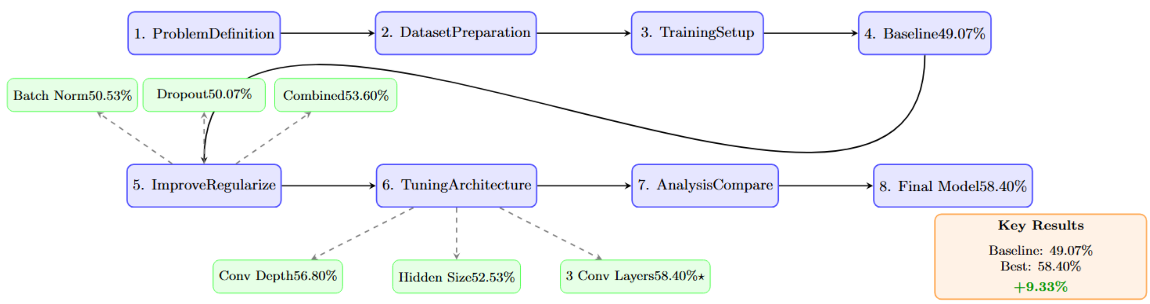
ARIN5101 Course Project: CNN-based Image Classification for Fine-Grained Mammal Recognition

Group Information

Item	Details
Group Number	20
Team Members	Huang Junqin, Rao Ridi, Li Shuying
Course	ARIN5101
Date	November 27th, 2025

CNN Image Classification Project Flow



1. Problem Definition and Objective

1.1 Problem Statement

This project addresses the challenge of **fine-grained image classification** - distinguishing between visually similar animal species using Convolutional Neural Networks (CNNs). Fine-grained classification is particularly challenging because categories share many visual characteristics while having subtle distinguishing features.

1.2 Objective

Build and systematically compare CNN models for mammal classification, exploring:

- Baseline CNN architecture
- Regularization techniques (Dropout, Batch Normalization)
- Architecture tuning (layer depth, width, number of layers)

2. Dataset Description

2.1 Data Source

We use a subset of the **CIFAR-100 dataset**, focusing on 15 mammal classes organized into three superclasses:

Superclass	Classes
Large Carnivores	bear, leopard, lion, tiger, wolf
Medium Mammals	fox, porcupine, opossum, raccoon, skunk
Small Mammals	hamster, mouse, rabbit, shrew, squirrel

2.2 Dataset Statistics

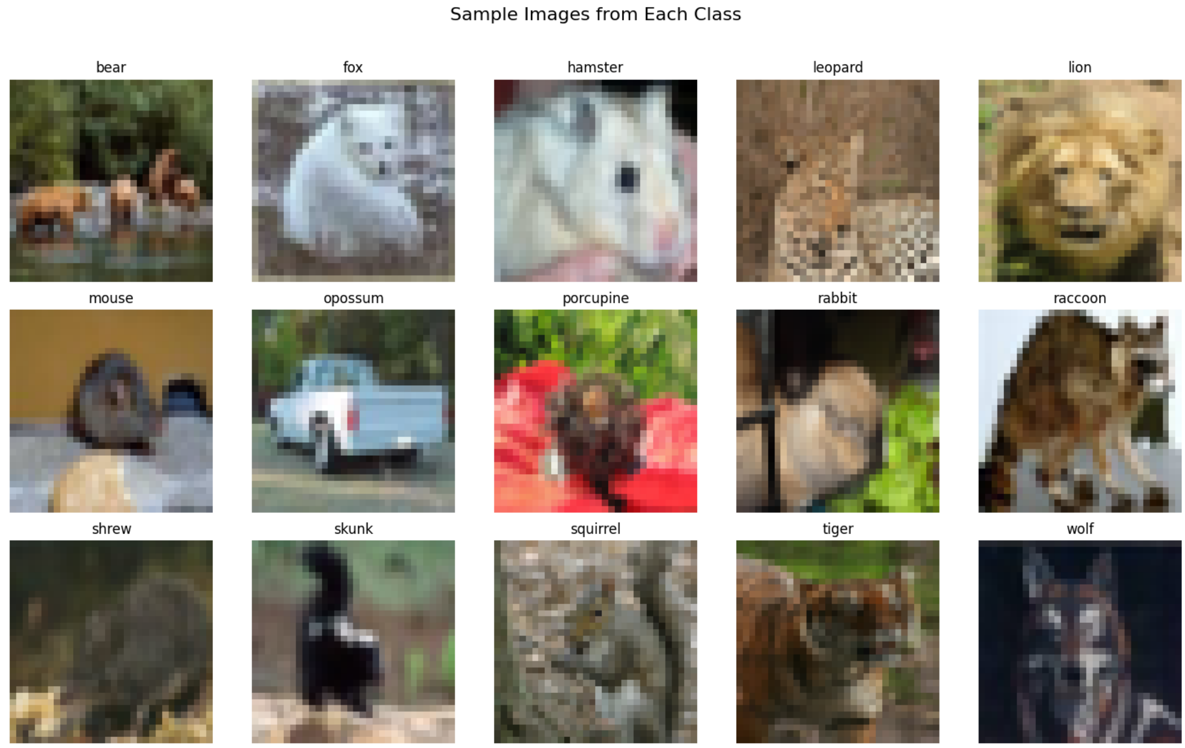
Metric	Value
Total Classes	15
Training Images	7,500 (500 per class)
Test Images	1,500 (100 per class)
Image Resolution	32 × 32 × 3 (RGB)
Color Channels	3 (Red, Green, Blue)

2.3 Data Distribution

The dataset is **balanced** across all classes, with approximately equal samples per category. This eliminates class imbalance as a confounding factor in model evaluation.

2.4 Sample Images

Sample images from each class demonstrate the challenge of fine-grained classification - many mammals share similar body shapes, fur textures, and color patterns.



2.5 Data Preprocessing

```
# Normalization: Scale pixel values to [0, 1]
X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0

# Convert to PyTorch format (NCHW)
X_tensor = torch.FloatTensor(X).permute(0, 3, 1, 2)
```

Preprocessing Steps:

1. Load CIFAR-100 dataset using `torchvision.datasets`
2. Filter to 15 mammal classes
3. Remap labels to 0-14
4. Normalize pixel values to [0, 1] range
5. Convert to PyTorch tensors (NCHW format)
6. Create DataLoader with batch size 64

3. Training Setup and Validation Strategy

3.1 Dataset Splitting Strategy

Split	Samples	Percentage
Training	6,000	80%
Validation	1,500	20%
Test	1,500	Held out

Stratification: We use stratified splitting to ensure each class is proportionally represented in both training and validation sets.

```
X_train_split, X_val, y_train_split, y_val = train_test_split(
    X_train, y_train,
    test_size=0.2,
    random_state=0,
    stratify=y_train  # Ensures balanced class distribution
)
```

3.2 Training Configuration

Parameter	Value
Optimizer	Adam
Loss Function	CrossEntropyLoss
Batch Size	64
Max Epochs	50
Early Stopping Patience	10 epochs
Learning Rate	Default (0.001)

3.3 Early Stopping Implementation

To prevent overfitting, we implement early stopping that:

- Monitors validation loss
- Saves best model weights
- Stops training if no improvement for 10 consecutive epochs
- Restores best weights after stopping

```
# Early stopping logic
if val_loss < best_val_loss:
    best_val_loss, wait = val_loss, 0
    best_weights = model.state_dict().copy()
else:
    wait += 1
    if wait >= patience:
        model.load_state_dict(best_weights)
        break
```

3.4 Metrics Tracked

- **Training Loss** - Cross-entropy loss on training batches
- **Training Accuracy** - Classification accuracy on training set
- **Validation Loss** - Cross-entropy loss on validation set
- **Validation Accuracy** - Classification accuracy on validation set
- **Test Accuracy** - Final evaluation metric

4. Baseline Model Architecture

4.1 Architecture Design

```
┌─────────────────────────────────────────────────────────┐
│                    INPUT (32×32×3)                      │
├─────────────────────────────────────────────────────────┤
│  Conv Layer 1: 32 filters, 3×3 kernel, padding=1        │
│  Activation: ReLU                                       │
│  MaxPooling: 2×2                                        │
│  Output: 16×16×32                                       │
├─────────────────────────────────────────────────────────┤
│  Conv Layer 2: 64 filters, 3×3 kernel, padding=1        │
│  Activation: ReLU                                       │
│  MaxPooling: 2×2                                        │
│  Output: 8×8×64                                         │
├─────────────────────────────────────────────────────────┤
│  Flatten: 8×8×64 = 4,096 features                       │
├─────────────────────────────────────────────────────────┤
│  Dense Layer: 128 units, ReLU                           │
├─────────────────────────────────────────────────────────┤
│  Output Layer: 15 units (one per class)                 │
└─────────────────────────────────────────────────────────┘
```

4.2 PyTorch Implementation

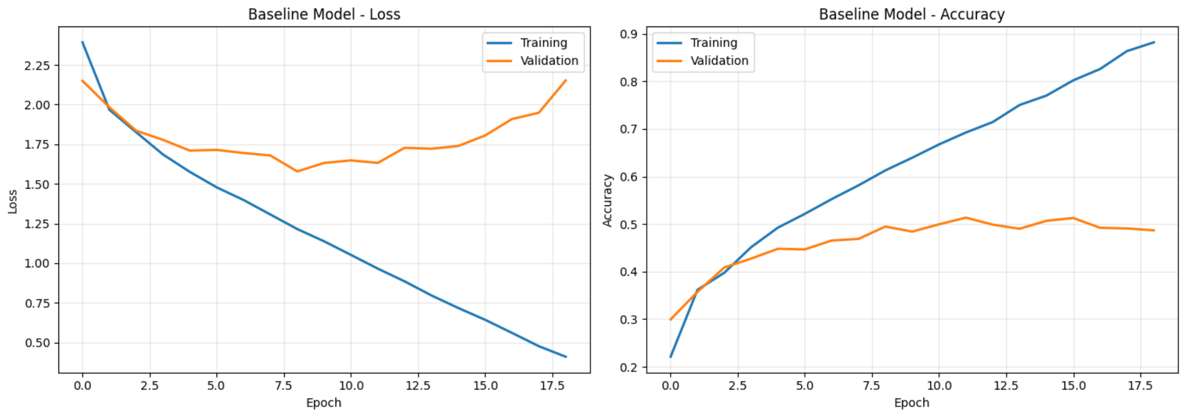
```
class BaselineCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
            nn.Conv2d(32, 64, 3, padding=1), nn.ReLU(), nn.MaxPool2d(2),
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(64 * 8 * 8, 128), nn.ReLU(),
            nn.Linear(128, 15)
        )

    def forward(self, x):
        return self.classifier(self.features(x))
```

4.3 Model Parameters

Component	Parameters
Conv Layer 1	3×3×3×32 + 32 = 896
Conv Layer 2	3×3×32×64 + 64 = 18,496
Dense Layer	4,096×128 + 128 = 524,416
Output Layer	128×15 + 15 = 1,935
Total	**545,743**

4.4 Baseline Performance

Metric	Value
Test Accuracy	**49.07%**
Test Loss	1.5958



Observations:

- Training accuracy reaches ~70% while validation plateaus at ~50%
- Clear signs of overfitting (gap between training and validation)
- Model learns basic features but struggles with fine-grained distinctions

5. Model Improvements

We implement two regularization techniques:

1. **Dropout** - Prevents overfitting by randomly dropping connections
2. **Batch Normalization** - Stabilizes training by normalizing layer inputs

5.1 Dropout Implementation

Dropout Rates:

- After each conv layer: 25%
- Before output layer: 50%

```
class DropoutCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, 3, padding=1), nn.ReLU(),
            nn.MaxPool2d(2), nn.Dropout2d(0.25),  # 25% dropout
            nn.Conv2d(32, 64, 3, padding=1), nn.ReLU(),
            nn.MaxPool2d(2), nn.Dropout2d(0.25),
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(64*8*8, 128), nn.ReLU(),
            nn.Dropout(0.5),  # 50% dropout
            nn.Linear(128, 15)
        )
```

Dropout Results:

Metric	Value
Test Accuracy	**50.07%**
Improvement	+1.00%

5.2 Batch Normalization Implementation

```
class BatchNormCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, 3, padding=1),
            nn.BatchNorm2d(32),  # Normalize after conv
            nn.ReLU(), nn.MaxPool2d(2),
            nn.Conv2d(32, 64, 3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(), nn.MaxPool2d(2),
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(64*8*8, 128),
            nn.BatchNorm1d(128),  # Normalize dense layer
            nn.ReLU(),
            nn.Linear(128, 15)
        )
```

Batch Normalization Results:

Metric	Value
Test Accuracy	**50.53%**
Improvement	+1.46%

5.3 Combined Improvements (Dropout + Batch Normalization)

```
class BothCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, 3, padding=1), nn.BatchNorm2d(32),
            nn.ReLU(), nn.MaxPool2d(2), nn.Dropout2d(0.25),
            nn.Conv2d(32, 64, 3, padding=1), nn.BatchNorm2d(64),
            nn.ReLU(), nn.MaxPool2d(2), nn.Dropout2d(0.25),
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(64*8*8, 128), nn.BatchNorm1d(128),
            nn.ReLU(), nn.Dropout(0.5),
            nn.Linear(128, 15)
```

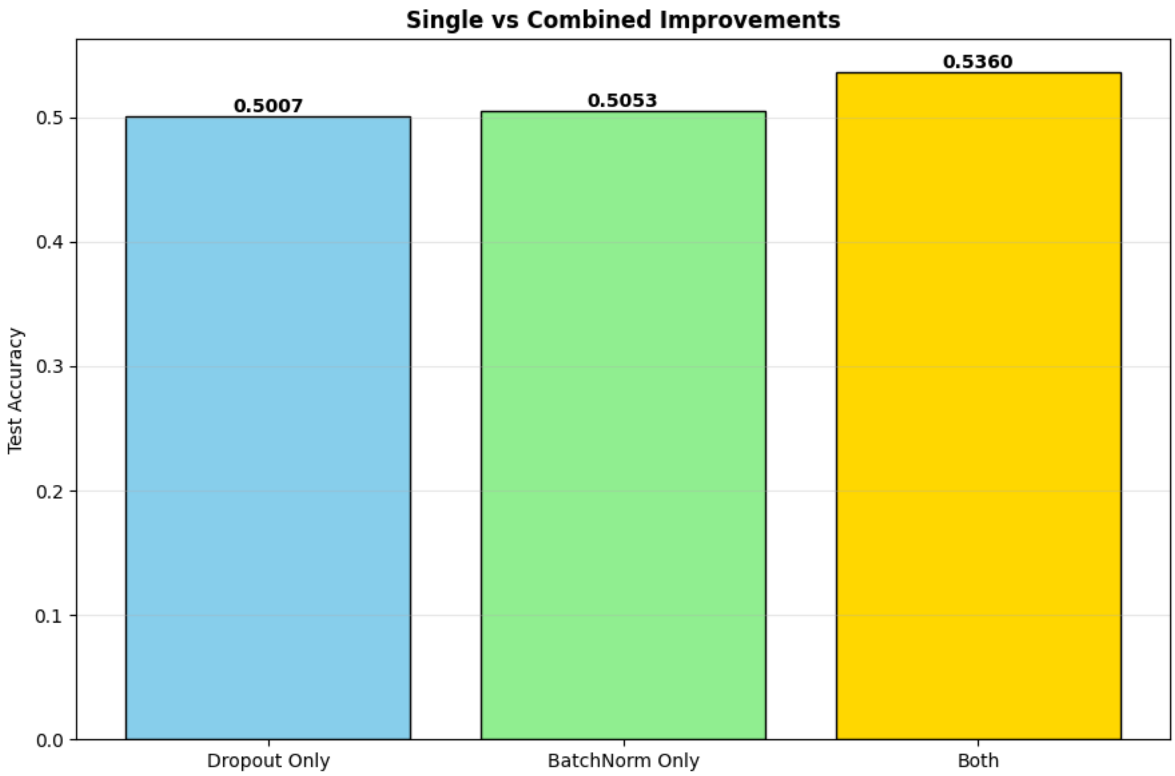
```
        )
```

Combined Results:

Metric	Value
Test Accuracy	**53.60%**
Improvement	+4.53%

5.4 Improvement Techniques Comparison

Model	Test Accuracy	Improvement
Baseline	49.07%	-
Dropout Only	50.07%	+1.00%
BatchNorm Only	50.53%	+1.46%
Both Combined	**53.60%**	**+4.53%**



Key Finding: Combining both techniques provides synergistic benefits - Dropout prevents co-adaptation of features while BatchNorm stabilizes the gradients during training.

6. Model Tuning Experiments

6.1 Convolution Layer Depth (64, 128)

Increasing filter counts from (32, 64) to (64, 128):

```
depth_model = TunedCNN(conv_depths=(64, 128))
```

Metric	Value
Test Accuracy	**56.80%**
Improvement	+7.73%

6.2 Hidden Layer Size (256 units)

Increasing hidden layer from 128 to 256 units:

```
hidden_model = TunedCNN(hidden_units=256)
```

Metric	Value
Test Accuracy	**52.53%**
Improvement	+3.46%

6.3 Three Convolutional Layers

Adding a third conv layer with 128 filters:

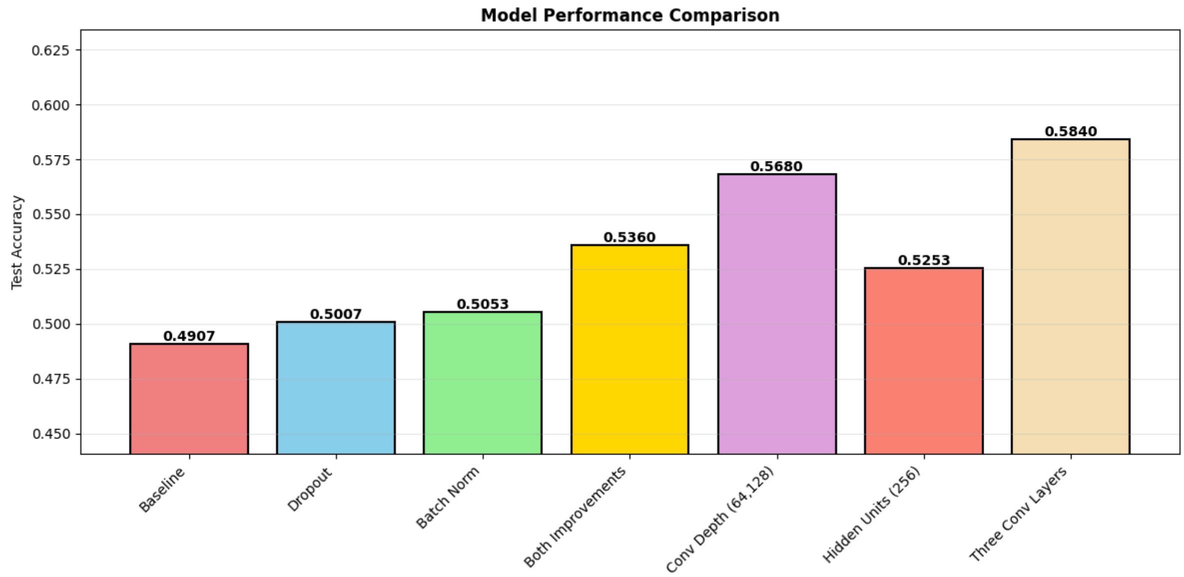
```
class TunedCNN(nn.Module):
    def __init__(self, conv_depths=(32, 64), hidden_units=128,
three_conv=False):
        super().__init__()
        layers_list = [
            nn.Conv2d(3, conv_depths[0], 3, padding=1),
nn.BatchNorm2d(conv_depths[0]),
            nn.ReLU(), nn.MaxPool2d(2), nn.Dropout2d(0.25),
            nn.Conv2d(conv_depths[0], conv_depths[1], 3, padding=1),
nn.BatchNorm2d(conv_depths[1]),
            nn.ReLU(), nn.MaxPool2d(2), nn.Dropout2d(0.25),
        ]

        if three_conv:
            layers_list += [
                nn.Conv2d(conv_depths[1], 128, 3, padding=1),
nn.BatchNorm2d(128),
                nn.ReLU(), nn.MaxPool2d(2), nn.Dropout2d(0.25)
            ]
```

Metric	Value
Test Accuracy	**58.40%**
Improvement	+9.33%

7. Performance Analysis

7.1 Complete Results Summary

Model	Test Accuracy	Test Loss	Rank
Baseline	49.07%	1.5958	7
Dropout	50.07%	1.5691	6
Batch Norm	50.53%	1.5829	5
Both Improvements	53.60%	1.4348	4
Hidden Units (256)	52.53%	1.4700	5
Conv Depth (64,128)	56.80%	1.3754	2
Three Conv Layers	**58.40%**	**1.2970**	**1**



7.2 Training Curves Analysis

Improvements Comparison:

- Baseline shows clear overfitting with diverging train/val curves
- Dropout reduces overfitting but slightly slows convergence
- BatchNorm accelerates early training but still overfits
- Combined approach shows best balance

Tuning Comparison:

- Deeper conv layers (64, 128) improve feature extraction
- Larger hidden layer (256) shows diminishing returns
- Three conv layers achieve best generalization

7.3 Loss Analysis

Model	Test Loss	Change from Baseline
Baseline	1.5958	-
Three Conv Layers	1.2970	-18.7%

Lower loss indicates more confident and accurate predictions.

8. Final Model Summary and Reflections

8.1 Best Performing Model

Three Convolutional Layers with Dropout and Batch Normalization

Metric	Value
Architecture	3 Conv + 1 Dense
Test Accuracy	**58.40%**
Improvement over Baseline	**+9.33%**
Test Loss	1.2970

8.2 Architecture Summary

```
┌─────────────────────────────────────────────────────────┐
│                    INPUT (32×32×3)                      │
├─────────────────────────────────────────────────────────┤
│  Conv1: 32 filters → BatchNorm → ReLU → MaxPool → Drop  │
│  Output: 16×16×32                                       │
├─────────────────────────────────────────────────────────┤
│  Conv2: 64 filters → BatchNorm → ReLU → MaxPool → Drop  │
│  Output: 8×8×64                                         │
├─────────────────────────────────────────────────────────┤
│  Conv3: 128 filters → BatchNorm → ReLU → MaxPool → Drop │
│  Output: 4×4×128                                        │
├─────────────────────────────────────────────────────────┤
│  Flatten: 4×4×128 = 2,048 features                      │
├─────────────────────────────────────────────────────────┤
│  Dense: 128 units → BatchNorm → ReLU → Dropout(0.5)     │
├─────────────────────────────────────────────────────────┤
│  Output: 15 classes                                     │
└─────────────────────────────────────────────────────────┘
```

8.3 Why Three Conv Layers Performs Best

1. **Hierarchical Feature Learning:**
    - Layer 1: Low-level features (edges, textures)
    - Layer 2: Mid-level features (patterns, shapes)
    - Layer 3: High-level features (body parts, distinctive features)
2. **Progressive Abstraction:**
    - Each pooling layer reduces spatial dimensions
    - Deeper layers capture more abstract, class-specific information
3. **Better Regularization:**
    - More opportunities for dropout
    - BatchNorm at each layer stabilizes training
4. **Appropriate Complexity:**
    - Sufficient capacity to learn fine-grained distinctions

- Not too deep to cause vanishing gradients on small images

8.4 Key Findings

Finding	Evidence
Regularization is crucial	Combined Dropout+BatchNorm outperforms single techniques
Depth matters more than width	3 conv layers > larger filters or hidden units
Early stopping prevents overfitting	Best models stopped before 50 epochs
Feature hierarchy is important	More layers = better abstract feature learning

8.5 Lessons Learned

1. **Start Simple:** Baseline model helps identify overfitting issues
2. **Combine Techniques:** Synergistic effects from Dropout + BatchNorm
3. **Depth Over Width:** Adding layers more effective than wider layers
4. **Monitor Validation:** Early stopping crucial for generalization
5. **Systematic Comparison:** Testing one variable at a time isolates effects

8.6 Limitations and Future Work

Limitations:

- 58.40% accuracy still leaves room for improvement
- Small image size (32×32) limits feature detail
- Simple augmentation not explored

Future Improvements:

- Data augmentation (rotation, flipping, color jittering)
- Transfer learning with pre-trained models (ResNet, VGG)
- Learning rate scheduling
- More sophisticated architectures (skip connections, attention)

9. Confusion Matrix & Classification Report

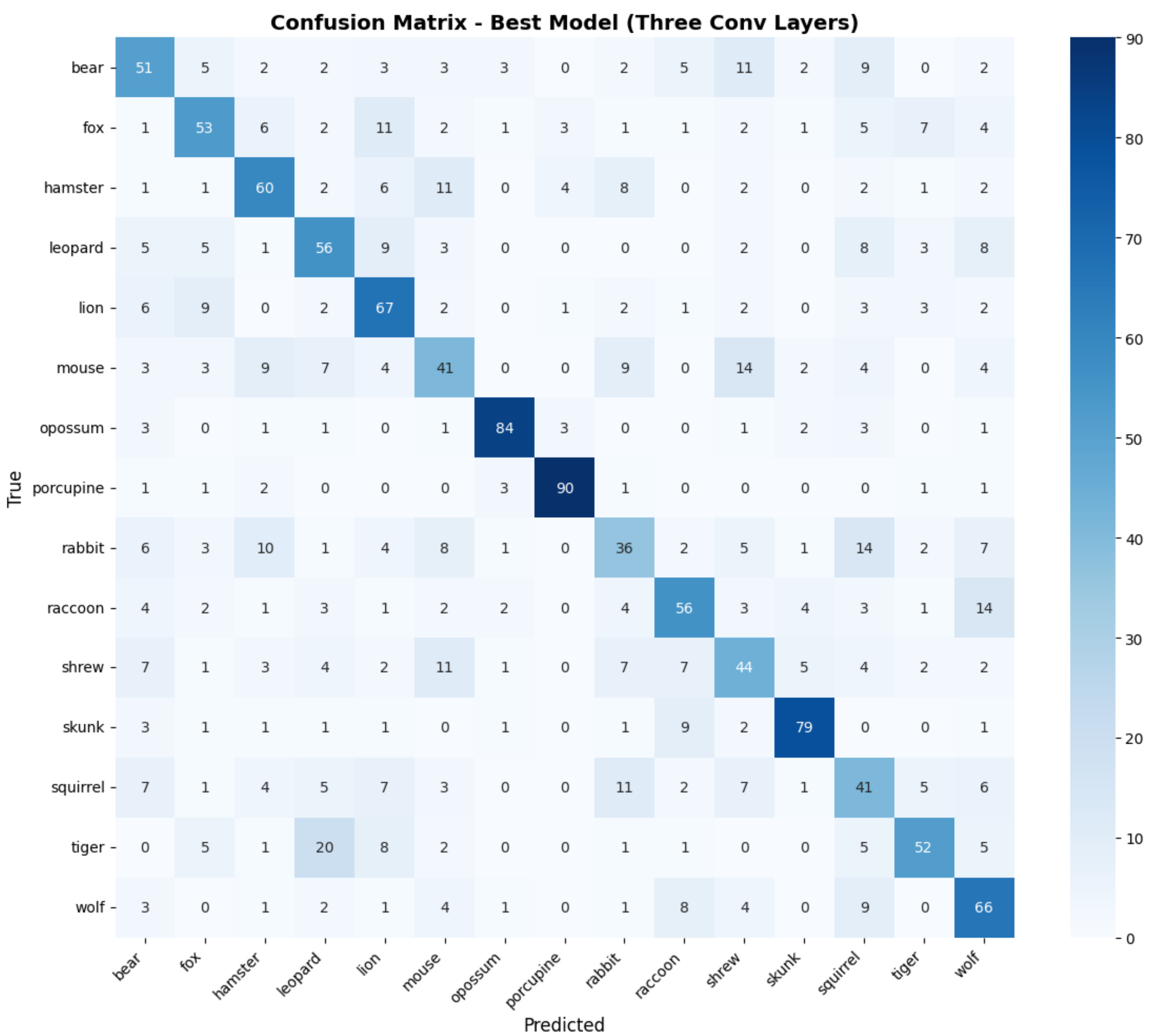
To gain deeper insights into model performance across individual classes, we analyze the confusion matrix and per-class metrics of our best performing model (Three Conv Layers).

9.1 Implementation

```
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns

# Evaluate best model on test set
best_model.eval()
all_preds, all_labels = [], []

with torch.no_grad():
    for X, y in test_loader:
```

```
        X = X.to(device)
        outputs = best_model(X)
        preds = outputs.argmax(dim=1).cpu().numpy()
        all_preds.extend(preds)
        all_labels.extend(y.numpy())

# Generate classification report
print(classification_report(all_labels, all_preds, target_names=class_names))

# Plot confusion matrix
cm = confusion_matrix(all_labels, all_preds)
plt.figure(figsize=(12, 10))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=class_names, yticklabels=class_names)
plt.title('Confusion Matrix - Best Model (Three Conv Layers)')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.tight_layout()
plt.show()
```

9.2 Confusion Matrix Visualization



The confusion matrix shows prediction patterns across all 15 mammal classes. Diagonal values represent correct predictions, while off-diagonal values indicate misclassifications.

9.3 Per-Class Performance Analysis

Class	Precision	Recall	F1-Score	Support
bear	0.50	0.51	0.51	100
fox	0.59	0.53	0.56	100
hamster	0.59	0.60	0.59	100
leopard	0.52	0.56	0.54	100
lion	0.54	0.67	0.60	100
mouse	0.44	0.41	0.42	100
opossum	0.87	0.84	0.85	100
porcupine	0.89	0.90	0.90	100
rabbit	0.43	0.36	0.39	100
raccoon	0.61	0.56	0.58	100
shrew	0.44	0.44	0.44	100
skunk	0.81	0.79	0.80	100
squirrel	0.37	0.41	0.39	100
tiger	0.68	0.52	0.59	100
wolf	0.53	0.66	0.59	100
Macro Avg	**0.59**	**0.58**	**0.58**	**1500**

9.4 Key Observations from Confusion Matrix

Best Classified Classes:

- **Porcupine** (90% recall, 89% precision) - Distinctive spiny appearance
- **Opossum** (84% recall, 87% precision) - Unique body shape and facial features
- **Skunk** (79% recall, 81% precision) - Distinctive black and white coloring

Most Confused Classes:

- **Rabbit** (36% recall) - Often confused with other small mammals
- **Mouse** (41% recall) - Very similar to shrew and hamster
- **Squirrel** (41% recall) - Confused with similar-sized rodents

Analysis:

1. Animals with distinctive visual patterns (porcupine, skunk, opossum) achieve highest accuracy
2. Small mammals (mouse, shrew, rabbit, squirrel) show significant confusion due to similar size and shape
3. Large carnivores (leopard, lion, tiger) show moderate confusion due to similar feline body structure

4. The model successfully learns species-specific features but struggles with subtle inter-class differences within similar animal groups

10. Conclusion

This project successfully demonstrated the systematic development and optimization of CNN models for fine-grained image classification. Starting from a baseline model achieving 49.07% accuracy, we progressively improved performance through:

1. **Regularization techniques** (Dropout + Batch Normalization) → 53.60%
2. **Architecture tuning** (increased depth and filters) → 56.80%
3. **Adding depth** (three convolutional layers) → **58.40%**

The best model achieved a **9.33% improvement** over the baseline, demonstrating that careful architecture design and proper regularization are essential for training effective CNNs on relatively small datasets.

The confusion matrix analysis revealed that the model performs best on visually distinctive animals (porcupine: 90%, opossum: 84%, skunk: 79%) while struggling with similar-looking species within the same family groups.

Appendix: Code Repository Structure

```
HKUST_Python/
├── Python_project.ipynb    # Main Jupyter notebook with all code
├── report.md               # This report
└── data/                   # CIFAR-100 dataset
```