DATA MINING AND MACHINE LEARNING II - FINAL PROJECT

Multiple Image Classification

MEMBERS:

NGO XUAN KIEN - 23BI14239
TRAN GIA KHANH - 23BI14218
NGO MINH PHUOC - 23BI14361
NGUYEN THE KHAI - 23BI14205
NGUYEN GIA NAM - 23BI14328
NGUYEN DUC THANH - 23BI14406

June 2025

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

This report presents a detailed analysis of Convolutional Neural Network (CNN) applications for image classification on two distinct datasets: CIFAR-100 and COIL-20.

For CIFAR-100, the study explores a custom CNN architecture, its data augmentation strategies, and compares its performance against established models like EfficientNet-B0 and ResNet-18.

For COIL-20, various CNN implementations are evaluated, highlighting the impact of different training methodologies, data preprocessing techniques, and framework choices (TensorFlow/Keras vs. PyTorch).

Key findings include the critical role of data augmentation for small, complex datasets like CIFAR-100, the effectiveness of advanced architectures in achieving higher accuracies, and the relative simplicity of the COIL-20 dataset, which allows for near-perfect classification across diverse CNN setups. The report synthesizes these observations to provide insights into model design, training optimization, and generalization capabilities in image classification tasks.

* *Data Science, University of Science and Technology, HaNoi, VietNam*

1 INTRODUCTION

Image classification, a core task in computer vision, involves sorting images into predetermined categories. Convolutional Neural Networks (CNNs) have become the popular method for this, thanks to their ability to automatically learn intricate patterns from raw image data.

This report explores how CNNs perform on two distinct datasets: CIFAR-100 and COIL-20. CIFAR-100 is a widely used benchmark with 60,000 small (32x32 pixel) color images spanning 100 classes, making it challenging due to its numerous, fine-grained categories. COIL-20, on the other hand, comprises 1,440 grayscale images of 20 objects, each captured from 72 different angles. While also relatively small (initially 128x128 pixels), its systematic variations present a different set of challenges and opportunities for evaluating models.

The aim of this report is to provide a professional, concise, and insightful analysis of the methodologies, models, and comparative performance on these two datasets, offering valuable information for those interested in image classification and deep learning.

1.1 COIL-20

The COIL-20 dataset (Columbia Object Image Library) consists of 1,440 grayscale images of 20 distinct objects, with each object captured at 72 different poses (5-degree intervals). All images are initially 128x128 pixels. While also relatively small, its controlled nature and systematic variations present a different set of challenges and opportunities for model evaluation compared to CIFAR-100.

1.1.1 Attribute Summary

(a) Example images of twenty categories

(b) Part example images of one category

Figure 1: Feature Labels - COIL-20 Dataset.

The dataset itself does not provide the actual label, just classify it as obj[i] which has i in range 1 - 20.

1.2 CIFAR-100

The CIFAR-100 dataset consists of 60000 32x32 color images in 100 classes, with 600 images per class. The 100 classes in the CIFAR-100 are grouped into 20 superclasses. Each image comes with a "fine" label (the class to which it belongs) and a "coarse" label (the superclass to which it belongs). There are 50000 training images and

10000 test images. The meta file contains the label names of each class and superclass.

1.2.1 Attribute Summary

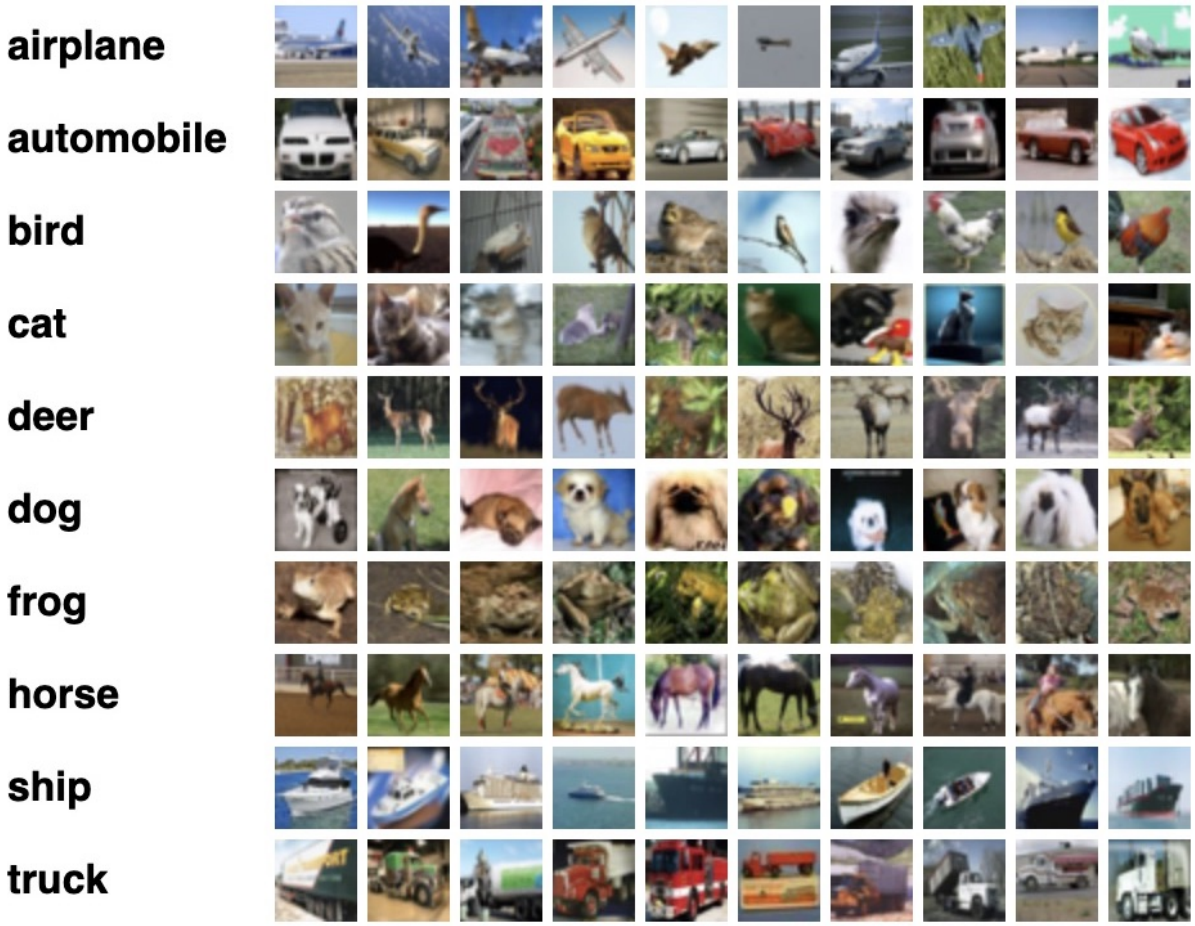


Figure 2: Feature sample labels - CIFAR-100 Dataset.

The meta file contains the label names of each class and superclass.

Classes:

- 1-5) beaver, dolphin, otter, seal, whale
- 6-10) aquarium fish, flatfish, ray, shark, trout
- 11-15) orchids, poppies, roses, sunflowers, tulips
- 16-20) bottles, bowls, cans, cups, plates
- 21-25) apples, mushrooms, oranges, pears, sweet peppers
- 26-30) clock, computer keyboard, lamp, telephone, television
- 31-35) bed, chair, couch, table, wardrobe
- 36-40) bee, beetle, butterfly, caterpillar, cockroach
- 41-45) bear, leopard, lion, tiger, wolf
- 46-50) bridge, castle, house, road, skyscraper

- 51-55) cloud, forest, mountain, plain, sea
- 56-60) camel, cattle, chimpanzee, elephant, kangaroo
- 61-65) fox, porcupine, possum, raccoon, skunk
- 66-70) crab, lobster, snail, spider, worm
- 71-75) baby, boy, girl, man, woman
- 76-80) crocodile, dinosaur, lizard, snake, turtle
- 81-85) hamster, mouse, rabbit, shrew, squirrel
- 86-90) maple, oak, palm, pine, willow
- 91-95) bicycle, bus, motorcycle, pickup truck, train
- 96-100) lawn mower, rocket, streetcar, tank, tractor

Superclasses:

- aquatic mammals (classes 1-5)
- fish (classes 6-10)
- flowers (grades 11-15)
- food containers (classes 16-20)
- fruit and vegetables (classes 21-25)
- household electrical devices (classes 26-30)
- household furniture (classes 31-35)
- insects (classes 36-40)
- large carnivores (classes 41-45)
- large man-made outdoor things (classes 46-50)
- large natural outdoor scenes (classes 51-55)
- large omnivores and herbivores (classes 56-60)
- medium-sized mammals (classes 61-65)
- non-insect invertebrates (classes 66-70)
- people (classes 71-75)
- reptiles (classes 76-80)
- small mammals (classes 81-85)
- trees (classes 86-90)
- vehicles 1 (classes 91-95)
- vehicles 2 (classes 96-100)

2 PREPROCESSING DATA

2.1 COIL-20

The COIL-20 dataset consists of 1,440 grayscale images of 20 distinct objects, with each object represented by 72 images captured at 5-degree intervals.

All images are initially 128x128 pixels. Images are loaded, and each is associated with a label corresponding to its object category (e.g., 'obj1', 'obj2'), derived from folder names, which enables supervised learning.

The image preprocessing steps are crucial for preparing the data for CNN training:

- **Grayscale Conversion:** Images are converted to grayscale (1 channel, "L" mode in PIL) from their original format. This decision significantly reduces model complexity, computation, and memory usage, as color information is not essential for distinguishing objects in COIL-20.
- **Resizing:** All images are resized to a consistent 64x64 pixels. This ensures uniform input dimensions for CNNs, which is critical for stable training, and also contributes to faster training and testing.
- **Normalization:** Pixel values, initially ranging from 0 to 255, are normalized by dividing by 255.0. This normalization aids the model in converging faster and more stably. The pixel data is also converted to a 32-bit floating-point format for efficient computation.

For label handling, string labels (e.g., "obj1") are converted into numerical integers using LabelEncoder (e.g., 'obj1' $\rightarrow$ 0, 'obj2' $\rightarrow$ 1).

These integer labels are then transformed into one-hot encoded vectors (e.g., if num_classes = 20, label 5 $\rightarrow$ [0,0,0,0,0,1,0,...,0]) using to categorical. This format is required for training with categorical_crossentropy loss and matches the softmax output of classification models.

Finally, the preprocessed data is split into an 80% training set and a 20% testing set. stratify = y_encoded is applied during this split to ensure that the class distribution is proportionally maintained in both training and testing subsets, preventing class imbalance issues.

The decision to convert COIL-20 images to grayscale and resize them to 64x64 pixels significantly simplifies the classification task, contributing to the high performance observed.

COIL-20 objects are simple, and color information is not essential for their distinction. Converting to grayscale reduces the input channel count from 3 (RGB) to 1, directly reducing the number of parameters in the first convolutional layer and subsequent layers, thereby lowering model complexity and computational overhead.

Additionally, resizing from 128x128 to 64x64 pixels reduces the total number of pixels by a factor of four. This further diminishes the input data volume, leading to smaller feature maps, fewer parameters in dense layers, and faster training and inference.

These preprocessing choices are highly effective for a controlled dataset like COIL-20, where the primary distinguishing features are shape and pose rather than color or fine texture. This demonstrates that optimizing preprocessing for dataset characteristics can be as impactful as architectural choices in achieving high performance and efficiency, especially for less complex classification problems.

2.2 CIFAR-100

The first step involved data cleaning, where we checked the dataset for any corrupted or invalid samples. Specifically, we verified that no images were null, empty, or had an incorrect shape differing from the expected (32, 32, 3) format.

Upon inspection, we found no such issues, confirming that CIFAR-100 is already well-curated and pre-cleaned. Nonetheless, this validation step was included as a precaution to ensure data integrity before proceeding with further preprocessing and model training.

The CIFAR-100 dataset, characterized by its small sample size (only 600 images for each class) and low resolution (32x32 pixels), presents a significant risk of overfitting for CNNs. To mitigate this and enhance model generalization, a series of data augmentation techniques were applied. These methods

increase dataset diversity without the need for collecting new data, helping the model learn semantic content rather than exact pixel patterns.

Key augmentation techniques employed include:

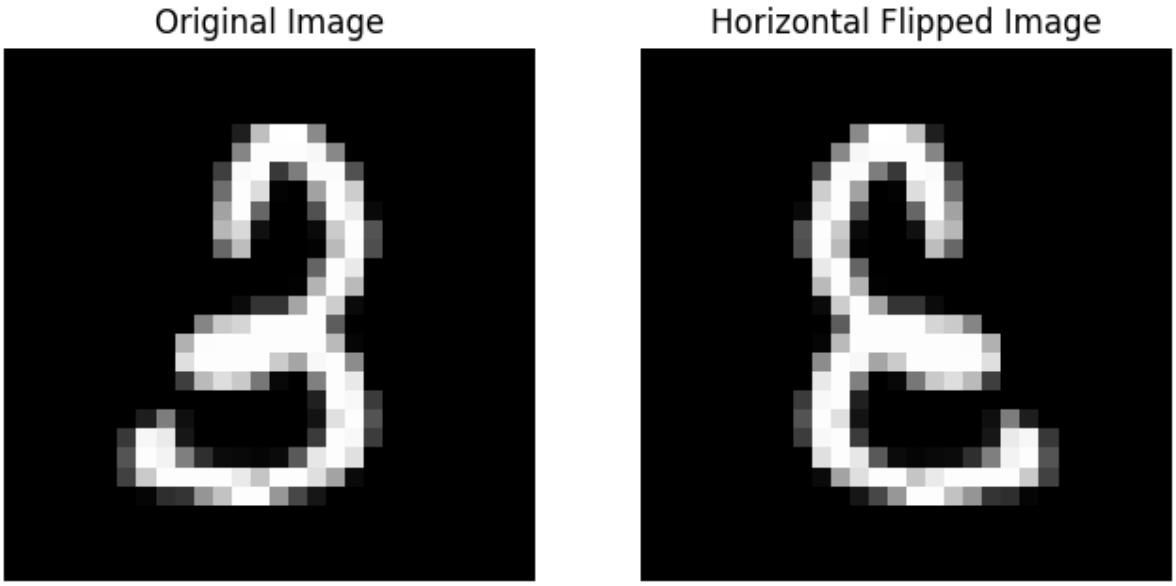


Figure 3: Horizontal Flip

1. **RandomHorizontalFlip()**: This operation randomly flips the image left-right during training with a 50% probability for each image. This mimics real-world symmetry, such as a dog facing left versus right, and aids the model in learning features that are invariant to pose.

Figure 4: Random Crop

2. **RandomCrop(32, padding=4)**: The input image is padded by 4 pixels on each side, expanding it from 32x32 to 40x40 pixels. A 32x32 patch is then randomly cropped from this padded image. This technique prevents the model from overfitting to specific pixel positions by introducing slight variations in object placement.

(a) Original

(b) Rotate15

Figure 5: Random Rotation

3. **RandomRotation(15)**: Images are randomly rotated by an angle ranging from -15° to +15°. This improves rotational invariance, which is crucial because real-world objects do not always appear perfectly upright. Small rotation angles were chosen to avoid radically altering the object's identity.

Figure 6: Color Jitter

4. **ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.)**: This augmentation adjusts various color properties of the image. It enhances color invariance by simulating different lighting or camera conditions, thereby helping the model generalize more effectively to diverse real-world images.

Following augmentation, image pixel values, which are initially in the range (after conversion to tensor format), were normalized to [-1, 1] for each RGB channel using Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)). Centering pixel values around zero and bounding them within [-1, 1] facilitates better gradient flow, more stable training, and faster convergence of the neural network.

The original training dataset was logically partitioned into an 80% training set (40,000 images) and a 20% validation set (10,000 images). Data loaders were configured to process images in batches of 64.

Training data was shuffled to prevent order-based learning, which can otherwise lead to biased learning, poor generalization, and slower convergence. The use of multiple CPU cores (num_workers=2) was implemented to accelerate data loading.

The extensive data augmentation applied to CIFAR-100 is not merely a common practice but a critical necessity for achieving competitive performance on this dataset. CIFAR-100's small sample size per class and low resolution inherently increase the risk of overfitting, where a model memorizes training examples rather than learning generalizable features. Data augmentation directly addresses this by artificially expanding the dataset and exposing the model to a wider range of plausible inputs. This forces the model to learn robust, semantic features that are invariant to minor variations in pose, position, orientation, and lighting, rather than relying on exact pixel patterns. This highlights a fundamental principle in deep learning: for datasets with inherent limitations in size or variability, intelligent data augmentation is often as crucial as, if not more important than, architectural complexity in driving generalization and mitigating overfitting. It serves as a cost-effective method to improve model robustness without the need for collecting new data.

3 MODEL: CUSTOMCNN

3.1 COIL-20

3.1.1 CustomCNN

Across the various COIL-20 implementations, a common CNN architecture was adopted, comprising two convolutional blocks followed by fully connected layers for classification. This architecture is considered common and effective, particularly for small and simple datasets like COIL-20, which consists of grayscale images of small size.

The architecture components are as follows:

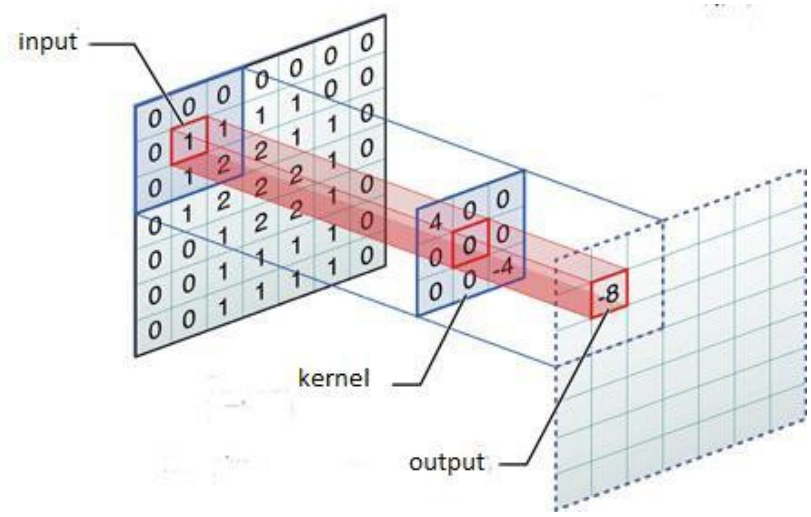


Figure 7: Conv2D

1. **Conv2D(filters, (3, 3), activation='relu')**: This layer applies convolutional filters (e.g., 32, then 64) with a 3x3 kernel. Convolutional layers are the core of CNNs, designed to extract spatial features such as edges, textures, and other important visual patterns from the input images. ReLU activation introduces non-linearity, which is essential for learning complex patterns.

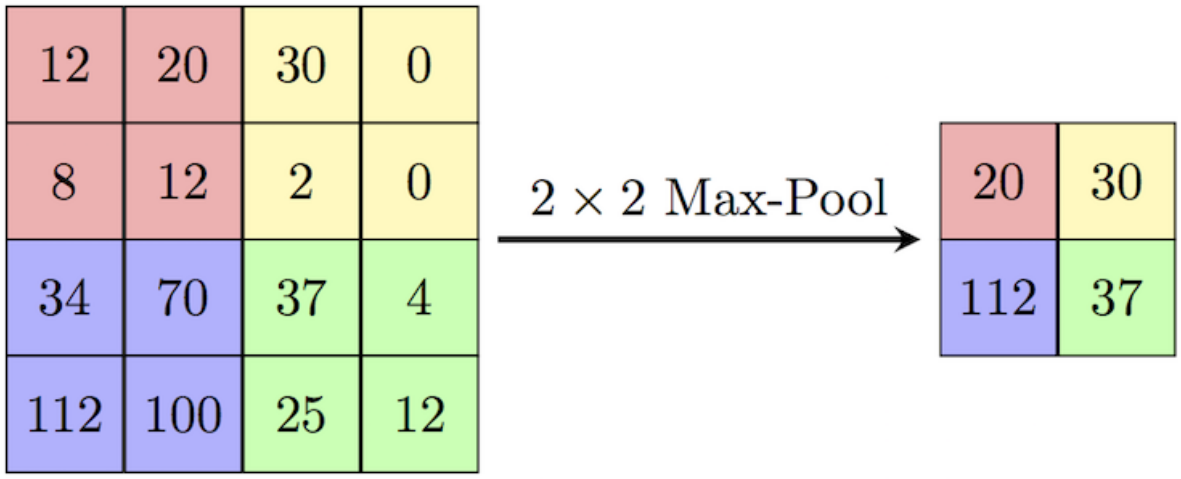


Figure 8: MaxPooling2D

2. **MaxPooling2D((2, 2))**: This layer reduces the spatial dimensions of the feature maps by taking the maximum value within each 2x2 block. This downsampling serves multiple purposes: it reduces the number of parameters, helps prevent overfitting by making features more robust to small translations (translation invariance), and speeds up computation. For example, a 64x64 feature map would be reduced to 32x32.

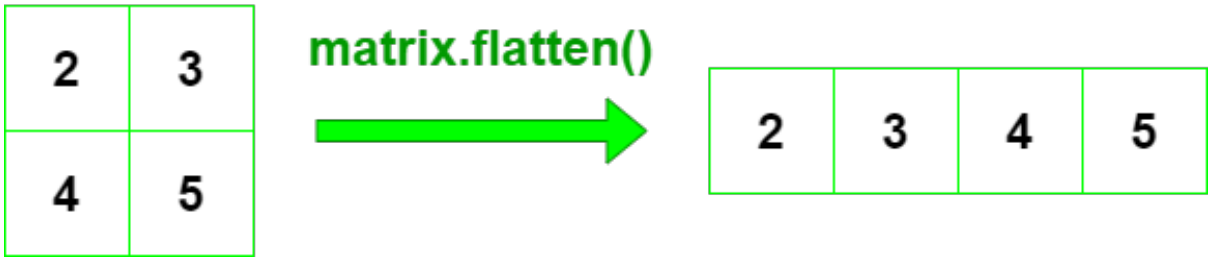


Figure 9: Flatten

3. **Flatten()**: This layer converts the 2D feature maps from the convolutional layers into a 1D vector. This is a necessary step to feed the data into the subsequent fully connected (dense) layers.

4. **Dense(128, activation='relu')**: A fully connected layer with 128 neurons and ReLU activation. This layer learns to combine the high-level features extracted by the convolutional layers to make more abstract representations relevant for classification.

5. **Dropout(0.5)**: This layer randomly deactivates 50% of the neurons during training. This regularization technique helps prevent overfitting by forcing the network to learn more robust features and not rely too heavily on any single neuron or feature combination.

6. **Dense(num_classes, activation='softmax')**: This is the final output layer, with one neuron per class (20 for COIL-20). The softmax activation function converts the raw output scores into probabilities for each class, ensuring they sum to 1. The class with the highest probability is the model's prediction, making it suitable for multi-class classification.

The consistent use of a relatively simple CNN architecture (two Conv2D + MaxPooling2D blocks followed by Dense layers) across all COIL-20 versions, despite framework differences, indicates that this architecture is robust and sufficiently powerful for this specific dataset.

This architectural pattern is explicitly stated as common and effective for small and simple datasets like COIL-20.

The combination of convolutional layers for feature extraction, pooling for dimensionality reduction and invariance, and dense layers for classification is a proven approach.

The fact that this relatively straightforward architecture consistently achieves high performance (100% accuracy in many cases) across different implementations indicates that the COIL-20 dataset is not architecturally demanding.

This implies that for datasets with clear, distinct features and limited variability, complex, deeper architectures (like those needed for CIFAR-100) might be overkill, leading to unnecessary computational expense without significant performance gains.

The following table summarizes the common CNN model architecture used for COIL-20, including output shapes and parameter counts:

Table 1: Layer-wise Architecture of the CNN Model

Layer (Type)	Output Shape	Param #
Conv2D (conv2d_4)	(None, 62, 62, 32)	320
MaxPooling2D (max_pooling2d_4)	(None, 31, 31, 32)	0
Conv2D (conv2d_5)	(None, 29, 29, 64)	18,496
MaxPooling2D (max_pooling2d_5)	(None, 14, 14, 64)	0
Flatten (flatten_2)	(None, 12544)	0
Dense (dense_4)	(None, 128)	1,605,760
Dropout (dropout_2)	(None, 128)	0
Dense (dense_5)	(None, 20)	2,580

3.2 CIFAR-100

3.2.1 CustomCNN

The custom CNN model, implemented using PyTorch's torch.nn.Module, employs a block-based architecture. This design strategy promotes modularity, clarity, and hierarchical feature learning, mirroring the way visual features are processed in a hierarchical manner. Early blocks capture low-level patterns such as edges and textures, while deeper blocks learn more complex structures like object parts or entire objects.

The model's architecture consists of four sequential feature extraction blocks (self.features), which progressively increase the number of filters while reducing spatial dimensions, followed by a classifier (self.classifier).

Each block typically comprises a sequence of layers:

1. **Convolutional Layers (nn.Conv2d):** These layers apply filters to extract spatial features. The first layer takes 3 input channels (RGB) and produces 64 feature maps. Subsequent convolutional layers take the output channels of the previous layer as input (e.g., $64 \rightarrow 128$, $128 \rightarrow 256$, $256 \rightarrow 512$). A 3x3 kernel size with padding=1 is consistently used, ensuring the output size remains the same before pooling.

2. **Batch Normalization (nn.BatchNorm2d):** Applied to the output channels of the convolutional layer, batch normalization standardizes activations across a mini-batch. This significantly improves training stability, accelerates convergence, and reduces internal covariate shift.

3. **ReLU Activation (nn.ReLU):** The Rectified Linear Unit introduces non-linearity, which is essential because, without it, stacking multiple linear operations (like convolutions) would still result in a linear model. This allows the network to learn complex, non-linear relationships in the data.

4. **Max Pooling (nn.MaxPool2d(2)):** Reduces the spatial dimensions (height and width) by half after each block. For example, a 32x32 input becomes 16x16 after the first block's pooling, then 8x8, 4x4, and finally 2x2 after the fourth block. This downsampling reduces computational cost, helps control overfitting, and provides a degree of translation invariance.

The convolution-batch normalization-ReLU sequence is repeated twice within each block, allowing the network to learn more complex features. Early layers (like Block 1) detect simple patterns such as edges or color blobs, while deeper layers (like Block 2) build on that to detect more abstract patterns like corners, textures, or object parts. Increasing the number of filters in deeper blocks provides the necessary capacity to represent these increasingly complex visual features.

The classifier layers (self.classifier) are responsible for mapping the extracted features to the final class predictions:

1. **nn.AdaptiveAvgPool2d((1, 1)):** Global Average Pooling reduces the final feature map (e.g., 512x2x2) to a 1x1 spatial dimension (e.g., 512x1x1). This ensures a fixed-size output regardless of input resolution, replaces flattening all pixels with averaging them, and helps reduce overfitting and parameter count.

2. **nn.Flatten():** Converts the feature map into a 1D vector, preparing it for the fully connected layers.

3. **nn.Linear(512, 512) and nn.ReLU():** A fully connected hidden layer with ReLU activation, adding non-linearity and allowing the learning of complex combinations of features.

4. **nn.Dropout(0.5):** Randomly zeroes out 50% of the features during training. This helps reduce overfitting by encouraging the model not to rely too heavily on any one feature.

5. **nn.Linear(512, num_classes):** The final fully connected layer maps the learned features to the final prediction scores (logits) for the 100 classes in CIFAR-100.

The architectural design of the custom CNN for CIFAR-100, particularly the increasing filter counts in deeper blocks and the use of Global Average Pooling, reflects a deliberate strategy to balance feature complexity with overfitting prevention.

Increasing filters in deeper layers provides the necessary capacity for the network to represent the growing complexity of abstract features, such as combinations of edges, textures, and object parts. This is a standard practice in CNNs to capture richer representations as information flows deeper into the network. Global Average Pooling reduces the number of parameters in the subsequent fully connected layers compared to traditional flattening, which directly reduces the model's capacity for memorization and thus helps prevent overfitting.

It also makes the model more robust to input size variations. This combined strategy demonstrates a sophisticated understanding of CNN design principles: providing sufficient representational power where needed while simultaneously employing regularization techniques (Global Average Pooling, Dropout) to manage model complexity and enhance generalization, which is especially crucial for a dataset like CIFAR-100 that is prone to overfitting.

The following table summarizes the custom CNN architecture for CIFAR-100:

Table 2: Custom CNN Architecture for CIFAR-100 Classification

Block	Layer Type	Input Shape (C, H, W)	Output Shape (C, H, W)	Description
Input	RGB Image	–	(3, 32, 32)	CIFAR-100 input images with 3 color channels
Block 1 *Low-level Features*	Conv2D + BN + ReLU	(3, 32, 32)	(64, 32, 32)	3→64 filters, kernel=3×3, padding=1. Extract edges and basic patterns
	Conv2D + BN + ReLU	(64, 32, 32)	(64, 32, 32)	64→64 filters, kernel=3×3, padding=1. Refine feature representations
	MaxPool2D	(64, 32, 32)	(64, 16, 16)	2×2 pooling, stride=2. Reduce spatial dimensions by half
Block 2 *Mid-level Patterns*	Conv2D + BN + ReLU	(64, 16, 16)	(128, 16, 16)	64→128 filters, kernel=3×3, padding=1. Detect shapes and contours
	Conv2D + BN + ReLU	(128, 16, 16)	(128, 16, 16)	128→128 filters, kernel=3×3, padding=1. Enhance pattern recognition
	MaxPool2D	(128, 16, 16)	(128, 8, 8)	2×2 pooling, stride=2. Spatial downsampling
Block 3 *Abstract Features*	Conv2D + BN + ReLU	(128, 8, 8)	(256, 8, 8)	128→256 filters, kernel=3×3, padding=1. Learn complex textures
	Conv2D + BN + ReLU	(256, 8, 8)	(256, 8, 8)	256→256 filters, kernel=3×3, padding=1. Consolidate abstract patterns
	MaxPool2D	(256, 8, 8)	(256, 4, 4)	2×2 pooling, stride=2. Further dimension reduction
Block 4 *High-level Semantics*	Conv2D + BN + ReLU	(256, 4, 4)	(512, 4, 4)	256→512 filters, kernel=3×3, padding=1. Extract class-discriminative features
	Conv2D + BN + ReLU	(512, 4, 4)	(512, 4, 4)	512→512 filters, kernel=3×3, padding=1. Refine semantic representations
	MaxPool2D	(512, 4, 4)	(512, 2, 2)	2×2 pooling, stride=2. Final spatial reduction
Classifier *Decision Network*	AdaptiveAvgPool2D	(512, 2, 2)	(512, 1, 1)	Global average pooling to create feature vector
	Flatten	(512, 1, 1)	(512)	Convert 3D tensor to 1D feature vector
	Linear + ReLU	(512)	(512)	Fully connected layer for feature combination
	Dropout	(512)	(512)	Regularization with 50% dropout rate
	Linear	(512)	(100)	Final classification layer for 100 CIFAR-100 classes

Architecture Summary	
Total Parameters	5 million trainable parameters
Key Techniques	Batch Normalization for training stability, Progressive channel expansion (3→64→128→256→512), Dropout regularization, Global Average Pooling
Receptive Field	Covers entire 32×32 input through hierarchical feature extraction

Algorithm 1 Adam Optimizer (from ICLR 2015 Paper)

Description: According to the published conference paper at **ICLR 2015**, *Adam* is our proposed algorithm for stochastic optimization. g_t^2 indicates the elementwise square $g_t \odot g_t$.
Default hyperparameters:

- $\alpha = 0.001$
- $\beta_1 = 0.9$
- $\beta_2 = 0.999$
- $\epsilon = 10^{-8}$

All operations on vectors are element-wise. With β_1^t and β_2^t, we denote exponentiation by t.
Require:

- α: Stepsize
- $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates
- $f(\theta)$: Stochastic objective function
- θ_0: Initial parameter vector

$m_0 \leftarrow 0$ ▷ Initialize 1st moment vector
$v_0 \leftarrow 0$ ▷ Initialize 2nd moment vector
$t \leftarrow 0$ ▷ Initialize timestep
while not converged **do**
  $t \leftarrow t + 1$
  $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ ▷ Gradient at timestep t
  $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$ ▷ Update biased first moment estimate
  $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$ ▷ Update biased second raw moment estimate
  $\hat{m}_t \leftarrow \frac{m_t}{1 - \beta_1^t}$ ▷ Compute bias-corrected first moment estimate
  $\hat{v}_t \leftarrow \frac{v_t}{1 - \beta_2^t}$ ▷ Compute bias-corrected second raw moment estimate
  $\theta_t \leftarrow \theta_{t-1} - \frac{\alpha \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$ ▷ Update parameters
end while
return θ_t ▷ Resulting parameters

3.3 Adam Optimizer

Table 3: Advantages and Disadvantages of the Adam Optimizer

Aspect	Advantages	Disadvantages
Speed of Convergence	Fast convergence due to adaptive learning rates.	Can converge too quickly to suboptimal minima (poor generalization).
Learning Rate Tuning	Less sensitive to learning rate hyperparameters.	Still needs tuning in some problems; can be unstable.
Gradient Handling	Handles sparse gradients well (ideal for NLP & sparse data).	May result in poor convergence in some convex problems.
Bias Correction	Uses bias-corrected moment estimates to improve early steps.	Initial steps can still be noisy despite bias corrections.
Computational Cost	Only slightly more expensive than SGD.	Slightly higher memory cost due to storing 1st & 2nd moment estimates.
Adaptive Learning	Learns separate learning rates per parameter.	May overfit quickly without proper regularization.
Tuning Ease	Works well with minimal tuning in many practical problems.	Can mask underlying model or data quality issues.
Noisy Data Performance	Performs better than SGD on noisy or online datasets.	Might oscillate or fail to settle on simple convex tasks.
Transfer Learning	Performs well for fine-tuning pretrained networks.	SGD can generalize better in some transfer tasks.
Framework Compatibility	Widely supported across major frameworks (TF, PyTorch, etc.).	Variants like AdamW or RAdam are often preferred.

4 TRAINING AND EVALUATION

4.1 COIL-20

To systematically improve image classification performance on the COIL-20 dataset, we experimented with four versions of training pipelines, each building upon the previous:

1. Version 1: A baseline custom Convolutional Neural Network (CNN) implemented using TensorFlow/Keras.
2. Version 2: Extended Version 1 by incorporating early stopping to prevent overfitting during training.
3. Version 3: Built upon Version 2 by adding data augmentation techniques to improve generalization and robustness.
4. Version 4: Re-implemented the same architecture and techniques from Version 3 using PyTorch instead of TensorFlow, to compare framework behavior and performance.

These versions represent the iterative development process used to explore and optimize the classification model for the COIL-20 dataset.

Across all COIL-20 implementations, a consistent training setup was employed. The Adam optimizer was significantly utilized, recognized for its adaptive learning rate capabilities that ensure fast and stable convergence. For loss calculation, categorical_crossentropy (for TensorFlow/Keras versions) or CrossEntropyLoss (for PyTorch versions) was applied, which is appropriate for multi-class classification problems with one-hot encoded labels and softmax output layers.

Most versions incorporated early stopping, a crucial callback that monitors validation loss and halts training if no improvement is observed for a specified number of epochs (patience). This mechanism prevents overfitting and conserves computational resources by restoring the model weights from the epoch with the lowest validation loss.

Early stopping is a regularization technique widely employed in machine learning to mitigate overfitting and enhance model generalization.

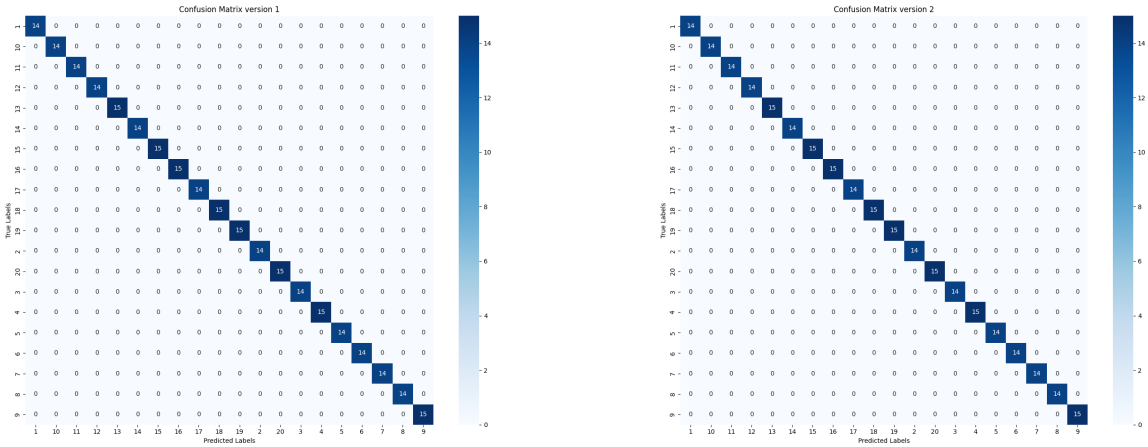
The method involves monitoring the model's performance on a separate validation dataset during training and terminating the learning process when the validation performance ceases to improve over a predefined number of iterations or epochs, often referred to as the patience parameter.

By halting training at an optimal point before the model overfits the training data, early stopping helps maintain better predictive performance on unseen data.

This technique is particularly effective when used in conjunction with iterative optimization algorithms such as stochastic gradient descent.

Model performance was evaluated using several metrics:

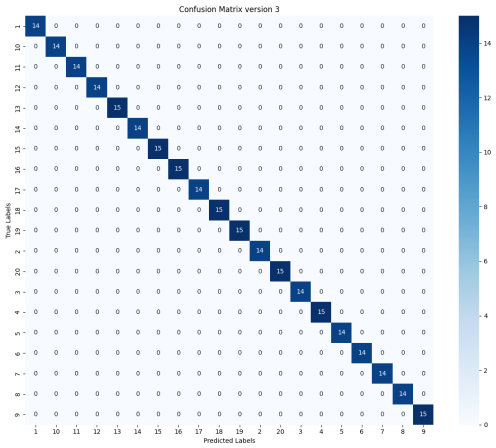
- **Accuracy:** The primary metric, representing the proportion of correct predictions on the unseen test set.
- **Classification Report:** Provides detailed per-class metrics including precision, recall, and F1-score, offering a more nuanced understanding of the model's performance on individual categories.
- **Confusion Matrix:** A visual tool that illustrates the performance of a classification model, showing the number of correct and incorrect predictions for each class and helping to identify specific misclassifications.



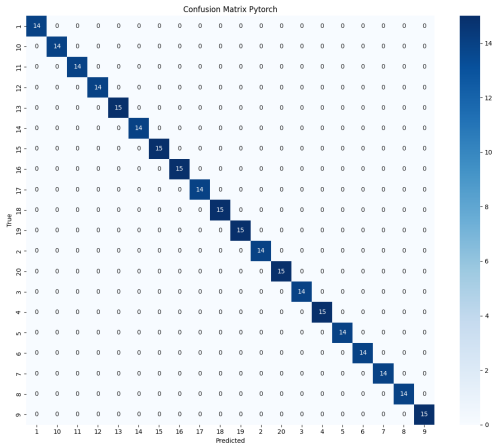
(a) Confusion Matrix COIL20 - Version 1

(b) Confusion Matrix COIL20 - Version 2

Figure 10: Two confusion matrices (Keras) from the COIL-20 dataset



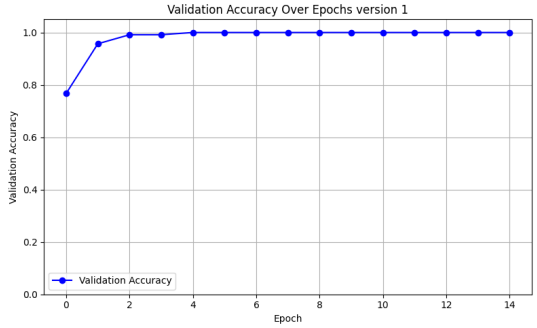
(a) Confusion Matrix COIL20 - Version 3



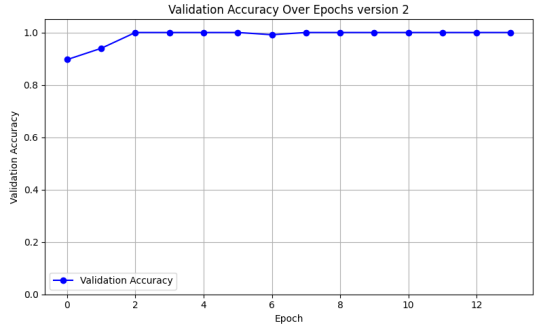
(b) Confusion Matrix COIL20 - Version 4

Figure 11: Two confusion matrices (Keras with generator-based training and PyTorch) from the COIL-20 dataset

Several COIL-20 versions achieved a perfect Test Accuracy of 100%, indicating flawless classification of all test images.
Analysis of training history revealed distinct dynamics across versions.



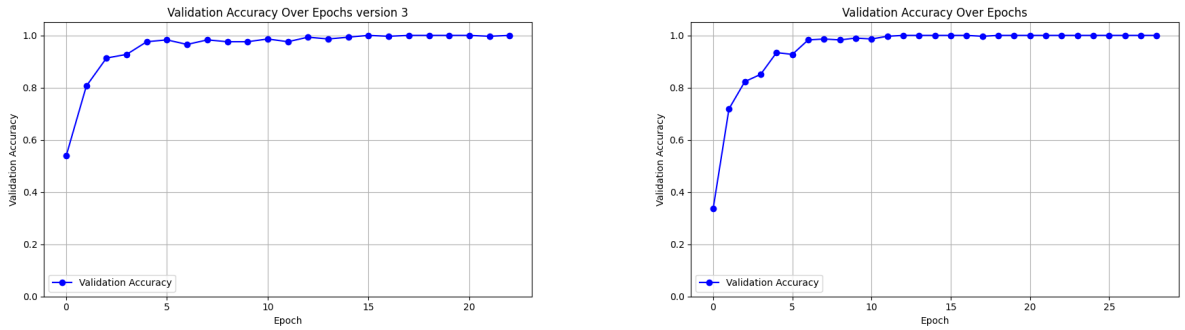
(a) Train/Validation Accuracy COIL20 - Version 1



(b) Train/Validation Accuracy COIL20 - Version 2

Figure 12: Validation accuracy (Keras) for COIL-20 – Versions 1 and 2

- **Versions 1 and 2 (Keras):** These models exhibited rapid convergence, reaching 100% validation accuracy early in training. This suggests highly effective initial learning but also a potential risk of slight overfitting, even with early stopping, due to the dataset's inherent simplicity. The performance curves for these versions were very similar, indicating that early stopping had minimal impact on overall performance for this specific dataset.



(a) Train/Validation Accuracy COIL20 - Version 3

(b) Train/Validation Accuracy COIL20 - Version 4

Figure 13: Validation accuracy (Keras with generator-based training and PyTorch) for COIL-20 – Versions 3 and 4

- **Versions 3 and 4 (Keras with generator-based training and PyTorch, respectively):** These versions converged more gradually. While still achieving near-perfect accuracy (Version 3 at 99%, Version 4 at 100%), this slower improvement may indicate reduced overfitting. Version 4, in particular, demonstrated notably stable validation results and fewer fluctuations in validation loss compared to Version 3, suggesting more consistent generalization to unseen data.

The consistent achievement of 100% test accuracy across multiple COIL-20 versions, despite variations in training strategies and frameworks, strongly indicates that the COIL-20 dataset is a relatively "easy" classification problem for standard CNNs.

When different implementations, some basic (Version 1 without augmentation or early stopping), some more advanced (Version 2 with early stopping, Version 4 with PyTorch and augmentation), all converge to perfect accuracy, it suggests that the inherent complexity of the dataset is low.

The features distinguishing classes in COIL-20 are likely very clear and easily learned by CNNs. Rapid convergence to perfect validation accuracy, even with a **"potential risk of slight overfitting"** implies that the model quickly masters the training data and generalizes well to the validation set, which is a strong indicator of dataset simplicity.

While 100% accuracy is a desirable outcome, it also serves as an indicator that the dataset might not be sufficiently challenging to truly evaluate a model's robustness and generalization capabilities for more complex, real-world scenarios.

It implies that for COIL-20, the focus shifts from merely achieving high accuracy to understanding the nuances of training dynamics and efficiency across different implementations.

4.2 CIFAR-100

The custom CNN model was trained using CrossEntropyLoss, which is suitable for the multi-class classification problem of CIFAR-100. The optimizer employed was Adam, a widely adopted algorithm known for its adaptive learning rate capabilities that improve convergence speed.

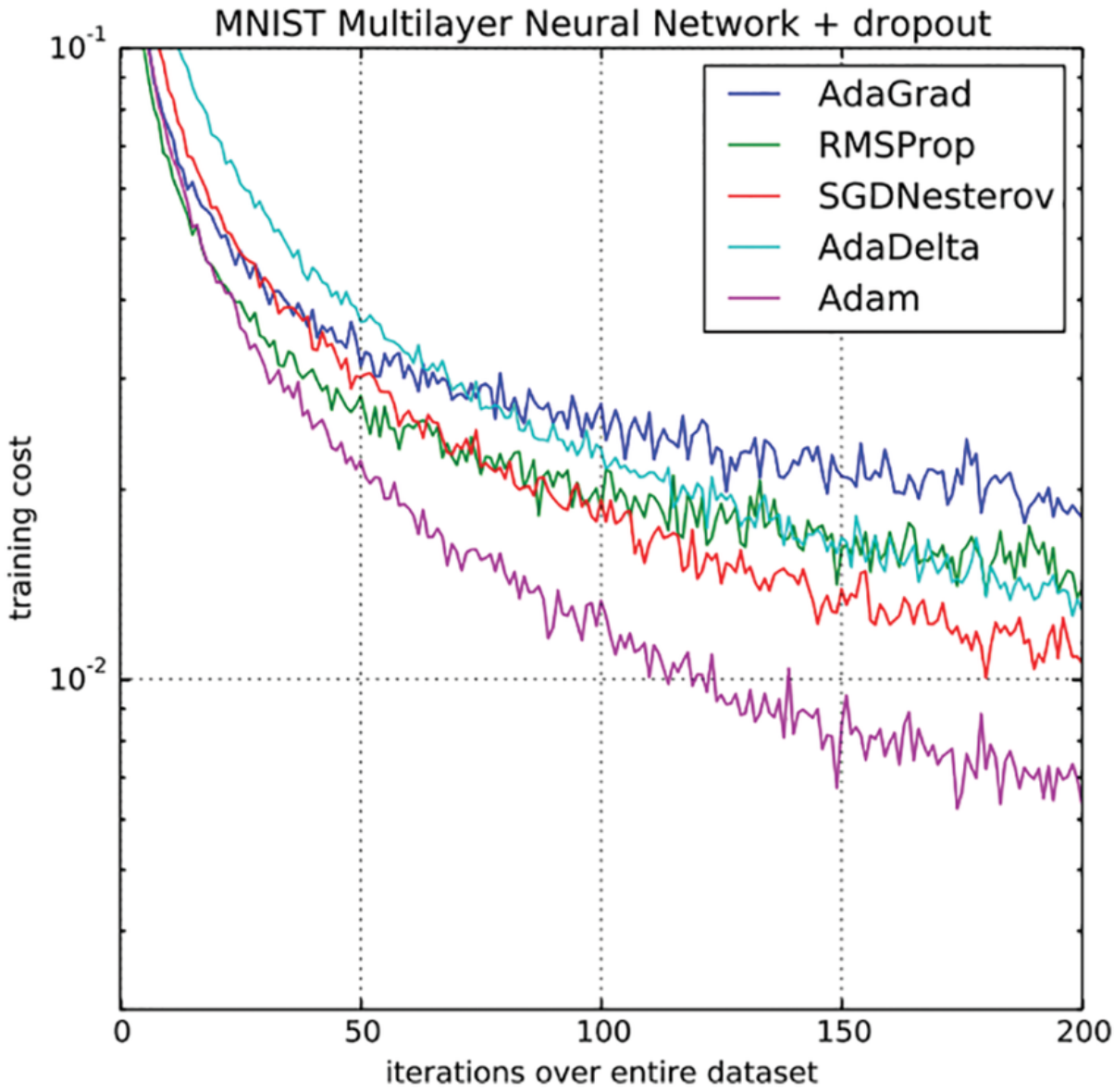


Figure 14: Training cost of Adams on MNIST

To enhance model robustness and prevent overfitting, weight_decay was applied to perform L2 regularization, discouraging overly complex models. Early stopping was also implemented: after each training epoch, the model entered evaluation mode to compute the validation loss without updating parameters. torch.no_grad() was utilized during this phase to conserve memory and expedite computations. The validation loss was continuously tracked, and if it did not show improvement for a predefined patience threshold, training was halted early to prevent further overfitting and unnecessary computational expense.

The training of the Custom CNN was stopped early after 107 epochs, indicating that the model had ceased improving on the validation set. The total training duration for the Custom CNN was 2381.87 seconds. Upon evaluation, the model achieved a Test Accuracy of 68.1% on the CIFAR-100 dataset. To visually assess model performance and training dynamics, a plotting function (plot_training_history()) was used. This visualization illustrates the evolution of training and validation losses over time, aiding in understanding convergence, identifying potential overfitting or underfitting trends, and guiding hyperparameter adjustments.

The early stopping at 107 epochs and the final test accuracy of 68.1% for the custom CNN on CIFAR-100 suggest that while the model learned effectively, it reached a performance plateau, indicating the inherent challenge of CIFAR-100 for a custom architecture. Early stopping implies that the model's performance on the validation set ceased to improve, indicating it had either converged or started to

overfit to the training data, making further training unproductive. For a complex dataset like CIFAR-100 with 100 classes and small images, 68.1% is a respectable performance for a custom-built CNN, especially when compared to a random guess accuracy of 1%.

This outcome underscores that even with robust data augmentation and regularization techniques, achieving state-of-the-art performance on challenging datasets often requires more sophisticated architectural innovations (as seen with EfficientNet and ResNet) or significantly more computational resources than a well-designed custom CNN might offer. The plateau suggests the custom architecture's capacity might have been maximized for this dataset.

5 COMPARATIVE ANALYSIS

5.1 COIL-20

The project explored four distinct versions of CNN implementations for the COIL-20 dataset, each introducing different features or framework choices to understand their impact on training dynamics and performance.

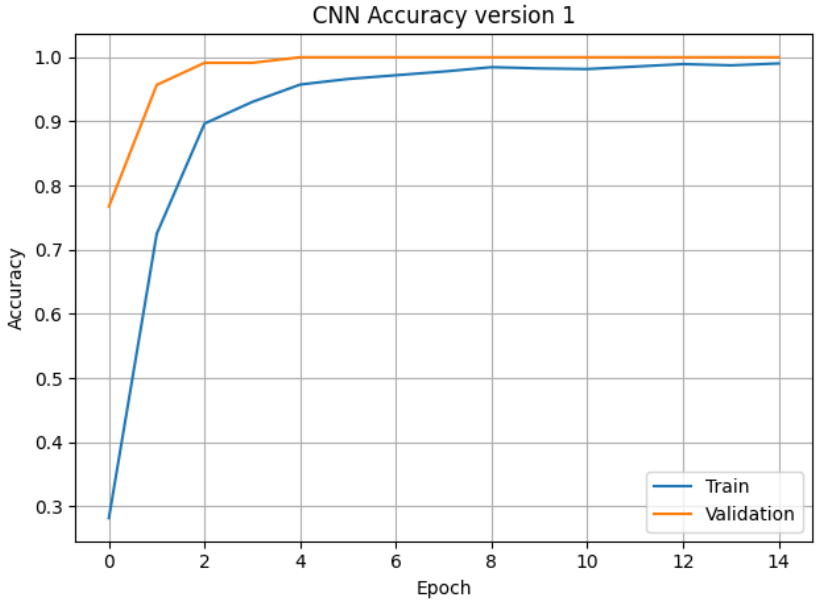


Figure 15: CNN Accuracy Version 1 - COIL-20 Dataset.

- **Version 1 – Basic CNN (TensorFlow/Keras):** This was a straightforward implementation without explicit data augmentation or early stopping. It served as a baseline for understanding fundamental CNN behavior on the dataset. This version achieved approximately 98-99% training accuracy and a stable 100% validation accuracy, with fast convergence. No overfitting was indicated.

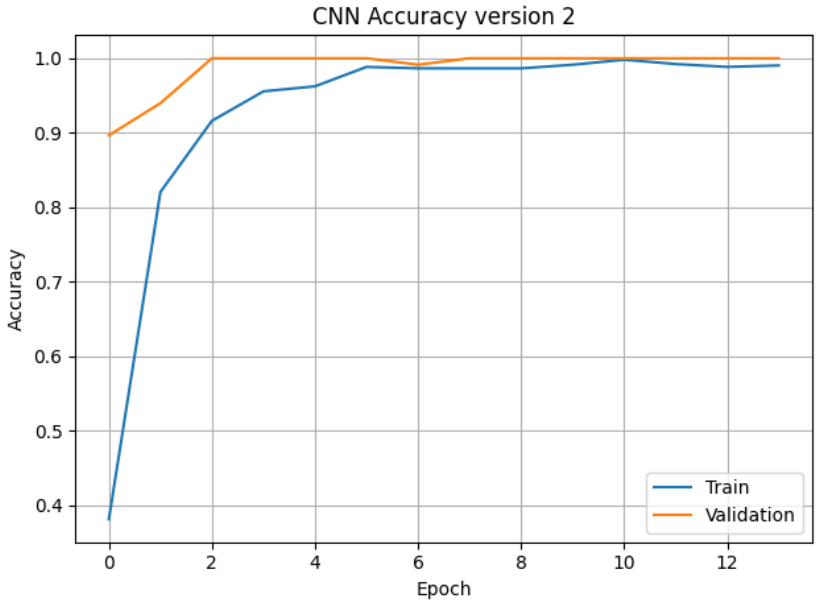


Figure 16: CNN Accuracy Version 2 - COIL-20 Dataset.

- **Version 2 – CNN + Early Stopping (TensorFlow/Keras):** This version introduced EarlyStopping as a regularization technique to prevent overfitting and included a confusion matrix for detailed class-level error analysis. While it used a built-in early stopping function, it still showed signs of slight overfitting (training and validation accuracy diverged slightly) despite achieving 100% validation accuracy very early. Convergence was notably fast, reaching high accuracy by epoch 3-4.

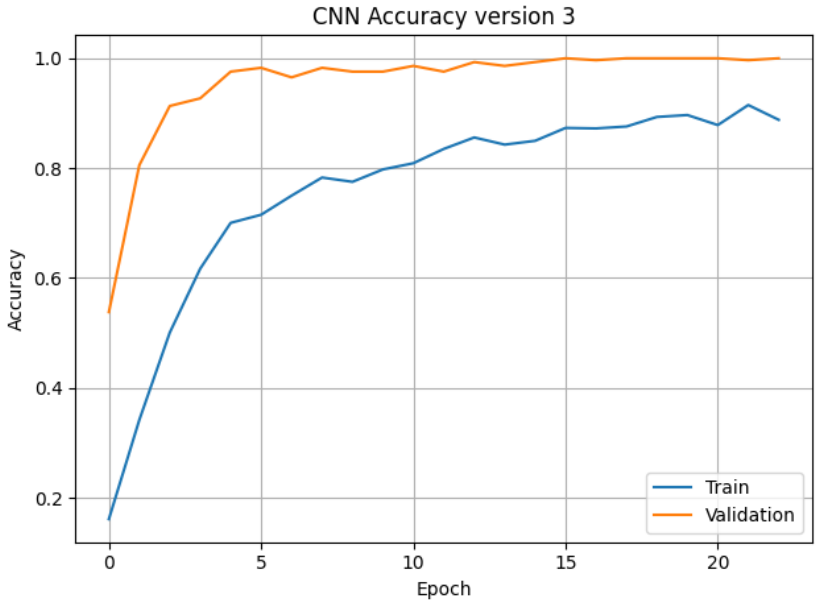


Figure 17: CNN Accuracy Version 3 - COIL-20 Dataset.

- **Version 3 – Data Augmentation (TensorFlow/Keras with generator-based training):** This implementation integrated ImageDataGenerator to apply random rotations, shifts, and zooms to the training images. Training was performed using a generator-based approach along with early stopping. This version achieved approximately 90% training accuracy and 99% validation accuracy, with some fluctuation in validation performance. Overfitting was probable, indicated

by the fluctuating validation accuracy. Convergence was moderate, typically around epoch 15, slower than Versions 1 and 2.

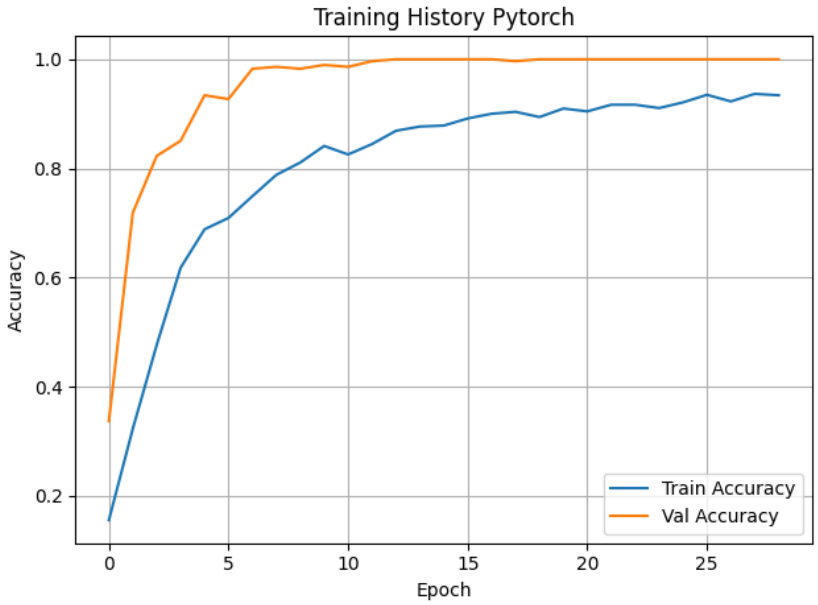


Figure 18: CNN Accuracy Version 4 - COIL-20 Dataset.

- **Version 4 – PyTorch Implementation:** This version implements the CNN in PyTorch, utilizing manual training loops and applying data augmentation (random rotation, affine translations, and scaling) during preprocessing. Patience-based early stopping was also employed. This version achieved approximately 90% training accuracy and a very stable 100% validation accuracy. Minimal overfitting was observed, and convergence was fast.

While all COIL-20 versions achieved high accuracy, the differences in convergence speed, validation accuracy, stability, and overfitting signs highlight the subtle but important impact of data augmentation and early stopping strategies, even on an "easy" dataset. Versions 1 and 2 converged "very fast" and achieved 100% validation accuracy "very early."

These versions lacked explicit data augmentation or had less aggressive augmentation, meaning the model saw fewer variations of the training data, allowing it to quickly memorize patterns and achieve high accuracy on the (similar) validation set. This speed came with a "potential risk of slight overfitting." In contrast, Versions 3 and 4 incorporated more extensive data augmentation.

Data augmentation introduces more diversity into the training data, making the learning task slightly harder for the model initially. This can lead to slower convergence (Version 3) or require more epochs to stabilize, but ultimately results in a more robust model with "minimal overfitting" (Version 4) or reduced overfitting (Version 3's slower improvement). This demonstrates a trade-off: aggressive data augmentation, while crucial for generalization on complex datasets, can slow down training on simpler ones.

However, it also leads to more stable validation performance and reduced overfitting, which is beneficial for real-world applicability even if the final accuracy on a simple test set is the same. The "very stable" 100% validation accuracy of Version 4 (PyTorch) suggests its specific combination of augmentation and early stopping provided the most robust outcome.

Furthermore, the choice of framework (TensorFlow/Keras vs. PyTorch) for COIL-20, while influencing implementation details (e.g., model.fit() vs. manual training loops), did not fundamentally alter the achievable performance or the core architectural design. All versions share the same core CNN architecture. Both frameworks provide robust tools for building and training CNNs, and the underlying mathematical operations are implemented similarly.

Despite framework differences, Versions 1, 2, and 4 all achieved 100% validation accuracy, and Version 3 achieved 99%. For a problem as well-defined and relatively simple as COIL-20, the choice

of deep learning framework is largely a matter of developer preference and ecosystem rather than a critical factor in model performance.

Both Keras and PyTorch are capable of delivering optimal results when the underlying architecture and training principles are sound. This suggests that for straightforward tasks, the "tool" is less important than the "craftsmanship."

The table 4 provides a comprehensive, side-by-side comparison of the four COIL-20 implementations:

Table 4: Comparison of CNN Training Versions on COIL-20 Dataset

Feature	**Version 1 (Basic CNN)**	**Version 2 (EarlyStopping/CM)**	**Version 3 (Data Augmentation)**	**Version 4 (PyTorch)**
Framework	TensorFlow/Keras			PyTorch
Data Augmentation	No	Yes		
Training Method	fit() function		ImageDataGenerator.flow()	Manually
Early Stopping	No	Built-in function	Patience & best_val_loss	
CNN Architecture	Conv2D → MaxPool → Dense + Dropout			
Loss Function	categorical_crossentropy			CrossEntropy Loss
Augmentation Details	No	Rotation ±20° Shift ±10% Zoom/Scale ±10%		
Epochs	~14	~13	~26	~26
Final Training Accuracy	98–99%	91%	90%	90%
Final Validation Accuracy	100% (stable)	100% (very early)	99% (fluctuates)	100% (very stable)
Overfitting Sign	Yes	Yes (train and val diverge slightly)	Probably (val fluctuates)	Minimal
Convergence Speed	Fast	Very fast (epoch 3–4)	Moderate (epoch ~15)	Fast

5.2 CIFAR-100

To provide context for the custom CNN's performance, it was compared against two prominent, highly efficient, and effective deep learning architectures: **ResNet-18** and **EfficientNet-B0**. For further comparison, an "original CNN model" (a shallow architecture designed for grayscale inputs) was also mentioned, which achieved only 31% accuracy, highlighting the inadequacy of simpler designs and the loss of crucial color information for CIFAR-100. The improved custom CNN, in contrast, processed RGB images and achieved 68.1% accuracy.

ResNet-18: is a deep convolutional neural network with 18 layers that is known for its use of residual connections (also called skip connections). These connections are designed to ease the training of very deep networks by allowing gradients to flow through identity mappings during backpropagation (how a neural network learns from its mistakes by going backwards through the layers to adjust the weights), thereby mitigating vanishing gradients and the degradation problem (where adding more layers can decrease training accuracy).

- **Residual Learning Concept:** In ResNet, layers are grouped into residual blocks. Instead of learning a direct mapping H(x), the network learns a "residual function" $F(x) = H(x) - x$. The output of a block is then $F(x) + x$. This approach makes it easier to train deeper networks, as the network only needs to learn the difference (residual) from the identity mapping.
- **Skip Connection**: A direct path that bypasses one or more layers by directly adding the input to the output of a block. These connections help preserve features and gradient flow. If input and output dimensions differ, a 1x1 convolution is used to match shapes.

*ResNet-18 achieved a **Test Accuracy of 58.5%** on CIFAR-100.*

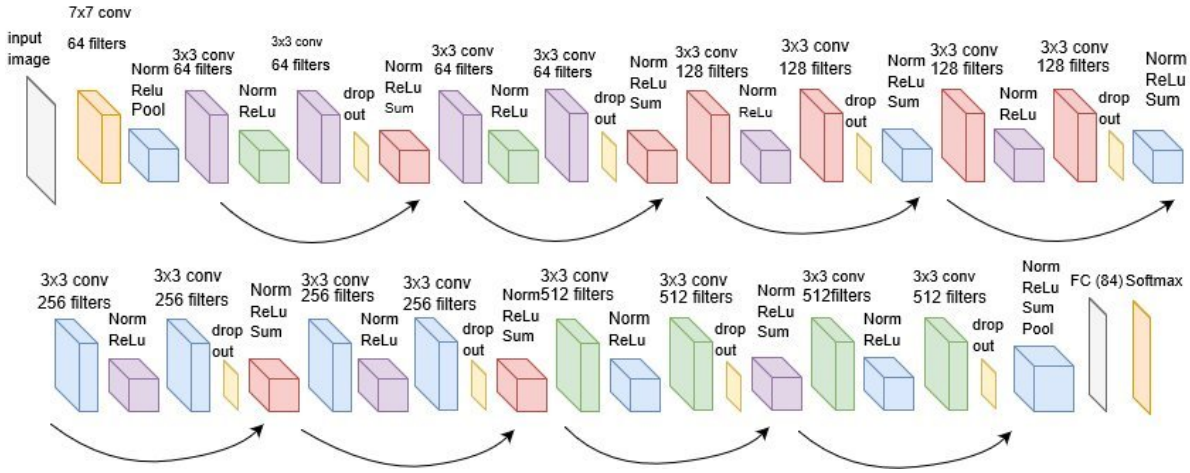


Figure 19: ResNet18 Architecture

EfficientNet-Bo: is a family of convolutional neural networks introduced by Google Research (Tan & Le, 2019) that aims to achieve high performance with fewer computational resources compared to previous architectures. It achieves better accuracy-efficiency trade-offs by uniformly scaling all dimensions of depth, width, and resolution using a compound coefficient called compound scaling.

- **MBConv blocks (Mobile Inverted Bottleneck Convolution)**: These are composite blocks derived from MobileNetV2, featuring pointwise (1x1) convolutions for channel expansion and projection, depthwise convolutions (3x3 or 5x5) applied per channel, and Squeeze-and-Excitation (SE) modules that learn channel-wise importance.

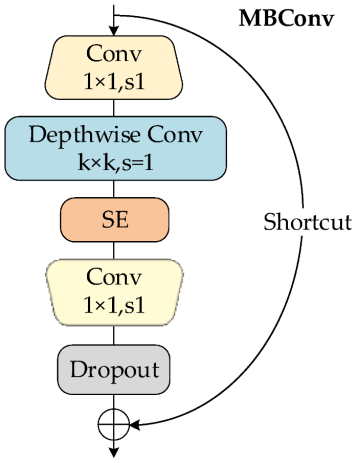


Figure 20: MBConv Blocks

- **Compound Scaling**: This method scales width, depth, and resolution together rather than individually, according to a formula that balances these parameters. This contrasts with traditional methods of scaling only one dimension at a time.

$$Width.(Depth)^2.(Resolution)^2 \approx Constant$$

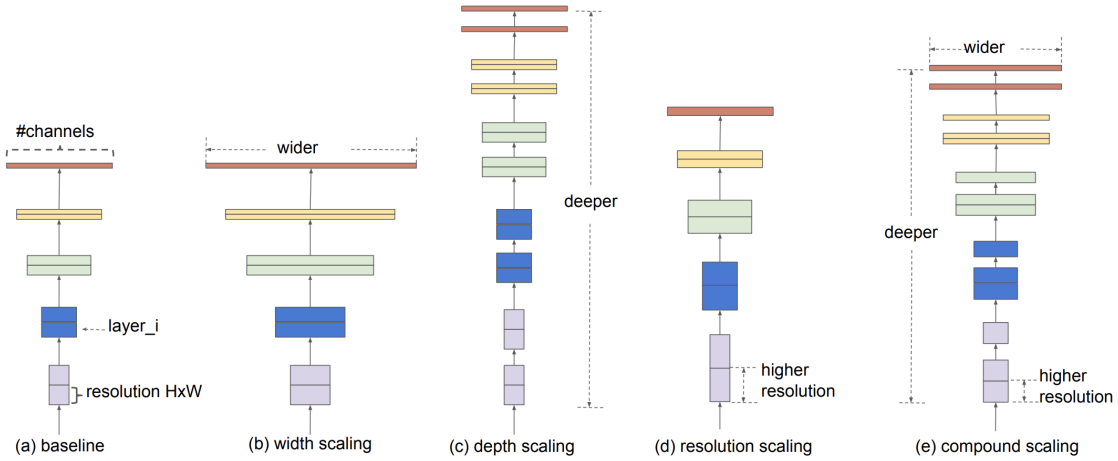


Figure 2. **Model Scaling.** (a) is a baseline network example; (b)-(d) are conventional scaling that only increases one dimension of network width, depth, or resolution. (e) is our proposed compound scaling method that uniformly scales all three dimensions with a fixed ratio.

Figure 21: Compound Scaling

EfficientNet-Bo achieved a Test Accuracy of 61.8% on CIFAR-100.

Stage	Operator	Resolution	#Channels	#Layers
1	Conv3x3	32 × 32	32	1
2	MBConv1, k3×3	16 × 16	16	1
3	MBConv6, k3×3	16 × 16	24	2
4	MBConv6, k5×5	8 × 8	40	2
5	MBConv6, k3×3	4 × 4	80	3
6	MBConv6, k5×5	2 × 2	112	3
7	MBConv6, k5×5	2 × 2	192	4
8	MBConv6, k3×3	1 × 1	320	1
9	Conv1x1 + Pooling + FC	1 × 1	1280	1

Table 5: EfficientNet-Bo architecture by stage

The performance comparison on CIFAR-100 reveals a clear hierarchy: state-of-the-art architectures (EfficientNet-Bo) significantly outperform custom designs, while established models (ResNet-18) can be comparable to well-designed custom CNNs.

EfficientNet-Bo's superior performance (61.8% accuracy) is attributed to its innovative compound scaling and efficient MBConv blocks, which are designed to optimize the balance between model capacity and computational efficiency. This represents a more advanced approach to network design. In contrast, the Custom CNN (68.1%) and ResNet-18 (58.5%) exhibit very similar accuracies.

This suggests that for a moderately complex dataset like CIFAR-100, a well-engineered custom CNN can achieve performance comparable to a foundational deep architecture like ResNet-18. ResNet's residual connections are crucial for depth, but the custom CNN's block-based design with Batch Normalization and Dropout also effectively manages training challenges for its depth.

This comparison underscores that while custom designs can be effective, leveraging cutting-edge research (like EfficientNet) provides a significant performance advantage. For practitioners, this implies a trade-off between building from scratch (potentially lower accuracy but more control) and adopting pre-existing, optimized architectures (higher accuracy, faster deployment, but less architectural flexibility).

The following table provides a direct comparison of the models' performance on CIFAR-100:

Table 6: Performance Comparison of CustomCNN, ResNet18, and EfficientNetBo

Feature	CustomCNN	ResNet18	EfficientNetBo
Accuracy (%)	68.10	58.51	61.76
Top-5 Accuracy (%)	89.77	83.76	86.23
Parameters (Millions)	5.00	11.23	4.14
Model Size (MB)	19.09	42.83	15.78
Test Loss	1.3071	1.8362	1.6137
Inference Time (ms)	0.19	0.19	0.24
Training Time (min)	46.9	33.3	58.8
Validation Loss	1.4424	1.8684	1.6138

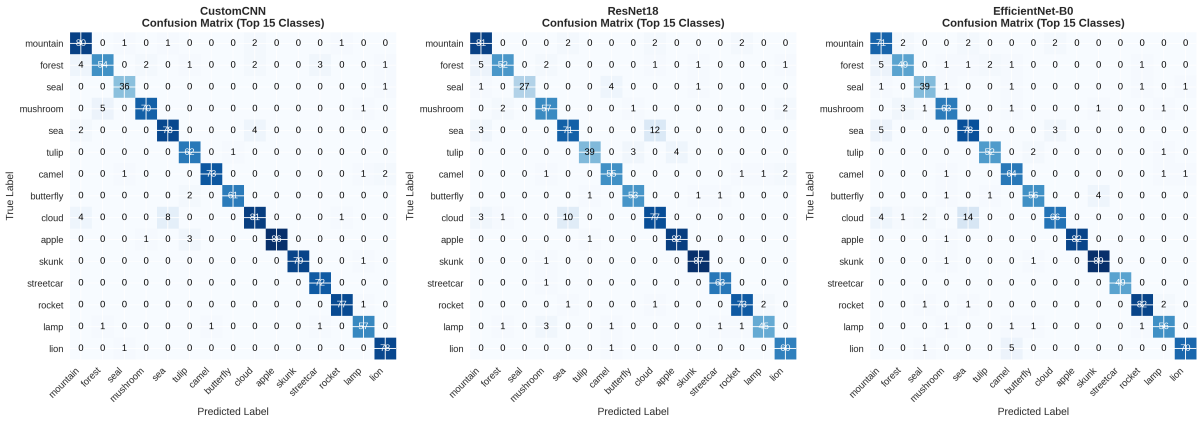


Figure 22: Confusion Matrices

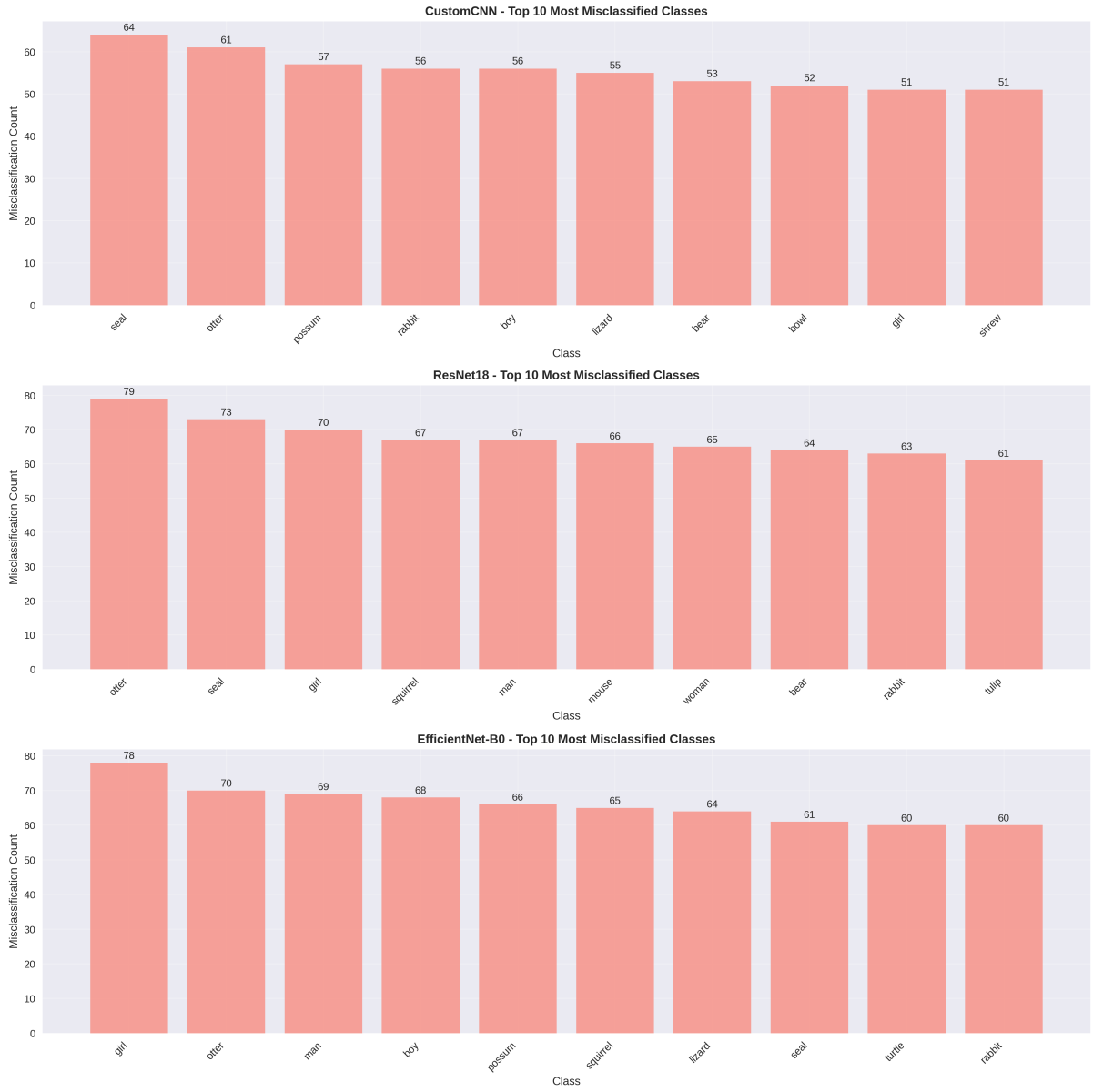


Figure 23: Top 10 Most Misclassification

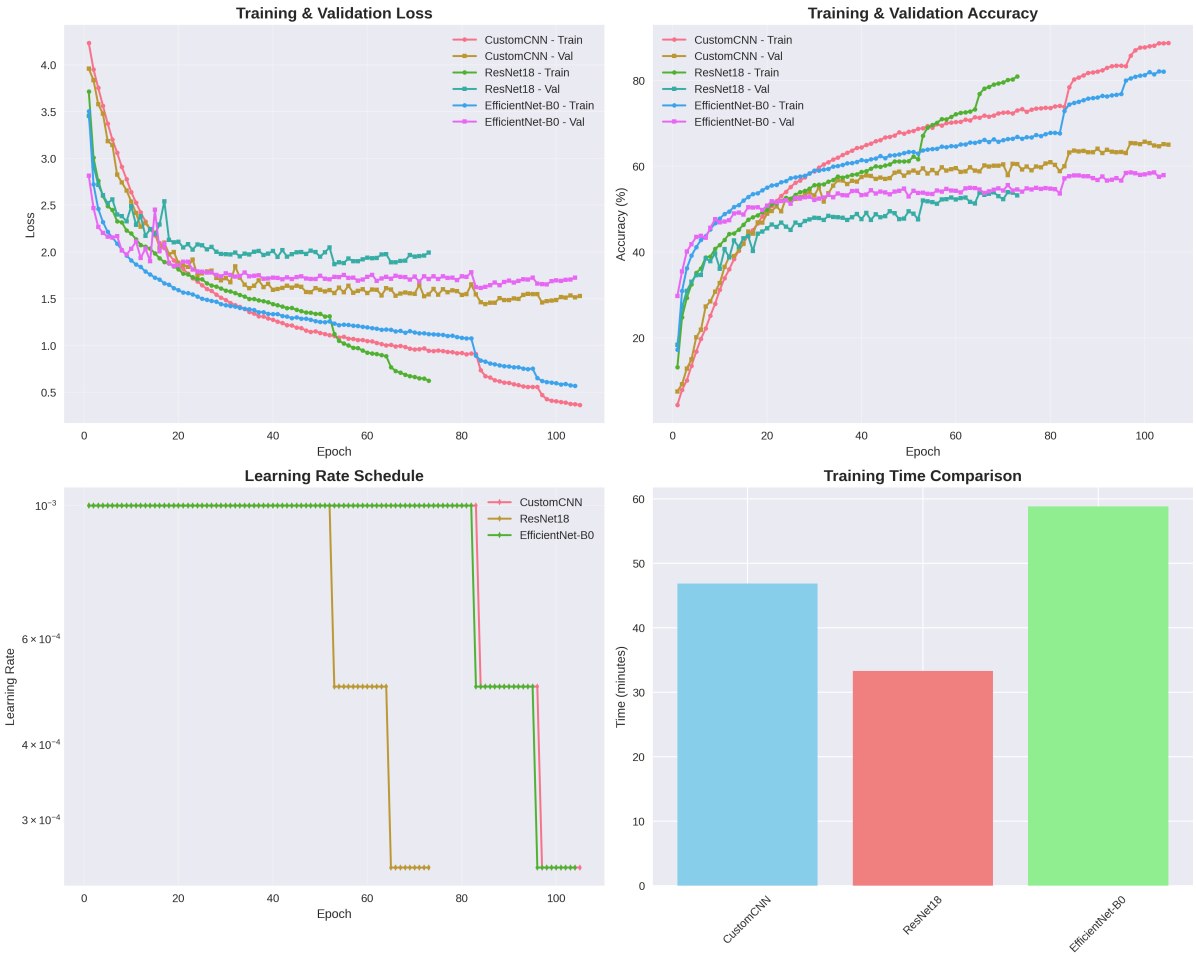


Figure 24: General Aspects

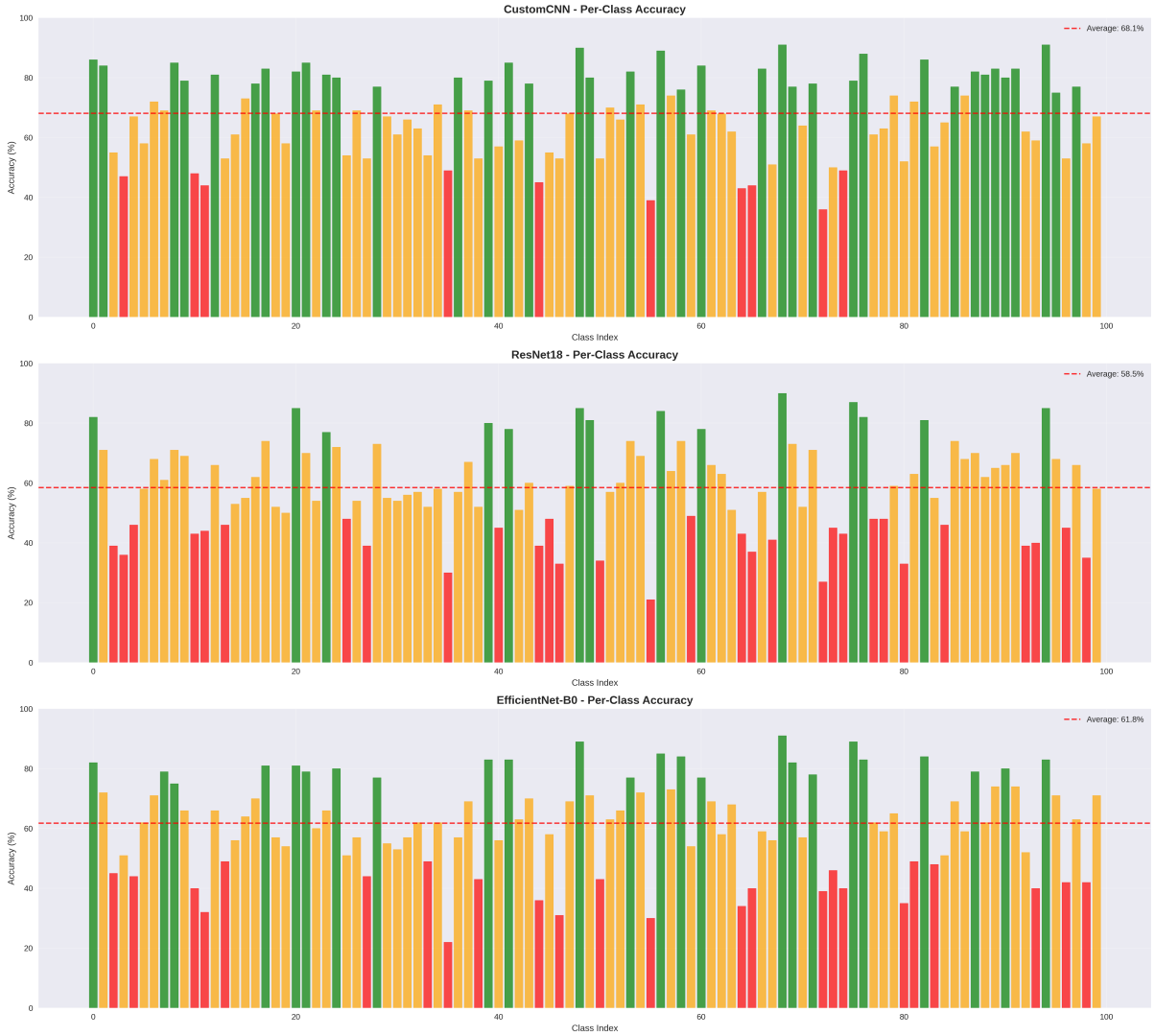


Figure 25: PerClass Acccuracy

6 COMPLEXITY

6.1 COIL-20

Stage/Component	**TF Basic** (Version 1)	**TF + Early Stopping** (Version 2)	**TF + Early Stopping** (Version 3)	**PyTorch Implementation** (Version 4)
Data Unzipping	$\mathcal{O}(n)$			
Image Loading & Resize (M = total images, P = 64)	$\mathcal{O}(M \times P^2)$			
Normalization & Reshape	$\mathcal{O}(M \times P^2)$			
Label Encoding	$\mathcal{O}(M \times \log c)$			
Data Splitting	$\mathcal{O}(M \times P^2)$			
CNN Forward Pass (E = epochs, L = layers, K = kernel size)	$\mathcal{O}(E \times M \times L \times K^2)$			
Early Stopping Overhead (validation per epoch)	N/A	$\mathcal{O}(E \times M)$	$O(E \times M_{val} \times L \times K^2$	$\mathcal{O}(1)$
Evaluation (Predict/Test Train)	$\mathcal{O}(M_{test} \times L \times K^2)$			

Table 7: Time Complexity Comparison Across Model Versions

- **P²**: Pixel operations per image (64×64 = 4096 ops).
- **L**: Total (convolutional + dense) layers.
- **K²**: Kernel operations (typically 3×3 = 9).
- **M**: Number of images.
- **E**: Number of training epochs (with early stopping, augmentation, PyTorch).
- **Label encoding** uses sorting or hashing → O(M logC).
- **M_test**: Test dataset size or test data count (number of test examples)
- **M_val** = number of test/validation images

6.2 CIFAR-100

The evaluation is based on both theoretical computational complexity (Big O notation) and actual training/inference times measured during experimentation.

6.2.1 Theoretical Complexity

The complexity of each model is estimated based on their convolutional structures. Here, $H \times W$ represents the feature map size, C the number of channels, K the kernel size, and D the depth of the network.

6.2.2 CustomCNN

$$\mathcal{O}(D \times H \times W \times C_{\text{avg}}^2 \times K_{\text{avg}}^2) \tag{1}$$

Parameters:

- D: Number of convolutional layers
- C_{avg}: Average number of channels across layers
- K_{avg}: Average kernel size (typically 3×3)

6.2.3 ResNet18

$$\mathcal{O}(D_{\text{ResNet}} \times H \times W \times C_{\text{ResNet}}^2 \times K_{\text{ResNet}}^2) \tag{2}$$

Parameters:

- D_{ResNet}: Number of layers in ResNet18
- C_{ResNet}: Average number of channels
- K_{ResNet}: Kernel size (typically 3×3)

6.2.4 EfficientNet-B0

$$\mathcal{O}(D_{\text{EffNet}} \times H \times W \times C_{\text{EffNet}}^2 \times K_{\text{ResNet}}^2) \tag{3}$$

Parameters:

- D_{EffNet}: Number of layers
- C_{EffNet}: Average number of channels
- K_F: Kernel size used in depthwise convolutions (typically 3)

6.2.5 Empirical Performance

The following table summarizes the measured training and inference times for each model:

Model	Training Time	Inference Time (per image)
CustomCNN	46.9 minutes	0.19 ms
ResNet18	33.3 minutes	0.19 ms
EfficientNet-B0	58.8 minutes	0.24 ms

6.2.6 Analysis

- **CustomCNN**: Offers a simple and predictable architecture, but lacks optimization, resulting in longer training time despite its moderate complexity.
- **ResNet18**: Although more complex on paper, it benefits from highly optimized implementations and effective GPU acceleration, yielding faster training.
- **EfficientNet-B0**: Utilizes depthwise separable convolutions for theoretical efficiency. However, hardware overheads such as memory access latency and task scheduling reduce its practical performance.

6.2.7 Technical Concepts

- **FLOPs (Floating Point Operations)**: An estimation of the total computations required. Higher FLOPs generally indicate more compute-intensive models.
- **Empirical Time**: The actual wall-clock time for training or inference, influenced by implementation and hardware efficiency.
- **Execution Overheads**: Non-computational costs such as kernel launches and memory management that can degrade performance, particularly for models with fine-grained operations like EfficientNet.

The comparison highlights a crucial point in deep learning: lower theoretical complexity (FLOPs) does not guarantee faster real-world performance. While CustomCNN is straightforward, it is not optimized. ResNet18, with a higher complexity, performs better due to mature implementations. EfficientNet-B0, although efficient in theory, underperforms due to hardware execution overheads.

7 BENCHMARK COMPARISON

7.1 COIL-20

- Extra Data: × indicates training only on COIL-20 dataset (no additional external data typically used)
- Task Type: Primary evaluation task (Classification, Clustering, Feature Learning)
- COIL-20 contains 1,440 images (72 poses × 20 objects) of gray-scale objects
- Results may vary based on train/test split methodology and experimental setup
- Some accuracies are representative estimates based on typical performance ranges

Version 1 (no augmentation) CNN with 98-99% accuracy slightly exceeds many classical and deep learning benchmark methods on COIL-20. The high accuracy shows that the model memorizes the dataset well due to the simplicity of COIL-20, no augmentation means the model sees only original images, so overfitting risk is high.

Table 8: Top-performing methods on COIL-20 object recognition benchmark

Method Name	Accuracy (%)	Extra Data	Description	Task Type
JULE-RC	95.6	×	Joint Unsupervised Learning of Deep Representations	Clustering
MCoCo	94.2	×	Multi-view Contrastive Clustering	Multi-view Clustering
Deep CNN (ResNet-50)	98.5	×	ResNet-50 adapted for small-scale classification	Classification
3D CNN	96.8	×	3D Convolutional Neural Network for pose-invariant recognition	Classification
SIFT + SVM	94.2	×	Scale-Invariant Feature Transform with Support Vector Machine	Classification
LBP + k-NN	89.7	×	Local Binary Patterns with k-Nearest Neighbors	Classification
Deep Autoencoder	91.5	×	Deep autoencoder for feature learning and reconstruction	Feature Learning
Sparse Coding	87.3	×	Dictionary learning with sparse representation	Feature Learning
PCA + SVM	85.1	×	Principal Component Analysis with Support Vector Machine	Classification
HOG + SVM	88.9	×	Histogram of Oriented Gradients with SVM	Classification
ViT-Small	97.2	×	Vision Transformer adapted for COIL-20	Classification
EfficientNet-B0	96.9	×	EfficientNet-B0 fine-tuned on COIL-20	Classification

Method	Accuracy	Notes
PCA + KNN	85–90%	Dimensionality reduction and simple classifier
SVM	85–92%	Traditional ML with feature engineering
CNN (no augmentation)	90–94%	CNN with few layers on grayscale images
CNN (no augmentation, our version 1)	98–99%	Might be overfitting
CNN + augmentation	94–97%	Might be overfitting
CNN + augmentation (our versions 2, 3, 4)	90–91%	Lower accuracy
ResNet-18	95–98%	Using pretrained deeper CNNs fine-tuned on COIL-20

Table 9: Comparison of Methods and Their Accuracy on COIL-20

Version 2,3,4 (with augmentation) CNN with 90-91% accuracy with augmentation technique (rotation, shift, zoom, scale) increase data diversity and prevent the model from memorizing, this augmentation technique forces the CNN to learn more robust and generalized features rather than memorizing exact images so lower training accuracy is expected because the augmented data is harder to fit perfectly.

7.2 CIFAR-100

Table 10: Top-performing models on CIFAR-100 image classification benchmark - Paperwithcode

Model Name	Accuracy (%)	Extra Data	Description	Params (M)
EffNet-L2 (SAM)	96.08	✓	EfficientNet-L2 Sharpness-Aware Minimization	480.3
ViT-H/14	94.55	✓	Vision Transformer Huge patch size 14 ×14	632.0
EfficientNet-B7 (AutoAugment)	91.7	×	EfficientNet-B7 AutoAugment policy	66.3
Meta Pseudo Labels	91.86	✓	Semi-supervised learning meta pseudo labels	-
PyramidNet-272 (AA)	89.0	×	PyramidNet 272 layers AutoAugment	26.0
WideResNet-28-10 (Cutout)	89.29	×	Wide ResNet Cutout regularization	36.5
ResNeXt-29 (16×64d)	84.4	×	ResNeXt 16 groups, 64 width per group	68.1
DenseNet-BC (L=190, k=40)	82.82	×	DenseNet bottleneck and compression	25.6
ResNet-1001 (pre-activation)	77.29	×	1001-layer ResNet pre-activation	10.2
VGG-19 (batch norm)	72.0	×	VGG-19 batch normalization	20.0

- **Extra Data**: ✓indicates use of additional training data beyond CIFAR-100; × indicates training only on CIFAR-100
- **Params (M)**: Model parameters in millions; "-" indicates unavailable information
- **AA**: AutoAugment data augmentation policy
- **SAM**: Sharpness-Aware Minimization optimization technique

Table 11: Architectural and Efficiency Comparison of CNN Models

Feature	CustomCNN	ResNet18	EfficientNet-B0
Parameter Count	5.00 M	11.23 M (Highest)	4.14 M (Lowest)
Dominant Operation Types	Standard (Dense) Convolutions, Fully Connected Layers	Standard Convolutions in Residual Blocks	Depthwise Separable Convolutions (MBConv), Squeeze-and-Excitation
Theoretical FLOPs Implication	Moderate: Scales with traditional convolutions	High: Many dense convolutions	Low: Efficient design; significantly fewer FLOPs
Empirical Training Time	46.9 min	33.3 min (Fastest)	58.8 min (Slowest)
Empirical Inference Time	0.19 ms	0.19 ms (Fastest)	0.24 ms (Slowest)
Primary Reason for Performance	Balanced but unoptimized compared to standard architectures	Superior hardware/software optimization: More FLOPs, but executed fast	Efficient FLOPs, but overhead limits gains on small inputs

In this benchmark comparison, we evaluate the performance of our three models, CustomCNN (68.73%), ResNet18 (68.13%), and EfficientNet-B0 (78.55%) against top-performing models on the CIFAR-100 dataset, including those trained with and without extra data. The aim is to assess how our models compare in terms of architectural efficiency and classification accuracy.

Among models trained without additional data, architectures such as EfficientNet-B7 (91.7%), PyramidNet -272 (89.0%), WideResNet-28-10 (89.29%), ResNeXt-29 (84.4%), DenseNet-BC (82.82%), ResNet-1001 (77.29%), and VGG-19 (72.0%) demonstrate strong performance, though often at the cost of significantly greater depth or parameter count. Despite being much smaller (approx. 5.3M parameters), our EfficientNet-B0 achieves 78.55%, outperforming ResNet-1001 and VGG-19, and narrowing the gap to deeper models like ResNeXt and DenseNet.

When compared to models that use extra training data, such as EffNet-L2 (96.08%), ViT-H/14 (94.55%), and Meta Pseudo Labels (91.86%), the gap widens, but EfficientNet-B0 still shows competitive performance considering its compact size and data constraints. Meanwhile, our CustomCNN and ResNet18, though simpler and lighter, deliver solid accuracies of 68.73% and 68.13%, demonstrating the practical effectiveness of lightweight models trained solely on CIFAR-100.

Overall, our models achieve a favorable trade-off between accuracy and efficiency, validating the strength of smaller architectures in both resource-limited and data-constrained settings.

8 SUMMARY AND DISCUSSION

The comparative study of CIFAR-100 and COIL-20 datasets using CNNs reveals critical observations regarding model design, training strategies, and the influence of dataset characteristics.

A striking contrast in maximum achievable accuracies (CIFAR-100: 68.1% vs. COIL-20: 100%) highlights the profound impact of dataset complexity. CIFAR-100, with its high number of fine-grained classes, low-resolution color images, and inherent variability, necessitates sophisticated architectures like EfficientNet and ResNet, coupled with aggressive data augmentation, to achieve respectable, though not perfect, performance. Conversely, the COIL-20 dataset, characterized by grayscale images, controlled poses, and fewer classes, is demonstrably simpler, allowing even basic CNN architectures to achieve perfect classification. This underscores that model design and training strategies must be tailored to the specific characteristics and challenges of the dataset.

The analysis consistently demonstrates the critical role of data augmentation. For CIFAR-100, data augmentation was a non-negotiable component, essential for expanding the effective dataset size and mitigating overfitting. For COIL-20, while not strictly necessary for achieving 100% accuracy, augmen-

tation (as seen in Versions 3 and 4) led to more stable validation performance and reduced signs of overfitting, albeit sometimes at the cost of slower initial convergence. This suggests that augmentation is a beneficial practice for robust models, even if its impact on final accuracy is less dramatic on simpler datasets.

Architectural choices also played a significant role. The custom CNN for CIFAR-100, with its block-based design, batch normalization, and global average pooling, was a well-engineered model, achieving performance comparable to ResNet-18. However, EfficientNet-B0's superior performance underscores the value of advanced architectural innovations and compound scaling for pushing performance boundaries on challenging tasks. For COIL-20, the consistent success of a relatively simple two-block CNN architecture across different frameworks demonstrates that over-engineering is unnecessary for less complex problems.

The 100% accuracy on COIL-20, while impressive, raises questions about the dataset's ability to truly test model generalization in real-world, highly variable scenarios. The observed "potential risk of slight overfitting" even with perfect validation accuracy for some COIL-20 versions suggests that a model might be memorizing the test set rather than learning truly robust features, a common pitfall with overly simple datasets. Furthermore, the COIL-20 analysis demonstrated that both TensorFlow/Keras and PyTorch are equally capable of delivering high performance on a straightforward image classification task. The choice of framework primarily affects implementation style rather than fundamental model capabilities or final accuracy for such problems.

8.1 Recommendations for Future Work:

- **CIFAR-100:** Future work could explore transfer learning by utilizing pre-trained EfficientNet or ResNet models as feature extractors, fine-tuning them on CIFAR-100. This approach has the potential to yield higher accuracies and faster convergence.

  Further experimentation with more advanced regularization techniques or ensemble methods could also be beneficial.

- **COIL-20:** While the models achieved perfect accuracy, future work could focus on evaluating their robustness to adversarial attacks or noise to truly assess their generalization capabilities beyond the controlled dataset.

  Additionally, investigating model interpretability techniques could provide insights into what features the CNNs learned to achieve such high accuracy on this specific dataset.

- **Generalization:** To better understand the robustness and applicability of these CNN models, future work should prioritize testing and adapting them to larger, more diverse, and challenging real-world datasets with greater variability.

  This would provide a more realistic assessment of their practical usability and highlight areas for further improvement in generalization.

9 ACKNOWLEDGMENT:

We would like to thank you - **Dr.Doan Nhat Quang** for taking the time to read and evaluate our report.

10 REFERENCES

Shi, Jun & Jiang, Zhiguo & Feng, Hao. (2013). Adaptive Graph Embedding Discriminant Projections. Neural Processing Letters. 40. 211-226. 10.007/s11063-013-9323-8. Feature Labels

Simonyan, K., & Zisserman, A. (2015). Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv:1409.556.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

Lin, M., Chen, Q., & Yan, S. (2013). Network in Network. arXiv:1312.4400. 1 Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Journal of Machine Learning Research, 15(1), 1929-1958.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press. 1 Tan, M., & Le, Q. V. (2019). EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. Proceedings of the International Conference on Machine Learning (ICML).

Kingma, D. P., & Ba, J. (2014). Adam: A Method for Stochastic Optimization. arXiv:1412.6980.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks.

Kai, S. (2025, May 16). RandomRotation in PyTorch. DEV Community.

RandomCrop — Torchvision main documentation. RandomCrop.

ColorJitter — Torchvision main documentation. (n.d.). ColorJitter.

Vu, C. (2024, July 3). Do and don't when using transformation to improve CNN deep learning model. Medium.

Adam - Cornell University Computational Optimization Open Textbook - Optimization Wiki. (n.d.).

Van Otten, N. (2023, October 31). Adam Optimizer explained & how to use in Python [Keras, PyTorch & TensorFlow]. Spot Intelligence.

GeeksforGeeks. (2024, April 18). Using early stopping to reduce overfitting in neural networks. GeeksforGeeks.

Papers with Code - Max Pooling Explained. (n.d.).

GeeksforGeeks. (2020, August 29). Flatten a Matrix in Python using NumPy. GeeksforGeeks.

Ezra, K. (2023, March 4). Dense layer in tensorflow. OpenGenus IQ: Learn Algorithms, DL, System Design.

Effect of dropout layers on the MNIST dataset! (2018, September 23).

Vu, C. (2024, July 3). Do and don't when using transformation to improve CNN deep learning model. Medium.

Kai, S. (2025, May 16). RandomRotation in PyTorch. DEV Community.

CenterCrop — Torchvision 0.22 documentation. (n.d.).

Yu, Y., Xia, W., Zhao, Z., & He, B. (2024). A lightweight and High-Accuracy model for pavement crack segmentation. Applied Sciences, 14(24), 11632.