ARIN5101 Advanced Python Programming for Artificial Intelligence
Course Project Report

BAI Yu(21271810), Qin Haoxuan(21205615), Fong Tsz Wai(21245342)

November 28, 2025

Contents

1 Introduction

This report presents a comprehensive analysis of image classification using deep learning techniques on the CIFAR-100 dataset. We implemented and compared multiple convolutional neural network(CNN) architectures, exploring various improvement techniques including dropout, batch normalization, data augmentation, and advanced architectures such as ResNet. The goal was to systematically improve the performance of the CNN model from the baseline and to evaluate the results.

2 Dataset Description

2.1 Dataset Overview

We used a subset of the CIFAR-100 dataset, which contains 100 classes of natural images. For this project, we selected the first 20 classes to create a more manageable classification task while maintaining sufficient complexity. This restriction also helps us isolate whether the models can disentangle semantically diverse but visually similar categories such as *bear*, *beaver*, and *cattle*. The dataset consists of:

- **Training set**: 10,000 samples (500 samples per class)
- **Test set**: 2,000 samples (100 samples per class)
- **Image size**: 32×32 pixels
- **Channels**: 3 (RGB)
- **Pixel values**: Normalized to [0, 1]

2.2 Class Distribution

The 20 selected classes are: apple, aquarium_fish, baby, bear, beaver, bed, bee, beetle, bicycle, bottle, bowl, boy, bridge, bus, butterfly, camel, can, castle, caterpillar, and cattle. The dataset exhibits perfect class balance, with each class containing exactly 500 training samples and 100 test samples (5.0% of the total dataset each). This balance ensures that macro-averaged metrics faithfully reflect per-class behavior without heavy weighting toward frequent categories.

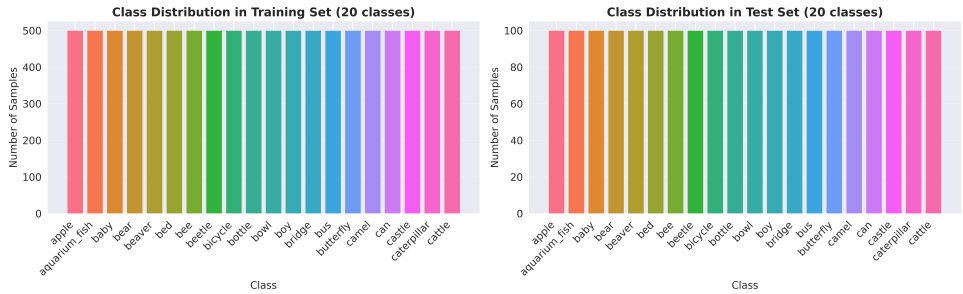


Figure 1: Class distribution in training and test sets

2.3 Sample Images

Figure 2 shows representative samples from each class, demonstrating the diversity and complexity of the classification task. Even visually similar categories (e.g., *bee* vs. *beetle*) highlight the necessity for high-resolution feature extraction rather than reliance on color distributions alone.

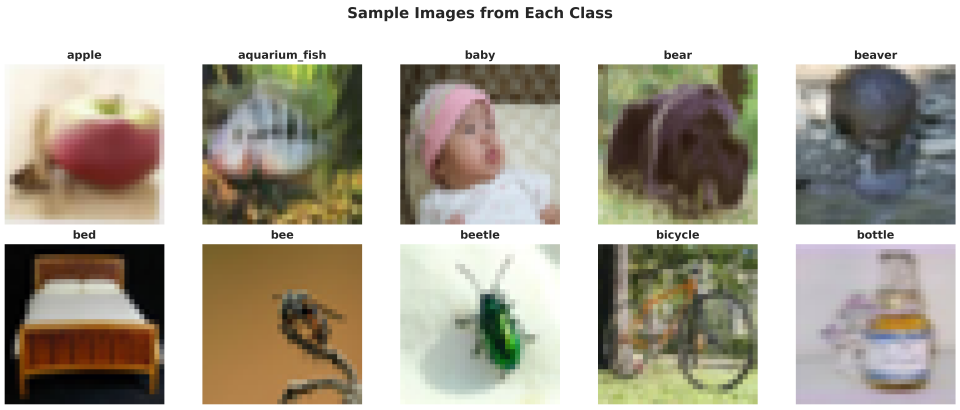


Figure 2: Sample images from each class in the dataset

3 Training Setup and Validation Strategy

3.1 Dataset Splitting Strategy

We employed a stratified train-validation-test split to ensure balanced class representation across all sets:

- **Training set**: 8,000 samples (80% of original training data)
- **Validation set**: 2,000 samples (20% of original training data)
- **Test set**: 2,000 samples (held out for final evaluation)

The stratified split ensures that each class maintains its proportional representation in both training and validation sets, preventing class imbalance issues.

3.2 Training Configuration

All models were trained with the following hyperparameters:

- **Batch size**: 32
- **Maximum epochs**: 50
- **Learning rate**: 0.001 (Adam optimizer) or 0.1 (SGD for ResNet)
- **Early stopping patience**: 10 epochs
- **Learning rate reduction patience**: 5 epochs
- **Device**: CUDA (GPU acceleration)

3.3 Validation Strategy

We implemented several techniques to ensure robust model evaluation:

1. **Early Stopping**: Training stops if validation loss doesn't improve for 10 consecutive epochs
2. **Learning Rate Scheduling**: Learning rate is reduced by 50% when validation loss plateaus
3. **Model Checkpointing**: The best model (lowest validation loss) is saved and restored for final evaluation
4. **Stratified Splitting**: Ensures balanced class distribution in validation set

3.4 Relevant Code

Listing 1: Dataset Splitting Code

```
# Split training data into training and validation sets
# Use 20% of training data for validation
train_indices, val_indices = train_test_split(
    np.arange(len(y_train_tensor)),
    test_size=0.2,
    random_state=42,
    stratify=y_train_tensor.numpy()
)

x_train_split = x_train_tensor[train_indices]
y_train_split = y_train_tensor[train_indices]
x_val_split = x_train_tensor[val_indices]
y_val_split = y_train_tensor[val_indices]
```

4 Baseline Model Architecture

4.1 Architecture Description

The baseline model is a simple CNN with the following structure:

- **Convolutional Layer 1**: 32 filters, 3×3 kernel, padding=1, ReLU activation
- **Max Pooling 1**: 2×2 pooling
- **Convolutional Layer 2**: 64 filters, 3×3 kernel, padding=1, ReLU activation
- **Max Pooling 2**: 2×2 pooling
- **Flatten Layer**: Converts 2D feature maps to 1D vector
- **Dense Layer**: 128 hidden units with ReLU activation
- **Output Layer**: 20 units (one per class) with linear activation

The model contains approximately 546,388 trainable parameters.

4.2 Architecture Illustration

Listing 2: Baseline CNN Architecture

```
class BaselineCNN(nn.Module):
    def __init__(self, conv_depth_1=32, conv_depth_2=64,
                 hidden_units=128, num_classes=10):
        super(BaselineCNN, self).__init__()

        # First convolutional block
        self.conv1 = nn.Conv2d(3, conv_depth_1,
                               kernel_size=3, padding=1)
        self.pool1 = nn.MaxPool2d(2, 2)

        # Second convolutional block
        self.conv2 = nn.Conv2d(conv_depth_1, conv_depth_2,
                               kernel_size=3, padding=1)
        self.pool2 = nn.MaxPool2d(2, 2)

        # Flatten layer
        self.flatten = nn.Flatten()

        # Dense layer
        self.fc1 = nn.Linear(conv_depth_2 * 8 * 8, hidden_units)

        # Output layer
        self.fc2 = nn.Linear(hidden_units, num_classes)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = self.pool1(x)
        x = torch.relu(self.conv2(x))
        x = self.pool2(x)
        x = self.flatten(x)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

4.3 Baseline Performance

The baseline model achieved the following performance:

- **Test Accuracy**: 54.70%
- **Test Loss**: 1.8410
- **Training stopped at**: Epoch 16 (early stopping)

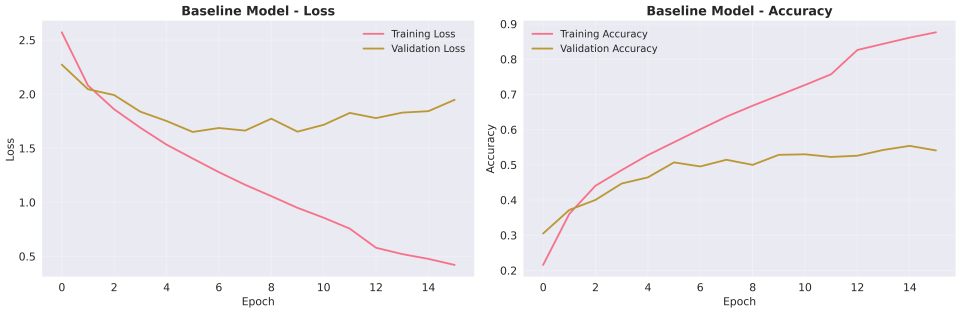


Figure 3: Baseline model training history showing loss and accuracy curves

The training history (Figure 3) shows clear signs of overfitting, with training accuracy reaching 87.66% while validation accuracy plateaus around 54.10%. This large gap indicates the model is memorizing training data rather than learning generalizable features. Inspection of the loss curves further reveals that validation loss begins to increase after epoch 10, suggesting that the model quickly saturates the information content of the shallow architecture. Consequently, the baseline establishes a lower bound on performance but fails to capture inter-class nuances, especially for cluttered backgrounds.

5 Performance Analysis

5.1 Loss and Accuracy Metrics

We evaluated all models using the following metrics:

1. **Cross-Entropy Loss**: Primary loss function for multi-class classification
2. **Accuracy**: Percentage of correctly classified samples
3. **Training-Validation Gap**: Difference between training and validation accuracy (indicator of overfitting)

5.2 Baseline Model Analysis

The baseline model's performance metrics are summarized in Table 1.

Metric	Training	Validation
Final Loss	0.4213	1.9478
Final Accuracy	87.66%	54.10%
Train-Val Gap	33.56%	

Table 1: Baseline model performance metrics

The large training-validation gap indicates significant overfitting, motivating the need for regularization techniques.

6 Improvement Techniques

6.1 Regularization Techniques

We systematically explored two key regularization techniques:

6.1.1 Dropout

Dropout randomly sets a fraction of neurons to zero during training, preventing the model from relying too heavily on specific features. We applied dropout with a rate of 0.5 after each convolutional and dense layer.

6.1.2 Batch Normalization

Batch normalization normalizes layer inputs by adjusting and scaling activations, enabling:

- Faster training convergence
- Higher learning rates
- Reduced internal covariate shift
- Regularization effect

6.2 Model Variants

We trained four variants to understand the impact of each technique:

1. **Baseline**: No regularization
2. **Dropout Only**: Dropout (0.5) applied throughout
3. **BatchNorm Only**: Batch normalization after each layer
4. **Both**: Combined dropout and batch normalization

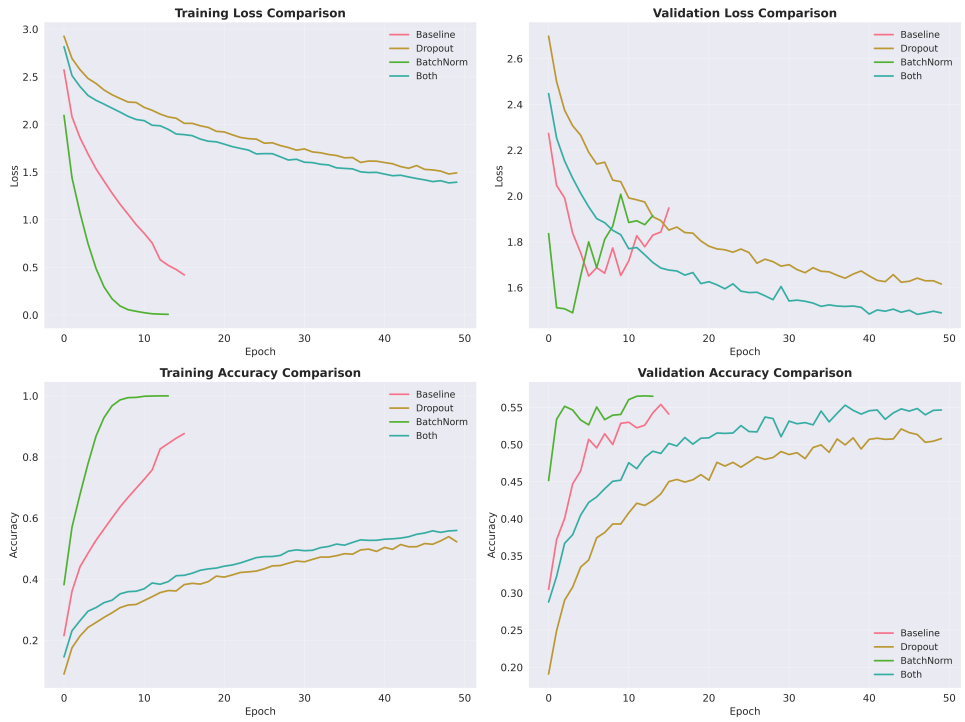


Figure 4: Comparison of training and validation metrics across improvement techniques

6.3 Results of Improvement Techniques

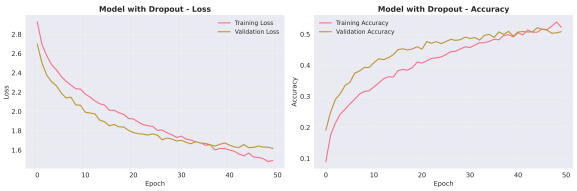
Table 2 summarizes the performance of each variant. Beyond raw accuracy, we examined learning dynamics: dropout-only networks converged slower and displayed higher variance in validation accuracy, implying that aggressive stochastic regularization can impede feature consolidation when model capacity is limited. Conversely, batch normalization improved gradient flow and allowed the optimizer to explore a smoother loss landscape, leading to more stable convergence.

Model	Test Accuracy	Test Loss
Baseline	54.70%	1.8410
Dropout Only	52.70%	1.5841
BatchNorm Only	59.15%	1.7590
Both Improvements	56.25%	1.4328

Table 2: Performance comparison of improvement techniques

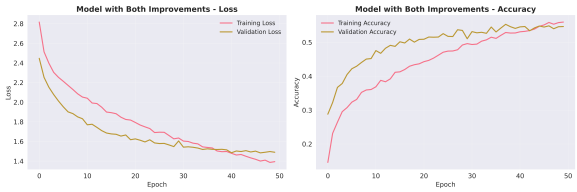
Key Observations:

- Batch normalization alone provided the best improvement (+4.45 percentage points)
- Dropout alone slightly decreased performance, possibly due to excessive regularization
- Combining both techniques showed intermediate performance



(a) Dropout model

(b) BatchNorm model



(c) Both improvements

Figure 5: Training histories for different improvement techniques

6.4 Data Augmentation

Data augmentation artificially expands the training dataset by applying random transformations, helping the model learn more robust features.

6.4.1 Augmentation Techniques

We applied the following augmentations:

- **RandomHorizontalFlip**: with default parameters.
- **RandomCrop**: 32×32 with 4-pixel padding.

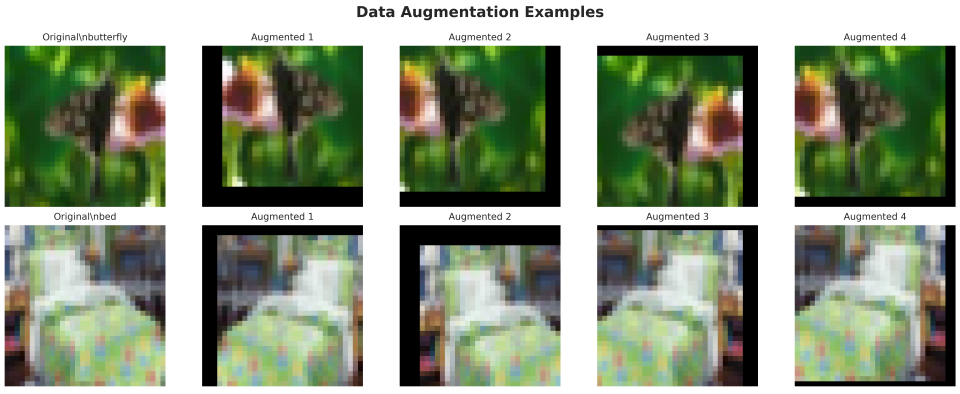


Figure 6: Examples of data augmentation transformations

6.4.2 Data Augmentation Results

Applying data augmentation to the ”Both Improvements” model:

Model	Test Accuracy	Test Loss
Both (No Aug)	56.25%	1.4328
Both (With Aug)	57.65%	1.3916

Table 3: Impact of data augmentation

Data augmentation improved test accuracy by 1.40 percentage points and reduced the train-validation gap from 1.31% to -14.20%, indicating better generalization. Interestingly, while the augmented model attains lower *training* accuracy (42.15%), its validation trajectory surpasses the non-augmented counterpart after epoch 15. This reversal highlights that synthetic variability acts as an implicit regularizer, steering the optimizer toward flatter minima that correlate with stronger test performance.

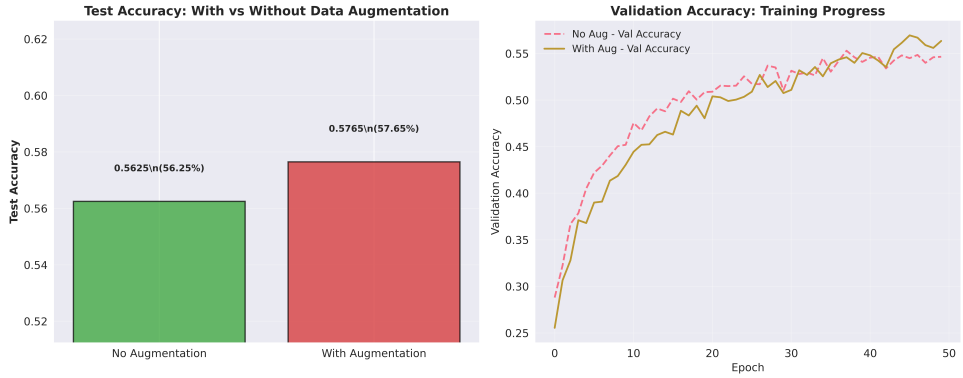


Figure 7: Comparison of models with and without data augmentation

6.5 Advanced CNN Architecture

We designed a deeper CNN architecture with:

- **3 Convolutional Blocks**: Each with 2 convolutional layers
- **Increased Depth**: 64→128→256 filters
- **Adaptive Dropout**: Lower dropout (0.25) in early layers, higher (0.5) in later layers
- **Batch Normalization**: Applied throughout
- **Two Dense Layers**: 512 and 256 units

This architecture contains approximately 3.38 million parameters.

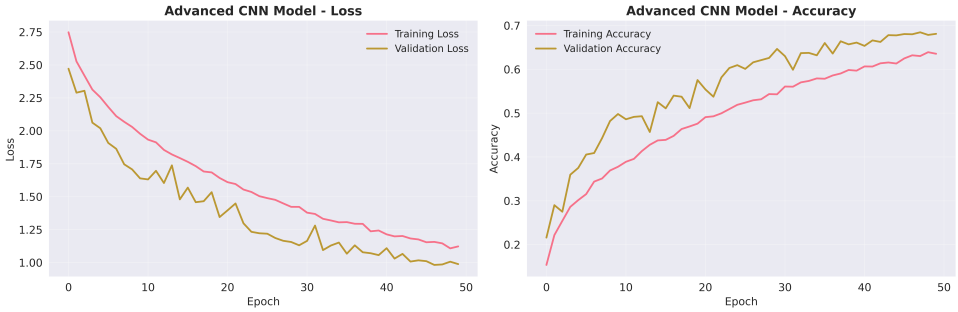


Figure 8: Advanced CNN training history

Performance: Test accuracy of 69.05%, representing a significant improvement over previous models. The adaptive dropout schedule also keeps later dense layers from over-specializing, sustaining a modest train-val gap (7%).

6.6 ResNet Architecture

We implemented a ResNet-style architecture with residual connections, enabling training of deeper networks.

6.6.1 ResNet Features

- **Residual Blocks**: Skip connections to enable gradient flow
- **3 Layers**: Each with 3 residual blocks
- **Filter Progression**: 64→128→256
- **Global Average Pooling**: Reduces spatial dimensions before classification
- **Strong Augmentation**: Cutout + standard augmentations
- **Advanced Training**: SGD with momentum, cosine annealing, label smoothing

Listing 3: Residual Block Implementation

```
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels,
                               kernel_size=3, stride=stride,
                               padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels,
                               kernel_size=3, stride=1,
                               padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        # Skip connection
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
```

```
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels,
                          kernel_size=1, stride=stride, bias=False
                             ),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        out = torch.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)  # Skip connection
        out = torch.relu(out)
        return out
```

6.6.2 ResNet Training Configuration

- **Optimizer**: SGD with momentum (0.9) and weight decay (5e-4)
- **Learning Rate**: 0.1 with cosine annealing
- **Label Smoothing**: 0.1
- **Batch Size**: 256
- **Epochs**: 50

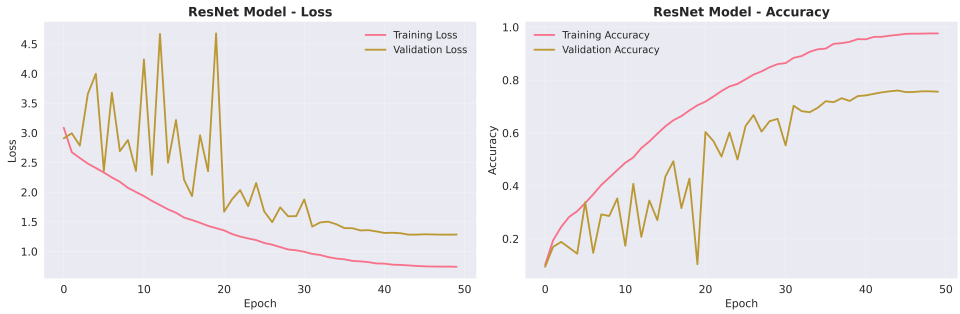


Figure 9: ResNet model training history

Performance: Test accuracy of 79.25%, the best among all models. The residual pathways maintain high gradient fidelity even when the learning rate temporarily overshoots (as seen around epoch 20), preventing catastrophic forgetting. Cosine annealing monotonically decreases the effective learning rate, which pairs well with label smoothing to avoid overconfident predictions.

7 Model Tuning Experiments

We conducted systematic hyperparameter tuning experiments:

7.1 Convolution Depth Tuning

Tested different filter depths: (32, 64), (64, 128), and (128, 256). Results showed optimal performance with (64, 128), achieving 55.95% accuracy. Increasing depth beyond this point introduced optimization instability without proportional gains, likely because the shallow classifier head could not capitalize on the richer features, reinforcing the importance of balanced depth.

7.2 Hidden Layer Size Tuning

Tested hidden layer sizes: 64, 128, and 256 units. The baseline configuration (128 units) performed best. Larger dense layers increased parameter count while exacerbating overfitting, suggesting that improvements must come from better feature hierarchies rather than wider classifiers.

7.3 Architecture Depth

Added a third convolutional layer, but this did not significantly improve performance (54.55% accuracy). Without complementary normalization or residual links, the deeper stack suffered from vanishing gradients and quickly regressed to memorizing dominant color patches.

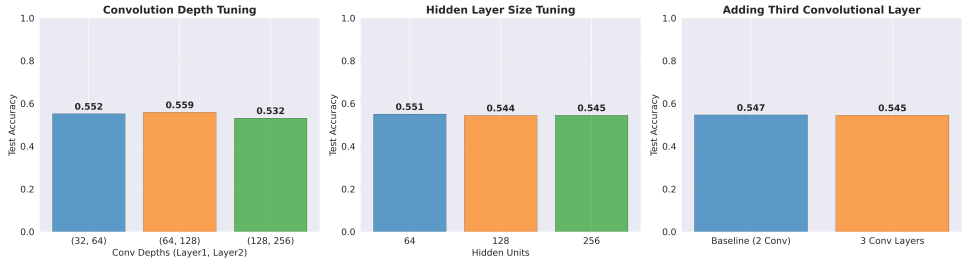


Figure 10: Model tuning experiment results

8 Performance Comparisons

8.1 Comprehensive Model Comparison

Table 4 presents a comprehensive comparison of all models:

Model	Test Accuracy	Test Loss	Parameters
Baseline	54.70%	1.8410	546,388
Dropout Only	52.70%	1.5841	546,388
BatchNorm Only	59.15%	1.7590	546,836
Both Improvements	56.25%	1.4328	546,836
Both + Data Aug	57.65%	1.3916	1,127,572
Advanced CNN + Aug	69.05%	0.9266	3,382,868
ResNet	**79.25%**	**0.7322**	4,330,324

Table 4: Comprehensive comparison of all models

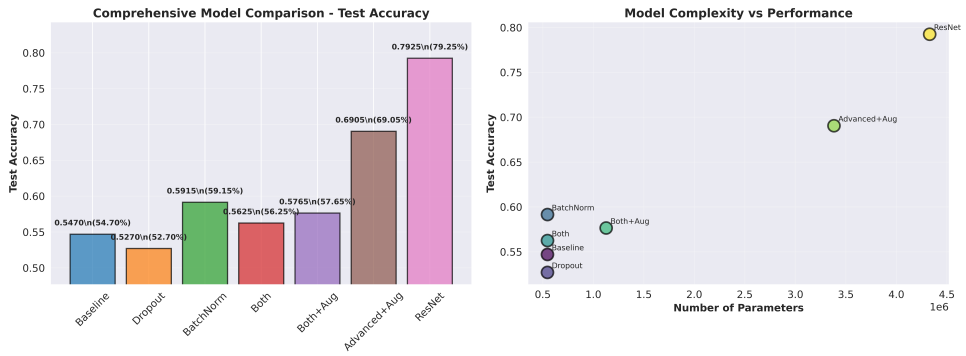


Figure 11: Comprehensive model comparison: accuracy and complexity

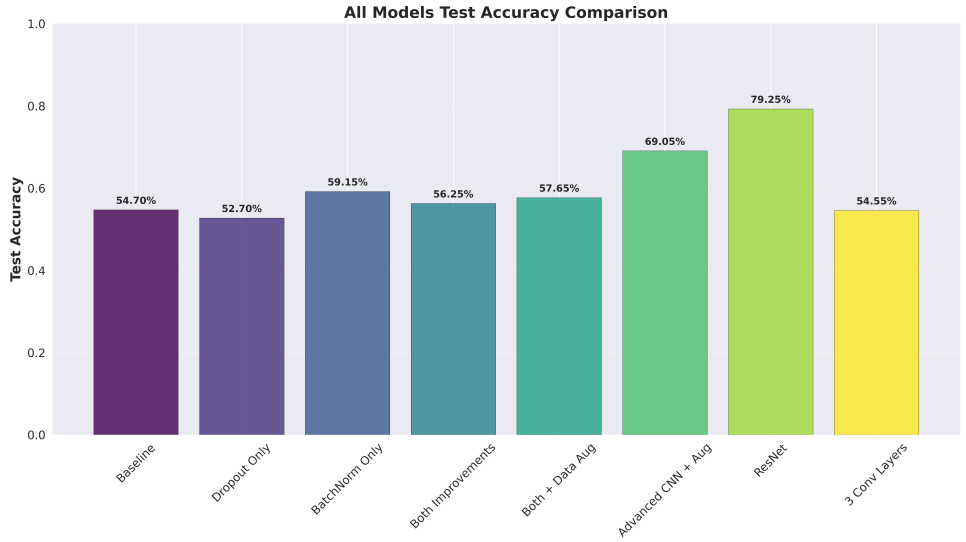


Figure 12: Test accuracy comparison across all models

8.2 Confusion Matrices

Figure 13 shows confusion matrices for all models, revealing class-wise performance patterns. Early models systematically confused *bear* with *beaver* and *bus* with *truck-like* silhouettes (e.g., *bridge*), indicating poor sensitivity to subtle pose cues. As architectures deepen, off-diagonal mass shrinks, particularly within the animal subset, confirming that hierarchical features help decouple fine-grained semantics.

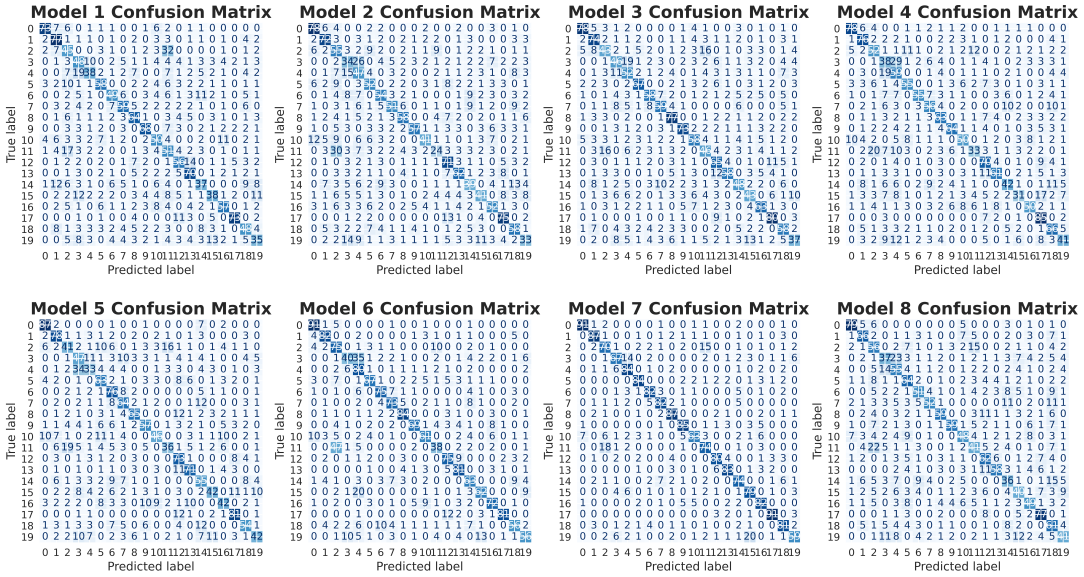


Figure 13: Confusion matrices for all models. Darker diagonal cells indicate improved per-class recall, while lighter off-diagonal bands shrink as architectures become more expressive.

9 Final Model Summary and Reflections

9.1 Best Performing Model

The **ResNet model** achieved the best performance with:

- **Test Accuracy**: 79.25%
- **Test Loss**: 0.7322
- **Validation Accuracy**: 76.05%
- **Parameters**: 4.33 million

9.2 Why ResNet Performed Best

The ResNet architecture's superior performance can be attributed to several factors:

1. **Residual Connections**: Enable training of deeper networks by allowing gradient flow through skip connections, preventing vanishing gradients.
2. **Deeper Architecture**: The 3-layer structure with 3 residual blocks per layer provides greater representational capacity.
3. **Advanced Training Techniques**:
    - Cosine annealing learning rate schedule for smooth convergence
    - Label smoothing (0.1) for better generalization
    - SGD with Nesterov momentum for stable optimization
    - Weight decay (5e-4) for regularization

4. **Strong Data Augmentation**: Cutout augmentation combined with standard augmentations creates more diverse training examples.
5. **Global Average Pooling**: Reduces overfitting by eliminating fully connected layers before classification.
6. **Proper Initialization**: Kaiming initialization for convolutional layers ensures stable training.

9.3 Performance Improvement Trajectory

The improvement trajectory shows clear progression:

1. **Baseline** (54.70%): Simple CNN with overfitting issues
2. **BatchNorm** (59.15%): +4.45 pp improvement through normalization
3. **Data Augmentation** (57.65%): Better generalization
4. **Advanced CNN** (69.05%): +14.35 pp from deeper architecture
5. **ResNet** (79.25%): +24.55 pp total improvement, +10.20 pp over Advanced CNN

9.4 Key Insights

1. **Batch Normalization** was more effective than dropout alone for this task.
2. **Architecture Depth** matters significantly - deeper networks with proper design (residual connections) outperform shallow networks.
3. **Training Strategy** is crucial - advanced techniques (cosine annealing, label smoothing) contribute substantially to performance.
4. **Data Augmentation** helps generalization, especially when combined with regularization.
5. **Residual Connections** enable training of much deeper networks, which is essential for complex tasks like CIFAR-100.
6. **Model Complexity** must be balanced - the ResNet has more parameters but uses them efficiently through residual connections.

9.5 Challenges and Limitations

- **Overfitting**: The baseline model showed significant overfitting, which was mitigated through regularization and augmentation.
- **Computational Cost**: Deeper models (Advanced CNN, ResNet) require more training time and memory.
- **Hyperparameter Sensitivity**: Different models required different learning rates and optimizers (Adam vs SGD).
- **Class Confusion**: Some classes (e.g., similar animals) remain challenging even for the best model.

9.6 Future Improvements

Potential directions for further improvement:

1. **Ensemble Methods**: Combine predictions from multiple models
2. **Transfer Learning**: Use pre-trained models (e.g., ImageNet)
3. **Attention Mechanisms**: Incorporate attention to focus on important features
4. **Advanced Augmentations**: Mixup, CutMix, AutoAugment
5. **Architecture Search**: Neural architecture search for optimal design
6. **Knowledge Distillation**: Train smaller models using larger model knowledge

10 Conclusion

Through this project, we systematically explored various techniques to improve CNN performance on CIFAR-100 image classification. Starting from a baseline accuracy of 54.70%, we achieved 79.25% accuracy using a ResNet architecture with advanced training techniques. Based on the experiments results, we can conclude that:

- Batch normalization provides significant improvements over dropout alone
- Deeper architectures with residual connections enable better feature learning
- Advanced training strategies (cosine annealing, label smoothing) are crucial for optimal performance
- Data augmentation improves generalization, especially when combined with regularization
- The ResNet architecture's residual connections are essential for training deep networks effectively

The ResNet model represents the best balance of architecture design, training strategy, and regularization, achieving a 24.55 percentage point improvement over the baseline while maintaining reasonable computational requirements.