introduced greater complexity and longer training times. For practicality and cleaner analysis, the project focused exclusively on CIFAR-10.

Model Design & Rationale
Baseline Architecture - To begin, a small reference CNN from a PyTorch tutorial was used as the baseline. This model contained:

- Two convolutional layers (6 and 16 filters)
- Max-pooling after each convolution

- Three fully connected layers

The baseline achieved approximately 60% accuracy. While useful for understanding workflow and debugging, its shallow architecture lacked the capability to extract sufficiently rich features from the data. This motivated the development of deeper architectures

Final Architecture - The final model was a substantially deeper CNN designed to progressively learn increasingly complex spatial features. The key architectural choices were:

- **Five convolutional layers:** Filters increased as: *32 → 64 → 128 → 256 → 512*.
  - This follows a common design principle in successful CNNs, where deeper layers capture higher-level patterns.
- **Batch normalization after every convolution:** BatchNorm stabilizes learning, reduces internal covariate shift, and allows for higher learning rates.
- **ReLU activation:** Provides non-linearity and encourages sparse feature activations.
- **Max-pooling after the first four convolutions only:** Keeping spatial dimensions too small (e.g., 1×1 feature maps) severely harmed performance in earlier attempts. Limiting pooling preserved more spatial information while still reducing computational cost.
- **Two fully connected layers:** Flattened features (512 × 2 × 2 → 2048 units)
  - → 64-unit hidden layer → 10-class output layer.

This final architecture produced significantly stronger learning capacity, enabling the model to reach 78% test accuracy.

Experimental Setup & Hyperparameter Exploration

Throughout development, hyperparameters were systematically adjusted and evaluated. Key experiments included:

- *Learning Rate & Optimizer*
  - Initially tested SGD with momentum (0.9) and learning rate 0.001 which had small improvements in loss, minimal accuracy gains.
  - Switched to Adam, which adapts learning rates per parameter which improved training stability but did not drastically increase accuracy alone.
  - The final learning rate of 0.001 yielded consistently strong performance.
- *Batch Size*
  - Started with very small batch sizes (4), as in the tutorial but it extremely slowed the training, noisy gradients.
  - Increasing to 128 significantly accelerated training and improved accuracy.
- *Number of Epochs*
  - Experiments ranged from 2 to 30 seconds
  - Training curves revealed diminishing returns beyond 20 epochs, so the final model used 20
- *Architecture Focused Hyperparameters*
  - Increasing filters dramatically improved performance, whereas large filter counts in early layers caused overfitting or degraded accuracy.

    - Adding more convolutional layers improved accuracy up to a point… performance plateaued beyond five conv layers.
    - Batch normalization contributed an additional 2-3% performance improvement and stabilized convergence.

These experiments highlighted that architectural choices were more influential than optimizer hyperparameters alone.

Results and Analysis:

Training Performance - Training accuracy rose steadily across epochs and eventually reached approximately 98%, while loss decreased sharply. This demonstrates effective learning and a strong fit to the training data. Training plots also revealed that the model converged reliably within 20 epochs.

Test Performance - The final model achieved:

- 78% accuracy on the test set
- Precision: ~0.80
- Recall: ~0.79
- F1-Score: ~0.79

Given the balanced dataset, precision, recall, and F1-score aligned closely. Per-class performance metrics revealed strong results across most classes but highlighted predictable confusions, such as:

- Cats ↔ Dogs
- Trucks ↔ Automobiles
- Birds ↔ Airplanes

Confusion Matrix - The confusion matrix provided a clear summary of correct and incorrect predictions across all classes. Strong diagonal values indicated correct classifications, while off-diagonal entries identified common misclassifications. This visualization helped reveal patterns not captured by accuracy alone.

Discussion, Limitations, and Lessons Learned

This project demonstrated how architectural choices, learning rates, and training strategies impact CNN performance on image classification tasks. The final model significantly outperformed the baseline, improving from 60% to 78% test accuracy.

Limitations -

- Filter and kernel sizes were only lightly explored due to limited intuition early on. More systematic tuning might yield additional gains.
- Training deeper models was time-consuming on available hardware, restricting the number of experiments.
- Only a single dataset was explored, preventing cross dataset comparisons

Lessons Learned -

- Architectural changes often have more impact than optimizer tweaks
- Batch normalization is essential when training deeper CNNs
- Over-pooling can destroy spatial features and severely harm accuracy
- Training curves are invaluable for identifying plateau points and avoiding unnecessary epochs.
- Small hyperparameter adjustments can have large effects on convergence speed.

Future Work -

With more time, further improvements could include:

- Testing different kernel sizes (1x1, 5x5), dilation or grouped convolutions
- Using regularization techniques such as dropout or weight decay
- Trying more advanced architectures
- Applying automated hyperparameter search tools