

Deep Learning Project

Natalia Safiejko, Wojciech Grabias

March 2025

Contents

1	Introduction	4
2	Literature review	4
2.1	Existing approaches	4
2.1.1	Memory-Replay Knowledge Distillation [20]	4
2.1.2	CINIC-10 Technical Report [4]	5
2.2	Theoretical Background	6
2.2.1	Deep Learning and Artificial Neural Networks (ANNs)	6
2.2.2	Convolutional Neural Networks (CNNs)	6
2.3	Formal context	7
2.3.1	Key Advantages of CNNs	7
2.3.2	Challenges in CNNs	7
2.3.3	Common CNN Architectures	8
2.4	State of the art approaches	8
3	Data Overview	8
4	Different Convolutional Network Architectures	9
4.1	DenseNet121	9
4.1.1	The key advantages of DenseNet-121	10
4.2	Resnet18	10
4.2.1	Key Features of ResNet-18	11
4.3	EfficientNet	11
4.3.1	Key features of EfficientNet	12
4.4	Custom Architecture	12
4.4.1	Feature Extraction Layers	12
4.4.2	Classifier Layers	13
5	Methodology	13
6	Initial Results	13
7	Data Augmentation	16
7.1	Basic	16
7.2	Advanced	17
7.2.1	AutoAugment	17

7.2.2	Cutout	17
7.3	Conclusion and Augmentation Strategy	19
8	Hyperparameters	20
8.1	Training-related hyper-parameters	20
8.1.1	Learning rate	20
8.2	Optimizer	22
8.3	Regularization-related hyper-parameters	22
8.3.1	Dropout	23
8.3.2	Weight decay	24
9	Mixture of Experts	25
9.1	Gating Mechanism	26
9.2	Results	26
10	Theoretical Background of Few-Shot Learning	28
10.1	Meta-Learning: Learning to Learn	28
10.1.1	Metric-Based Meta-Learning	28
10.1.2	Optimization-Based Meta-Learning	29
10.1.3	Model-Based Meta-Learning	29
10.2	Non-Meta-Learning Approaches	30
11	Proposed Few-shot Architecture	30
11.1	Convolutional path	30
11.2	Transformer-based path	30
11.3	Fusion layer	31
11.4	Training process	31
11.5	Model validation	31
11.6	Experiment settings	32
11.7	Results	32
12	Summary	35
13	Reproducibility	36

1 Introduction

Convolutional Neural Networks (CNNs) have been widely used for image recognition. Instead of requiring manually defined features, patterns such as edges, textures, and shapes are learned directly from the pixels. Because of this, CNNs are considered highly effective for image classification tasks.

In this project, a more challenging dataset, CINIC-10, is used. It is created by combining CIFAR-10 images with downsampled samples from ImageNet, resulting in a larger and more diverse dataset. Greater variability in lighting, angles, and backgrounds is introduced, making classification more complex and realistic.

A CNN is designed and trained to classify images from CINIC-10 with high accuracy. Through this process, the performance of CNNs on more complex datasets is examined, and potential challenges are identified.

In this report, the entire process is described, including data preparation, network design, model training, and performance evaluation. Finally, the results are analyzed, their implications are discussed, and possible directions for future work in image classification are considered.

2 Literature review

2.1 Existing approaches

2.1.1 Memory-Replay Knowledge Distillation [20]

In the work authors propose a novel self-knowledge distillation method called **Memory-replay Knowledge Distillation (MrKD)**. This method leverages historical model backups from the training trajectory as teachers to improve the generalization of deep neural networks (DNNs) without requiring external knowledge or model redesign. The key components of MrKD include:

- **Historical Model Backups:** During training, the model periodically saves its parameters, which are then used as teachers to guide the current model. This approach avoids the need for pre-trained teachers or additional model branches.
- **Fully Connected Network (FCN) Ensemble:** The outputs of the historical models are ensembled using a simple FCN to improve the accuracy and diversity of the teacher outputs.
- **Knowledge Adjustment (KA):** The teacher logits are adjusted to ensure that the maximum logit corresponds to the ground truth label, reducing the risk of misleading predictions from imperfect teachers.

The method was evaluated on several datasets, including CINIC-10. The results on CINIC-10 demonstrate that MrKD consistently improves the performance of various DNN architectures, such as ResNet, WideResNet, and ResNeXt. Specifically, MrKD reduced the error rate on CINIC-10 by **0.70% to 1.24%** compared to baseline models. The improvement is attributed to the regularization effect of the historical model backups, which helps stabilize the learning process and reduce overfitting.

Table 1: Test error rate comparison on the CINIC-10 dataset (mean(\pm standard deviation), in % using approaches tested in the article [20])

Model	#Parameter	Baseline	Self-KD	MSD	MrKD
ResNet20	0.3M	17.86 (± 0.21)	16.88 (± 0.23)	16.75 (± 0.09)	16.62 (± 0.08)
ResNet32	0.5M	16.63 (± 0.17)	15.61 (± 0.27)	15.76 (± 0.08)	15.55 (± 0.14)
ResNet56	0.9M	15.44 (± 0.20)	14.75 (± 0.16)	14.91 (± 0.07)	14.74 (± 0.07)
WRN-16-8	11.0M	11.85 (± 0.16)	11.09 (± 0.09)	11.12 (± 0.09)	11.15 (± 0.06)

2.1.2 CINIC-10 Technical Report [4]

In the article, the dataset has been benchmarked with several popular architectures, including **VGG-16**, **ResNet-18**, and **DenseNet-121**. The results show that CINIC-10 presents a more challenging task than CIFAR-10, with higher test error rates across all models. Benchmarks are presented in Table 2.

Table 2: CINIC-10 benchmarks

Model	No. Parameters	Test Error
VGG-16	14.7M	12.23 \pm 0.16
ResNet-18	11.2M	9.73 \pm 0.05
GoogLeNet	6.2M	8.83 \pm 0.12
ResNeXt29_2x64d	9.2M	8.55 \pm 0.15
DenseNet-121	7.0M	8.74 \pm 0.16
MobileNet	3.2M	18.00 \pm 0.16

2.2 Theoretical Background

2.2.1 Deep Learning and Artificial Neural Networks (ANNs)

Deep learning is a subfield of machine learning that focuses on training **Artificial Neural Networks (ANNs)** to learn complex patterns from data. ANNs are computational models inspired by the biological nervous system, consisting of interconnected computational units called **neurons**. These neurons are organized into layers, including an **input layer**, one or more **hidden layers**, and an **output layer**. The network learns by adjusting the weights of connections between neurons through a process called **backpropagation**, which minimizes a loss function.

- **Supervised Learning:** In supervised learning, the network is trained on labeled data, where each input has a corresponding target output. The goal is to minimize the error between the predicted and actual outputs.
- **Unsupervised Learning:** In unsupervised learning, the network learns patterns from unlabeled data, often by optimizing a cost function without explicit target labels.

2.2.2 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are a specialized type of ANN designed for processing grid-like data, such as images. CNNs leverage spatial hierarchies in data by using **convolutional layers**, **pooling layers**, and **fully connected layers**.

- **Convolutional Layers:** These layers apply a set of learnable filters (or kernels) to the input image. Each filter slides over the image, computing a dot product between the filter weights and the local region of the image. This process generates **activation maps** that highlight specific features, such as edges or textures.
 - **Receptive Field:** The region of the input image that a neuron in the convolutional layer "sees" is called the **receptive field**.
 - **Stride and Zero-Padding:** The **stride** determines how much the filter moves across the image, while **zero-padding** adds zeros around the border of the image to control the spatial dimensions of the output.
- **Pooling Layers:** Pooling layers reduce the spatial dimensions of the activation maps, typically using **max-pooling** or **average-pooling**. Max-pooling selects the maximum value from a local region, while average-pooling computes the average. This downsampling reduces computational complexity and helps prevent overfitting [9].

- **Fully Connected Layers:** After several convolutional and pooling layers, the output is flattened and passed through one or more fully connected layers. These layers perform high-level reasoning and produce the final class scores or regression outputs.

2.3 Formal context

In a convolutional neural network, the core operation is the convolution of an input tensor $I \in \mathbb{R}^{H \times W \times C}$ with a set of learnable filters $K \in \mathbb{R}^{k_h \times k_w \times C \times F}$, where H and W are the height and width of the input, C is the number of input channels, k_h and k_w are the filter dimensions, and F is the number of filters. The output feature map $O \in \mathbb{R}^{H' \times W' \times F}$ is computed as $O_{i,j,f} = \sum_{m=0}^{k_h-1} \sum_{n=0}^{k_w-1} \sum_{c=0}^{C-1} I_{i+m,j+n,c} \cdot K_{m,n,c,f} + b_f$, where b_f is a bias term for the f -th filter, and $H' = \lfloor (H - k_h + 2P)/S \rfloor + 1$, $W' = \lfloor (W - k_w + 2P)/S \rfloor + 1$ account for padding P and stride S . This operation is typically followed by a non-linear activation function σ , such as the rectified linear unit $\sigma(x) = \max(0, x)$, applied element-wise to O . The parameters K and b are optimized via backpropagation, minimizing a loss function L over a dataset $D = \{(I^{(i)}, y^{(i)})\}_{i=1}^N$, typically using gradient descent: $K \leftarrow K - \eta \frac{\partial L}{\partial K}$, where η is the learning rate [6, 11].

2.3.1 Key Advantages of CNNs

- **Parameter Sharing:** In CNNs, the same filter is applied across the entire image, significantly reducing the number of parameters compared to fully connected networks.
- **Spatial Hierarchy:** CNNs capture spatial hierarchies by learning local patterns (e.g., edges) in early layers and more complex patterns (e.g., object parts) in deeper layers.
- **Translation Invariance:** CNNs are robust to translations in the input image, meaning they can recognize objects regardless of their position in the image.[12]

2.3.2 Challenges in CNNs

- **Overfitting:** CNNs with many parameters are prone to overfitting, where the model performs well on training data but poorly on unseen data. Techniques like **dropout**, **data augmentation**, and **regularization** are used to mitigate this issue.
- **Computational Complexity:** Training CNNs on high-resolution images can be computationally expensive due to the large number of parameters and activations.

2.3.3 Common CNN Architectures

- **Stacked Convolutional Layers:** It is common to stack multiple convolutional layers before applying a pooling layer. This allows the network to learn more complex features.
- **Small Filters:** Using small filters (e.g., 3×3) with a stride of 1 and zero-padding helps maintain the spatial dimensions of the input while reducing the number of parameters.
- **Fully Connected Layers:** After convolutional and pooling layers, the output is passed through fully connected layers to produce the final predictions.

2.4 State of the art approaches

In the work [8], a novel approach is proposed to improve generalization in image classification tasks by introducing a background class. This method aims to reduce class activation uncertainty without the computational overhead of multitask learning. The background class consists of images that do not belong to any target class but represent common background patterns, helping the model distinguish between relevant objects and irrelevant backgrounds.

Using a **vision transformer (ViT-L/16)** with the proposed background class, the authors achieved **state-of-the-art (SOTA)** performance. The inclusion of the background class significantly improved generalization, leading to higher accuracy compared to traditional transfer learning and multitask learning approaches.

Table 3: Performance comparison of ViT-L/16 with different configurations on CINIC-10.

# Epoch	ViT-L/16	+ Spinal FC	+ Background	+ (Spinal FC & Background)
2	95.11 ± 0.14	95.43 ± 0.19	$95.80 \pm 0.07^*$	$95.80 \pm 0.10^*$

* SOTA Performance on May 2023.

3 Data Overview

The CINIC-10 dataset [4] is an extended alternative to CIFAR-10, designed to address the limitations of existing benchmark datasets. It consists of a total of 270,000 images, which is 4.5 times larger than CIFAR-10. The dataset is constructed by combining images from CIFAR-10 with down-sampled images from the ImageNet dataset, maintaining the same 32×32 resolution as CIFAR-10.

CINIC-10 is structured into three equally sized subsets: training, validation, and test sets. Each subset contains 90,000 images, ensuring a balanced distribution for evaluating model generalization.

The dataset covers the same 10 classes as CIFAR-10: *airplane*, *automobile*, *bird*, *cat*, *deer*, *dog*, *frog*, *horse*, *ship*, and *truck*, with each class having 9,000 images across the three splits.

A significant aspect of CINIC-10 is the inclusion of images from both CIFAR-10 and ImageNet, which introduces a distribution shift. This feature allows researchers to analyze how well models trained on CIFAR-10 generalize to ImageNet-style images. Additionally, the dataset provides a more challenging benchmark compared to CIFAR-10 while being less complex than full-scale ImageNet.

The dataset is publicly available and can be downloaded from the University of Edinburgh’s DataShare repository. It is compatible with popular deep learning frameworks such as PyTorch, enabling easy integration into machine learning pipelines. Some example images can be seen at Figure 1.

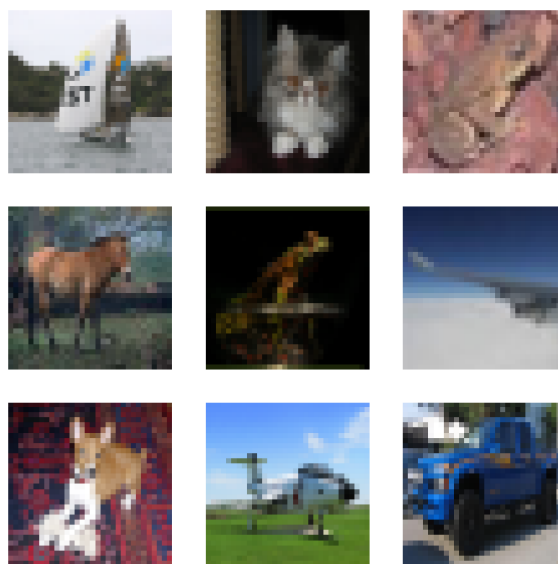


Figure 1: Example images from the CINIC-10 dataset

4 Different Convolutional Network Architectures

This section presents three main approaches to architectures: using pretrained architectures and developing a custom convolutional neural network.

4.1 DenseNet121

The DenseNet-121 architecture is a specific instance of the Dense Convolutional Network (DenseNet) family, which is characterized by its dense connectivity pattern. In DenseNets, each layer is con-

nected to every other layer in a feed-forward manner, ensuring maximum information flow between layers. This connectivity pattern alleviates the vanishing-gradient problem, strengthens feature propagation, encourages feature reuse, and significantly reduces the number of parameters [7].

DenseNet-121 consists of 121 layers, organized into four dense blocks. Each dense block contains multiple layers where each layer receives feature-maps from all preceding layers and passes its own feature-maps to all subsequent layers. The layers between dense blocks are referred to as transition layers, which perform convolution and pooling to downsample the feature-maps.

The initial layers of DenseNet-121 include a 7x7 convolutional layer with stride 2, followed by a 3x3 max-pooling layer with stride 2. This is followed by four dense blocks, each containing a series of composite functions. Each composite function consists of a sequence of operations: Batch Normalization (BN), followed by a Rectified Linear Unit (ReLU) activation, and a 3x3 convolution.

The growth rate k for DenseNet-121 is set to 32, meaning each layer adds 32 feature-maps to the global state of the network. The transition layers between dense blocks include a 1x1 convolutional layer followed by a 2x2 average pooling layer, which reduces the number of feature-maps and downsamples the spatial dimensions.

To improve computational efficiency, DenseNet-121 employs **bottleneck layers**. Before each 3x3 convolution, a 1x1 convolution is introduced to reduce the number of input feature-maps. This bottleneck structure is particularly effective in reducing the computational cost while maintaining the network’s performance.

At the end of the last dense block, a global average pooling layer is applied to reduce the spatial dimensions to 1x1. This is followed by a fully connected layer with a softmax activation function, which outputs the final classification probabilities.

4.1.1 The key advantages of DenseNet-121

- **Parameter Efficiency:** DenseNet-121 requires significantly fewer parameters compared to other architectures like ResNets, while achieving comparable or better performance.
- **Feature Reuse:** The dense connectivity pattern allows for extensive feature reuse, which enhances the network’s ability to learn complex patterns.
- **Improved Gradient Flow:** The direct connections between layers facilitate better gradient flow, making the network easier to train, especially for very deep architectures.

4.2 Resnet18

ResNet-18 is a convolutional neural network (CNN) architecture that is part of the ResNet (Residual Network) family. ResNet-18, as the name suggests, consists of 18 layers, including convolutional

layers, pooling layers, and fully connected layers. The key innovation of ResNet is the introduction of **residual connections** or **skip connections**, which helps in training very deep networks by addressing the vanishing gradient problem.

4.2.1 Key Features of ResNet-18

- **Residual Blocks:** ResNet-18 uses residual blocks, which allow the network to learn residual functions instead of direct mappings. This is achieved by adding skip connections that bypass one or more layers. The output of a residual block is the sum of the input and the output of the convolutional layers within the block. The residual blocks help in training deeper networks by allowing gradients to flow more easily through the network, mitigating the vanishing gradient problem. [1]
- **18 Layers:** ResNet-18 consists of 18 layers, including:
 - An initial convolutional layer.
 - Four stages of residual blocks, each containing two convolutional layers.
 - A global average pooling layer.
 - A fully connected layer for classification.
- **Pre-trained Models:** ResNet-18 is often used with transfer learning, where the model is pre-trained on large datasets like ImageNet. The pre-trained model can then be fine-tuned on a smaller, task-specific dataset, which is particularly useful when the target dataset is small.
- **Efficiency:** Compared to deeper ResNet variants (e.g., ResNet-50, ResNet-101), ResNet-18 is relatively lightweight and computationally efficient, making it suitable for applications where computational resources are limited.

4.3 EfficientNet

EfficientNet is a family of convolutional neural networks (ConvNets) designed to achieve state-of-the-art accuracy while maintaining high efficiency in terms of model size and computational cost. The key innovation of EfficientNet lies in its compound scaling method, which uniformly scales the network’s depth, width, and resolution using a fixed ratio. This approach ensures that the model can effectively capture more complex features while maintaining a balance between accuracy and efficiency.

Traditional ConvNets are typically scaled by increasing only one dimension—depth, width, or resolution. However, EfficientNet introduces a **compound scaling** approach that scales all three

dimensions simultaneously. The scaling is controlled by a compound coefficient ϕ , which determines how much to scale each dimension. The depth, width, and resolution are scaled as follows:

$$\text{depth: } d = \alpha^\phi, \quad \text{width: } w = \beta^\phi, \quad \text{resolution: } r = \gamma^\phi,$$

where α , β , and γ are constants determined through a grid search. This method ensures that the network can handle higher-resolution images by increasing both depth (for larger receptive fields) and width (for capturing finer details)[18].

4.3.1 Key features of EfficientNet

- **Balanced Scaling:** By scaling depth, width, and resolution together, EfficientNet achieves better accuracy and efficiency compared to models that scale only one dimension.
- **State-of-the-Art Performance:** EfficientNet-B7 sets new benchmarks on ImageNet and transfer learning datasets like CIFAR-100 and Flowers, often with significantly fewer parameters.
- **Computational Efficiency:** EfficientNet models are not only smaller but also faster, making them suitable for real-world applications with limited computational resources.

4.4 Custom Architecture

The custom CNN architecture presented here is a general-purpose convolutional neural network designed for image classification tasks. It consists of a series of convolutional layers for feature extraction, followed by fully connected layers for classification. The architecture is divided into two main components: the **feature extraction layers** and the **classifier layers**. The feature extraction part of the network is composed of multiple convolutional blocks, each followed by batch normalization, ReLU activation, and max-pooling. These blocks are designed to progressively extract hierarchical features from the input images:

4.4.1 Feature Extraction Layers

- **Convolutional Layers:** Each block contains one or more convolutional layers with a kernel size of 3x3 and padding of 1. The number of filters increases progressively across the blocks (e.g., 32, 64, 128, 256), allowing the network to capture both low-level and high-level features.
- **Batch Normalization:** After each convolutional layer, batch normalization is applied to stabilize training and improve convergence.
- **ReLU Activation:** ReLU activation is used to introduce non-linearity.

- **Max-Pooling:** Max-pooling with a 2x2 kernel and stride 2 is used to downsample the feature maps, reducing spatial dimensions while retaining important features.

4.4.2 Classifier Layers

After feature extraction, the network flattens the output and passes it through fully connected layers for classification:

- **Flatten Layer:** Converts the 3D feature maps into a 1D vector, preparing the data for the fully connected layers.
- **Fully Connected Layers:** The flattened features are passed through one or more fully connected layers. The first fully connected layer maps the features to a higher-dimensional space (e.g., 512 units), followed by ReLU activation. The final fully connected layer maps the features to the output space, corresponding to the number of classes in the dataset.

5 Methodology

Structured methodology had to be implemented in order to ensure reliable and reproducible results. Each model was trained three times for every verified hyperparameter value and the results were then aggregated using the mean and standard deviation of the final performance metrics. Every passage of training process was set by setting random seed of every aspect where randomness was involved, this includes the order of images passed in the training process, results of operations involving GPU acceleration and weight initialization of the networks themselves.

6 Initial Results

Initial experiments were conducted with several neural network architectures including ResNet18, DenseNet121, EfficientNet, and a Custom CNN, all adapted for the 10-class classification task. The models were trained without freezing any pre-trained weights, with additional fully connected layers added to match the target output dimensionality. All architectures demonstrated strong learning capacity, achieving high training accuracy within the first 10 epochs. The results are presented in Table 4.

Table 4: Initial results for different architectures for 10 epochs

Architecture	Accuracy	Recall	Precision	F1 Score	Loss
Resnet18	0.6775 ± 0.0015	0.6775 ± 0.0015	0.6763 ± 0.0010	0.6752 ± 0.0017	0.9602 ± 0.0293
DenseNet121	0.7197 ± 0.0004	0.7194 ± 0.0004	0.7198 ± 0.0002	0.7181 ± 0.0010	0.8725 ± 0.0111
EfficientNet	0.7262 ± 0.0011	0.7262 ± 0.0009	0.7258 ± 0.0011	0.7249 ± 0.0010	0.8138 ± 0.0232
Custom	0.7329 ± 0.0026	0.7329 ± 0.0026	0.7380 ± 0.0013	0.7335 ± 0.0023	0.8443 ± 0.0425

However, significant overfitting was observed when more epochs of training were used, as evidenced by the growing gap between training and validation accuracy metrics (it is presented in Figure 2). This overfitting phenomenon, visible in the learning curves after 30 epochs, motivated the introduction of data augmentation techniques as a form of regularization to improve model generalization. The augmentation pipeline was designed to artificially expand the training dataset while preserving label integrity, thereby helping the models learn more robust features without simply memorizing the training examples.

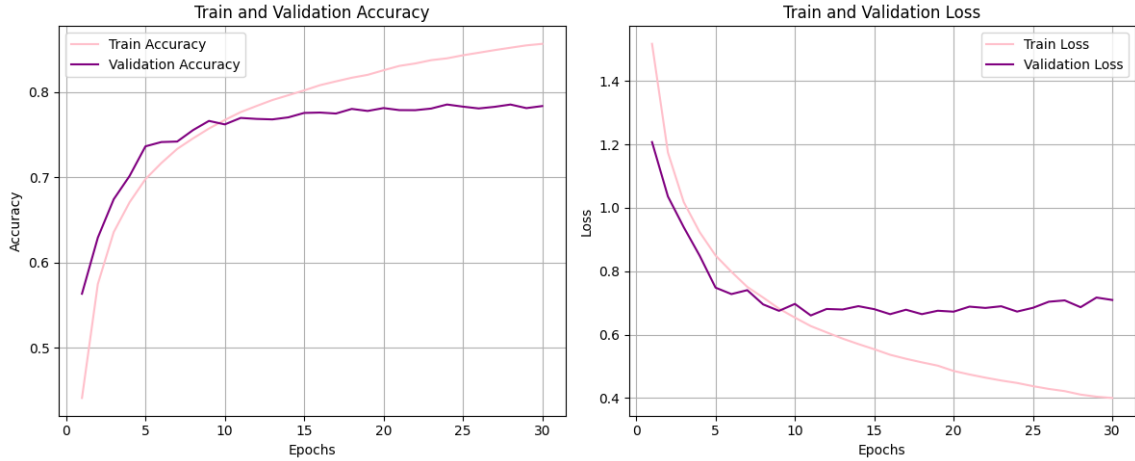


Figure 2: Accuracy and Loss change through 30 epochs without regularization using model with Custom Architecture

The initial model performance was further investigated through examination of the confusion matrix (Figure 3). The analysis reveals specific patterns in misclassification, particularly among four key classes:

- **Cats and dogs** show mutual confusion, suggesting the model struggles to distinguish between these animal categories

- **Automobiles and trucks** are frequently misclassified for each other, indicating difficulty in differentiating these vehicle types

This pairwise confusion pattern suggests that:

- The model may be relying on similar visual features for these class pairs
- The distinguishing characteristics between these categories may not be sufficiently emphasized in the training data
- Additional domain-specific augmentation could help improve differentiation

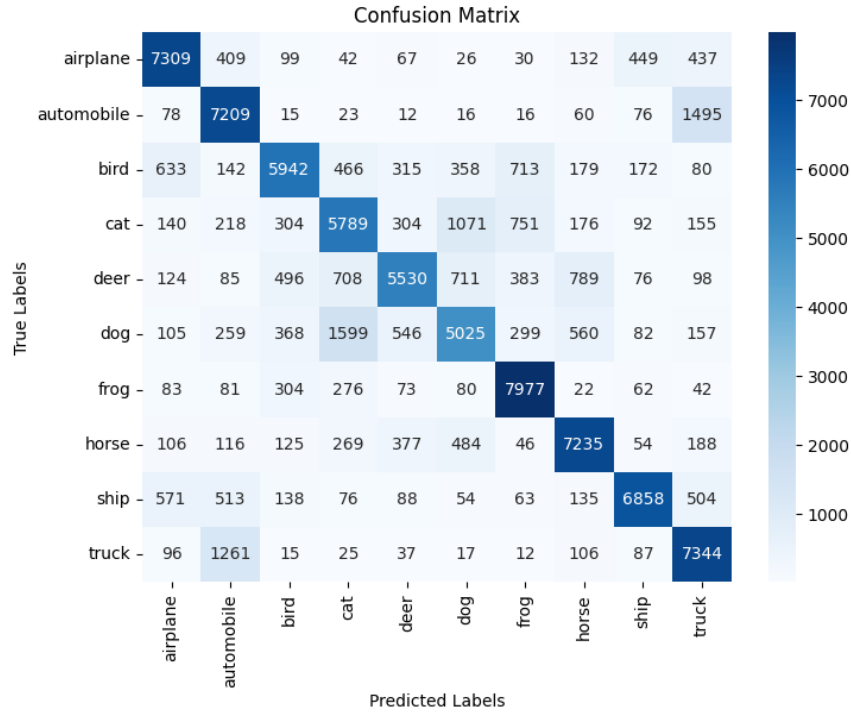


Figure 3: Confusion matrix of the Custom architecture after initial training of the model

These observations provide valuable insights for potential model improvements, particularly in feature learning for the problematic class pairs.

7 Data Augmentation

Data augmentation is a technique used to artificially increase the size and diversity of a training dataset by applying various transformations to the existing data. In the context of image data, these transformations can include rotations, translations, scaling, flipping, cropping, and changes in brightness or contrast, among others. The primary goal of data augmentation is to improve the generalization ability of machine learning models, particularly deep neural networks, by exposing them to a broader range of variations in the data.

7.1 Basic

Basic data augmentation techniques and their parameters which influence were examined in this project include:

- **Rotation:** Rotating the image by a certain angle (`degrees=15`)
- **Random crop:** Randomly cropping the image (`size=32, scale=(0.8,1.0)`)
- **Horizontal flipping:** Mirroring the image horizontally
- **Color jitter:** Randomly altering the color properties of the image, such as hue, saturation, and brightness, to simulate different lighting conditions or camera settings (`brightness=0.4, contrast=0.4, saturation=0.4, hue=0.1`)

The results of using these approaches are presented in the Table 5.

Table 5: Mean accuracy of different architectures using basic augmentations technics (10 epochs)

Architecture	Horizontal flip	Rotation	Random crop	Color jitter
Resnet18	0.6970 \pm 0.0030	0.6905 \pm 0.0015	0.6848 \pm 0.0028	0.6690 \pm 0.0025
DenseNet121	0.6548 \pm 0.0312	0.7222 \pm 0.0026	0.7197 \pm 0.0043	0.7151 \pm 0.0033
EfficientNet	0.7374 \pm 0.0121	0.7284 \pm 0.0008	0.7222 \pm 0.0039	0.1711 \pm 0.0427
Custom	0.7565 \pm 0.0022	0.7405 \pm 0.0013	0.7361 \pm 0.0015	0.6114 \pm 0.0019

The performance comparison of different architectures with basic augmentation techniques reveals several important insights. The custom architecture achieved the highest mean accuracy across most augmentation methods, particularly excelling with horizontal flip (0.7565 \pm 0.0022), suggesting its superior ability to leverage simple geometric transformations. However, color jitter produced mixed results - while DenseNet121 maintained strong performance (0.7151 \pm 0.0033),

EfficientNet suffered a dramatic accuracy drop to 0.1711 ± 0.0427 , potentially indicating sensitivity to color variations. The custom network demonstrated the most robust performance overall, with the smallest variations between different augmentation techniques, highlighting its balanced design for this classification task. These results suggest that simpler geometric transformations (flips and rotations) generally provide more reliable performance gains compared to color-based augmentations for these architectures.

7.2 Advanced

While traditional data augmentation is effective, advanced methods such as **AutoAugment** and **Cutout** have been developed to further enhance the performance of deep learning models.

7.2.1 AutoAugment

AutoAugment is an advanced data augmentation technique that uses reinforcement learning to automatically discover the best augmentation policies for a given dataset. Instead of manually selecting and tuning augmentation strategies, AutoAugment learns a set of transformations that maximize model performance on a validation set [3].

1. **Search Space:** AutoAugment defines a search space of possible augmentation operations, such as rotation, translation, shearing, and color adjustments.
2. **Reinforcement Learning:** A controller neural network is trained using reinforcement learning to select sequences of augmentation operations that improve model performance.
3. **Policy Optimization:** The controller explores different combinations of augmentations and evaluates their impact on model accuracy. The best-performing policies are then used to augment the training data.

7.2.2 Cutout

Cutout is another advanced data augmentation technique that randomly masks out square regions of an image during training. This forces the model to focus on other parts of the image, improving its ability to generalize and reducing overfitting.

- **Random Masking:** During training, Cutout randomly selects square regions of the image and sets their pixel values to zero (or another constant value).
- **Mask Size:** The size of the masked region is a hyperparameter that can be adjusted based on the dataset and task.

- **Training Impact:** By occluding parts of the image, Cutout encourages the model to learn more robust features that are not dependent on specific regions of the input.

The parameters that were used are: $p=0.5$, $scale=(0.02, 0.2)$, $ratio=(0.3, 3.3)$, $value=0$.

Table 6: Base image sampled from the dataset and its augmented versions.



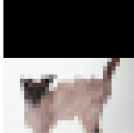







Base Image	Augmentations		
			
			
			
	Cutout	AutoAugment	Color Jitter

Table 7: Mean accuracy of different architectures using advanced augmentations technics (10 epochs)

Architecture	AutoAugment	Cutout
Resnet18	0.6771 ± 0.0013	0.6827 ± 0.0055
DenseNet121	0.6858 ± 0.0118	0.7226 ± 0.0022
EfficientNet	0.7160 ± 0.0025	0.7252 ± 0.0001
Custom	0.7261 ± 0.0051	0.7368 ± 0.0021

The evaluation of advanced augmentation methods (presented in the Table 7) demonstrates dis-

tinct patterns in architectural performance. The custom network again shows superior robustness, achieving the highest accuracy with both AutoAugment and Cutout, while maintaining relatively low variance. Cutout proves particularly effective across all architectures. The results reveal that DenseNet121 and EfficientNet benefit more significantly from advanced augmentations compared to ResNet18, suggesting their architectures are better suited to leverage complex transformation policies. Notably, AutoAugment shows more variable performance (standard deviations up to 0.0118), potentially due to its policy search space being less optimized for certain architectures. The consistent advantage of Cutout across all models highlights its effectiveness as a general-purpose regularization technique.

7.3 Conclusion and Augmentation Strategy

Based on the comprehensive analysis of both basic and advanced augmentation techniques, a hybrid augmentation strategy was adopted for the final implementation. For the training dataset, each image is processed in two parallel streams: the first version undergoes basic augmentations (horizontal flip, rotation, and random crop), while the second version receives both these basic transformations and one complex augmentation (either AutoAugment or Cutout, only the latter one was considered in further experiments). This dual-stream approach combines the reliability of simple geometric transformations with the regularization benefits of more sophisticated augmentation methods. The validation and testing datasets are processed using only the standardized test transform to ensure fair evaluation.

The custom architecture was chosen as the base model due to its consistent performance across all augmentation types, with particular emphasis on its strong results with Cutout augmentation. This balanced approach aims to maximize model generalization while minimizing the risk of overfitting observed in the initial experiments and their results proving the improvement over regular augmentations approaches are shown in Table 8.

Table 8: Mean accuracy of different architectures using hybrid augmentations technique (10 epochs)

Architecture	Hybrid Approach Accuracy
Resnet18	0.7063±0.0012
DenseNet121	0.6992±0.0103
EfficientNet	0.7321±0.0037
Custom	0.7647±0.0051

8 Hyperparameters

Hyperparameters play a critical role in determining the performance of a machine learning model. These external configurations, which are not learned during training, significantly influence the learning process. In this section, the hyperparameters used in the project are discussed.

8.1 Training-related hyper-parameters

Training-related hyperparameters are crucial for controlling the learning process and ensuring that the model converges to an optimal solution. These include the learning rate, number of epochs, and the choice of optimizer. Each of these parameters affects the speed, stability, and final performance of the model.

8.1.1 Learning rate

The learning rate determines the size of the steps taken during gradient descent to update the model's weights. A learning rate that is too high can cause the model to overshoot the optimal solution, while a learning rate that is too low can result in slow convergence or getting stuck in local minima. In the project, learning rates of 0.001, 0.0005, and 0.002 were examined. The results for each value of learning rate are presented in the Table 9.

Table 9: Different values of learning rates and the mean accuracy for each architecture

Architecture	Accuracy for lr=0.002	Accuracy for lr=0.001	Accuracy for lr=0.0005
DenseNet121	0.6125 \pm 0.0081	0.6136 \pm 0.0027	0.6018 \pm 0.0036
ResNet18	0.6775 \pm 0.0038	0.7101 \pm 0.0014	0.7097 \pm 0.0016
EfficientNet	0.7110 \pm 0.0064	0.7277 \pm 0.0024	0.7367 \pm 0.0024
Custom	0.7557 \pm 0.0035	0.7643 \pm 0.0032	0.7609 \pm 0.0043

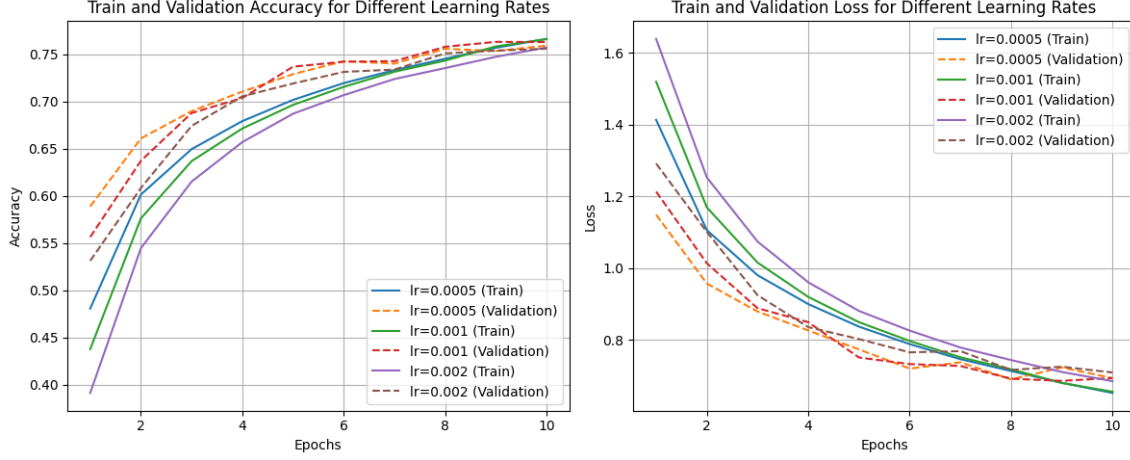


Figure 4: Different values of learning rate for Custom Architecture

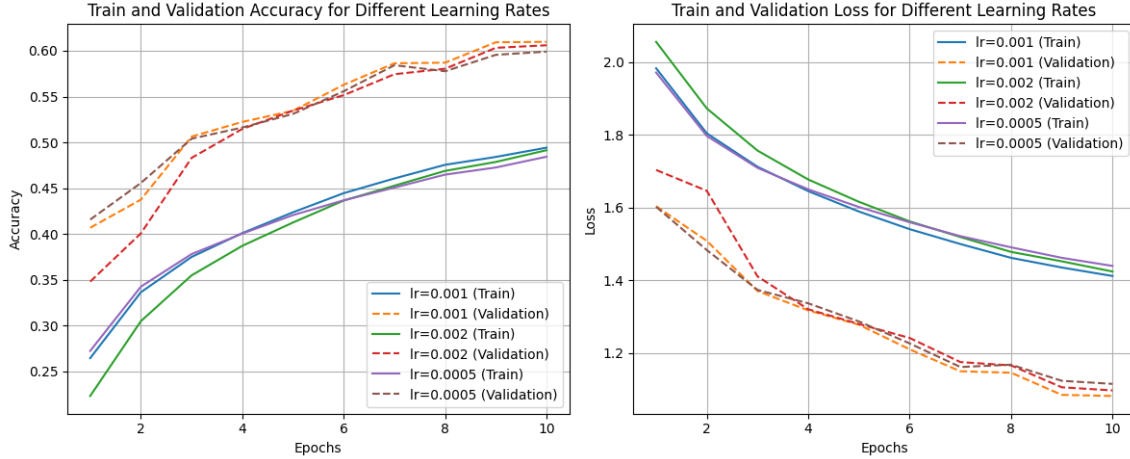


Figure 5: Different values of learning rate for DenseNet121 Architecture

It can be observed (at 4 and 5 Figures) that a learning rate of 0.001 provided the best balance between convergence speed and model accuracy, while 0.0005 led to slower training and 0.002 occasionally caused instability in the training process. Overall, however, the impact of the value of the learning rate is not as significant as one may expect. This may be due to the models already being in a region of the loss landscape where optimization is relatively stable, meaning small adjustments to the learning rate do not significantly affect convergence. A second reason may be that the dataset is already well-conditioned (due to augmentations), or the architectures may

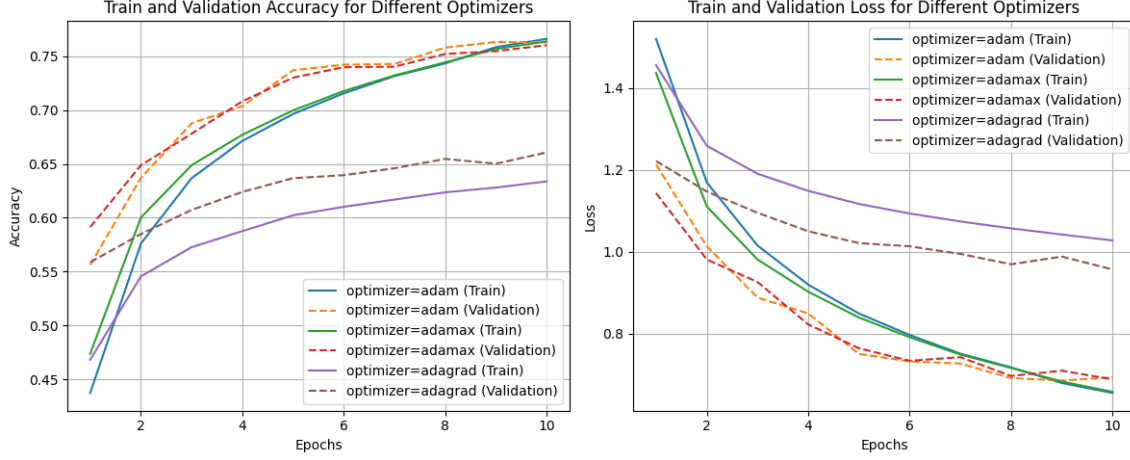


Figure 6: Different optimizers choices for Custom Architecture and their results

already generalize well, limiting the impact of learning rate changes.

8.2 Optimizer

The optimizer is responsible for updating the model’s weights based on the computed gradients. Different optimizers have unique characteristics that can affect the training dynamics and final performance. In the project, three popular optimizers were experimented with: **Adam**, **Adagrad**, and **Adamax**.

Figure 6 shows that the Adam optimizer demonstrated strong performance, achieving high training and validation accuracy with a relatively small gap between the two, indicating good generalization. The Adamax optimizer also performed well, with slightly lower training accuracy but comparable validation accuracy, suggesting it is robust against overfitting. The Adagrad optimizer, not only achieving low training accuracy, showed a larger gap between training and validation accuracy, indicating potential overfitting or less effective generalization compared to Adam and Adamax.

Results for different optimizer choices for Custom Architecture are presented in the Table 10.

8.3 Regularization-related hyper-parameters

Even after 30 epochs, phenomenon of overfitting can be observed, which can be seen in Figure 7. To mitigate overfitting and improve the generalization ability of the model, regularization techniques such as dropout and weight decay were employed. These techniques help in reducing the model’s

Table 10: Results of using different optimizers in Custom Architecture described in 4.4

Optimizer	Accuracy	Recall	Precision	F1 Score	Loss
Adam	0.7643 \pm 0.0032	0.7643 \pm 0.0032	0.7666 \pm 0.0019	0.7629 \pm 0.0028	0.6810 \pm 0.0102
Adagrad	0.6534 \pm 0.0033	0.6534 \pm 0.0033	0.6484 \pm 0.0031	0.6469 \pm 0.0039	0.9785 \pm 0.0080
Adamax	0.7593 \pm 0.0023	0.7593 \pm 0.0023	0.7594 \pm 0.0016	0.7567 \pm 0.0019	0.6927 \pm 0.0036

reliance on specific features and encourage it to learn more robust patterns.

8.3.1 Dropout

Dropout is a regularization technique that randomly deactivates a fraction of neurons during training, preventing the model from becoming overly dependent on specific neurons. This randomness forces the network to learn more distributed representations, which improves generalization. In the project, dropout was applied after each feature extraction block and in the classifier layers in the model 4.4. A dropout rate of 0.4 in the classification part and 0.2 in the feature extraction was found to be most effective in reducing overfitting while maintaining model performance. Figure 7 demonstrates that the model’s results converge more rapidly when dropout methods are applied.

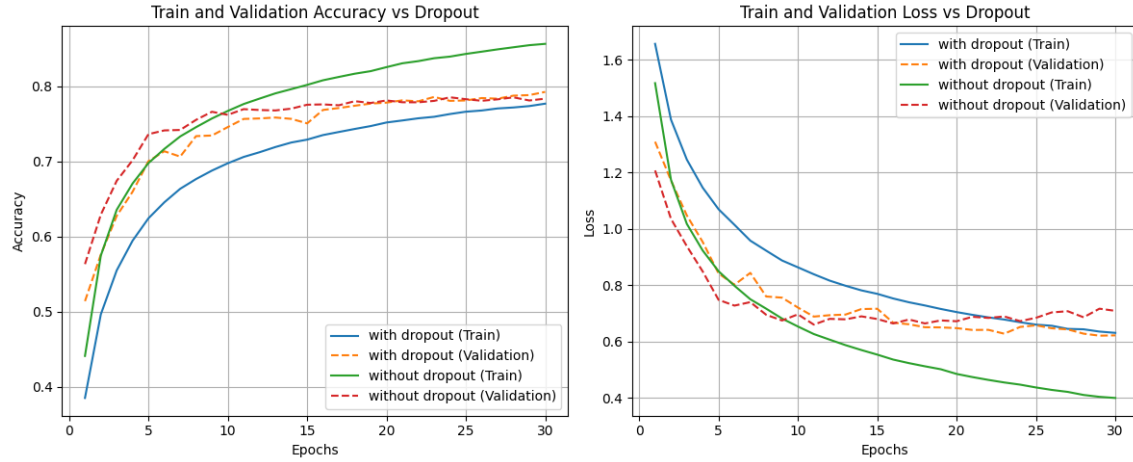


Figure 7: Accuracy and Loss through epochs with and without using Dropout for Custom Architecture

The overall approach was to apply a factor of dropout $k \in (0, 0.5)$ such that the dropout rate of the classifier layer would be $2k$ and the within-network dropout rate would be exactly k . The results of this approach are presented in Table 11.

Table 11: Results of the Custom architecture for different dropout coefficients k

k	Accuracy	Precision
0.1	0.7356 \pm 0.0022	0.7409 \pm 0.0031
0.2	0.7935 \pm 0.0114	0.8003 \pm 0.0132
0.3	0.7411 \pm 0.0314	0.7322 \pm 0.0401
0.4	0.7031 \pm 0.0218	0.6909 \pm 0.0146

The variation in accuracy values across different dropout coefficients k can be attributed to how dropout influences the learning dynamics of the model. At $k = 0.1$, the dropout is minimal, meaning the network retains most of its neurons during training. While this allows the model to learn more complex patterns, it also increases the risk of overfitting, as evidenced by the relatively modest accuracy of 0.7356. When $k = 0.2$, the model appears to achieve the best balance—sufficient regularization to prevent overfitting without excessively hindering the model’s capacity. This results in the highest accuracy of 0.7935. As k increases to 0.3 and 0.4, the accuracy starts to decline significantly. At $k = 0.3$, accuracy drops to 0.7411, and at $k = 0.4$, it falls further to 0.7031. These decreases are likely due to underfitting caused by too much dropout, where the network is unable to capture the necessary complexity of the data because too many neurons are being randomly deactivated during training. Thus, both extremes—too little or too much dropout—impair model performance, emphasizing the importance of tuning k to find the optimal regularization level.

8.3.2 Weight decay

Weight decay, also known as L2 regularization, adds a penalty to the loss function based on the magnitude of the model’s weights. This discourages the model from learning large weights, which can lead to overfitting. In the project, weight decay was applied with values of 0.0001, 0.0002 and 0.00005. The results are presented in the Table 12.

Table 12: Mean accuracy obtained after using different weight decay parameter values for each architecture

Architecture	wd = 0.0001	wd = 0.0002	wd = 0.00005
Resnet18	0.6730 \pm 0.0020	0.6673 \pm 0.0034	0.6753 \pm 0.0028
DenseNet121	0.6667 \pm 0.0016	0.6604 \pm 0.0026	0.6656 \pm 0.0052
EfficientNet	0.7210 \pm 0.0018	0.7148 \pm 0.0020	0.7230 \pm 0.0011
Custom	0.7290 \pm 0.0031	0.7300 \pm 0.0081	0.7301 \pm 0.0010

As can be seen at Figure 8 the model with a weight decay of 0.00005 exhibited the highest training accuracy but showed a significant gap between training and validation accuracy, indicating overfitting. In contrast, the model with a weight decay of 0.0001 demonstrated a more balanced performance, with closer alignment between training and validation accuracy, suggesting better generalization. The accuracy curve for the model with a weight decay of 0.0002 was less smooth compared to the others, suggesting more variability in its learning process.

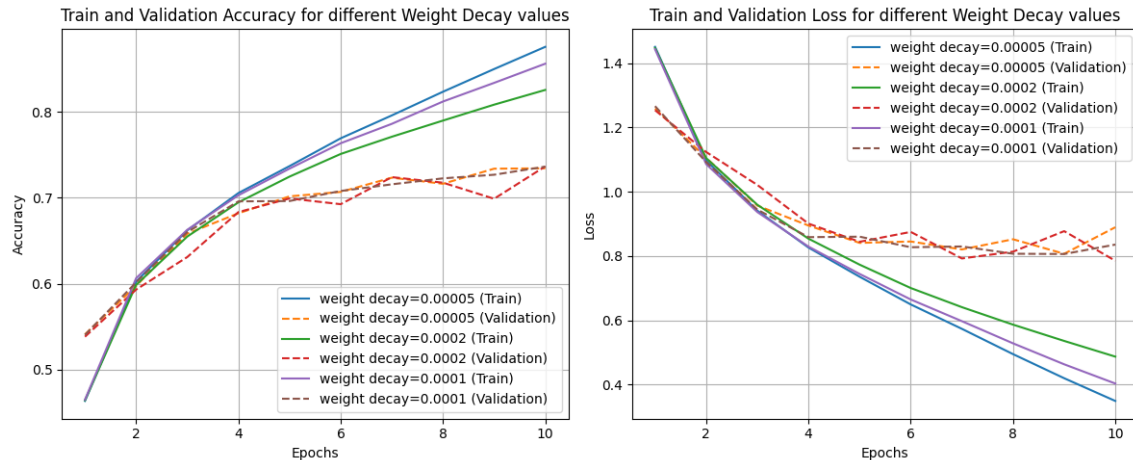


Figure 8: Accuracy and Loss through epochs for different weight decay values for Custom Architecture

9 Mixture of Experts

To improve classification performance, a Mixture of Experts (MoE) approach was explored using convolutional neural networks. This architecture leverages multiple convolutional models as expert networks, with a gating mechanism dynamically assigning input samples to the most relevant expert. By optimizing expert selection, the MoE structure enhances decision-making and overall model accuracy. Various configurations were tested to determine the most effective combination for classification improvement.

The implemented MoE architecture integrates three distinct convolutional networks as expert models:

- **Custom CNN:** The best-performing architecture from previous experiments.
- **Vehicles Recognition Expert:** Model that was trained specifically to distinguish vehicles.

- **Animal Recognition Expert:** A specialized network trained specifically to distinguish most problematic animal classes including: *cat, dog, deer and horse* (as identified in the confusion matrix, Figure 3).

To maintain learned knowledge, the weights of all expert networks were frozen. The overall architecture consists of:

- Three expert networks with adjusted output dimensions.
- A dynamic gating mechanism.
- A final classification layer.

9.1 Gating Mechanism

The gating network is responsible for selecting the most suitable expert for each input sample. It processes feature representations from all experts and assigns a probability distribution over them. The network consists of:

- Multiple fully connected layers with batch normalization.
- ReLU activation functions for non-linearity.
- Dropout layers ($p = 0.3$) for regularization.
- A final softmax layer producing expert selection probabilities.

The gating mechanism takes the averaged feature representations from all experts as input, ensuring adaptive and efficient model selection, ultimately improving classification performance.

9.2 Results

Finally, for the test dataset, the results are:

Loss:0.6208, Accuracy:0.8034, F1 Score:0.8036, Precision:0.8040, Recall:0.8034

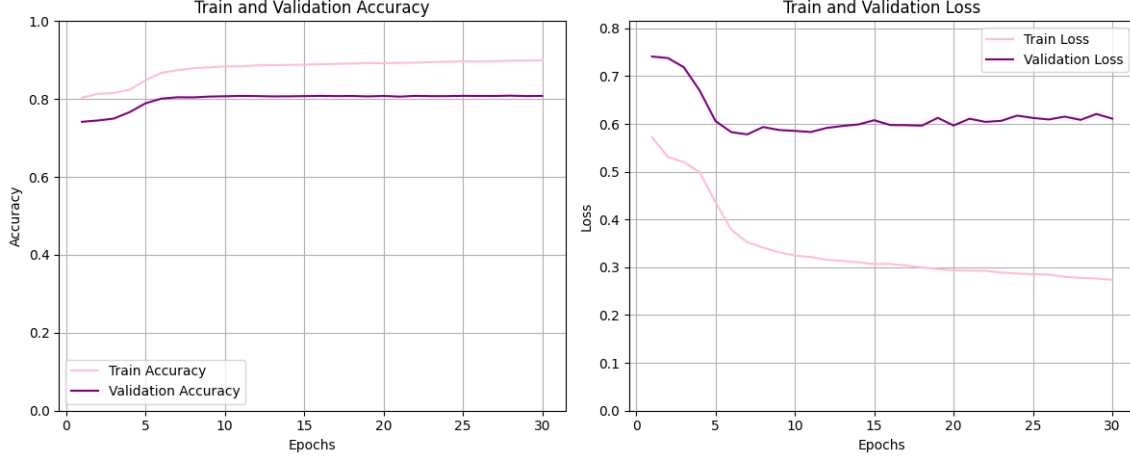


Figure 9: Accuracy and Loss through epochs for Mixture of Experts architecture

The learning curves shown in Figure 9 demonstrate clear signs of overfitting, with a noticeable divergence emerging between training and validation accuracy alongside persistent validation loss despite decreasing training loss. This behavior suggests the model’s capacity may exceed the complexity of the learning task or that additional regularization could improve generalization.

Several promising directions for boosting performance have been identified, including exploring more aggressive regularization techniques such as increased dropout rates, stronger weight decay, or label smoothing. Architectural modifications could also prove beneficial, particularly alternative convolutional block designs, refined gating mechanisms, or additional batch normalization layers. The model’s current performance plateau in later training epochs further indicates potential for improvement through learning rate scheduling adjustments.

While these optimization opportunities present clear paths for enhancing model performance, practical constraints of the current study necessitated prioritizing the establishment of a robust baseline implementation. The observed patterns of overfitting provide valuable empirical evidence to guide future investigations into more sophisticated regularization approaches and architectural refinements. This work establishes a foundation for subsequent research while clearly delineating specific areas where performance gains may be achieved through focused improvements to the model’s generalization capabilities.

10 Theoretical Background of Few-Shot Learning

Few-Shot Learning (FSL) addresses the challenge of learning robust patterns from a limited number of training examples, a scenario where traditional deep learning models, which rely on large datasets, often falter due to data scarcity and computational demands. As highlighted by Parnami and Lee, FSL aims to emulate human-like learning capabilities, where generalization occurs rapidly from minimal examples, such as recognizing a person from a few photographs [14]. The theoretical foundation of FSL is rooted in two primary paradigms: meta-learning (learning to learn) and transfer learning, with hybrid approaches further enriching the field. This section explores these foundations, focusing on meta-learning’s dominant role, and provides examples of architectures for each approach.

10.1 Meta-Learning: Learning to Learn

Meta-learning, or “learning to learn,” is a cornerstone of FSL, enabling models to leverage prior experiences across multiple tasks to adapt quickly to new, sparsely sampled tasks [16, 19]. Unlike traditional supervised learning, which optimizes a single task using a dataset $\mathcal{D} = \{(\mathbf{x}_k, y_k)\}_{k=1}^n$ split into training $\mathcal{D}^{\text{train}}$ and testing $\mathcal{D}^{\text{test}}$ sets, meta-learning operates over a distribution of tasks $p(\mathcal{T})$. A meta-learner is trained on a set of tasks $\mathcal{T}_{\text{meta-train}} = \{\mathcal{T}_1, \mathcal{T}_2, \dots, \mathcal{T}_n\}$ with datasets $\mathcal{D}_{\text{meta-train}} = \{\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_n\}$, and evaluated on unseen tasks $\mathcal{T}_{\text{meta-test}}$. The goal is to optimize parameters θ such that for a new task $\mathcal{T}_i \sim p(\mathcal{T})$, the model generalizes well on $\mathcal{D}_i^{\text{test}}$ given $\mathcal{D}_i^{\text{train}}$, formalized as:

$$\theta = \arg \min_{\theta} \sum_{\mathcal{D}_i \in \mathcal{D}_{\text{meta-train}}} \sum_{(\mathbf{x}, y) \in \mathcal{D}_i^{\text{test}}} \mathcal{L}(f(\mathcal{D}_i^{\text{train}}, \mathbf{x}; \theta), y),$$

where \mathcal{L} is a loss function, and f is the learned model [14].

Meta-learning in FSL is categorized into three main approaches: metric-based, optimization-based, and model-based, each differing in how they model the output probability $P_{\theta}(y \mid \mathbf{x})$ for a query sample \mathbf{x} given a support set S .

10.1.1 Metric-Based Meta-Learning

Metric-based meta-learning focuses on learning a similarity metric in an embedding space to classify queries based on proximity to support samples. The theoretical underpinning is metric learning, where an embedding function $g_{\theta_1} : \mathbb{R}^n \rightarrow \mathbb{R}^m$ (with $n > m$) maps high-dimensional inputs to a lower-dimensional space, and a distance function d_{θ_2} measures similarity [10]. The output probability is modeled as:

$$P_{\theta}(y \mid \mathbf{x}) = \sum_{(\mathbf{x}_k, y_k) \in S} k_{\theta}(\mathbf{x}, \mathbf{x}_k) y_k,$$

where k_θ is a kernel function (e.g., cosine similarity) [14].

Example Architecture: Prototypical Networks [17] use a 4-layer convolutional neural network (CNN) as g_{θ_1} to embed support and query images. For a class c , a prototype \mathbf{v}_c is computed as the mean of embedded support samples:

$$\mathbf{v}_c = \frac{1}{|S^c|} \sum_{(\mathbf{x}_k, y_k) \in S^c} g_{\theta_1}(\mathbf{x}_k).$$

The query \mathbf{x} is classified by a softmax over Euclidean distances to prototypes:

$$P(y = c \mid \mathbf{x}) = \frac{\exp(-d(g_{\theta_1}(\mathbf{x}), \mathbf{v}_c))}{\sum_{c' \in C} \exp(-d(g_{\theta_1}(\mathbf{x}), \mathbf{v}_{c'}))}.$$

Training minimizes the negative log-likelihood of the true class, making it computationally efficient and effective for tasks like 5-way 1-shot classification on miniImageNet.

10.1.2 Optimization-Based Meta-Learning

Optimization-based meta-learning aims to learn an initialization or optimization strategy that enables rapid adaptation to new tasks using gradient descent. The theoretical basis is to find a parameter set θ that, after a few gradient updates on $\mathcal{D}_i^{\text{train}}$, performs well on $\mathcal{D}_i^{\text{test}}$. The probability is modeled as $P_{\theta'}(y \mid \mathbf{x})$, where $\theta' = g_\phi(\theta, S)$ is adapted from an initial θ using a meta-learner g_ϕ [14].

Example Architecture: MAML (Model-Agnostic Meta-Learning) [5] initializes a neural network (e.g., a CNN) with parameters θ . For a task \mathcal{T}_i , an inner loop updates θ to θ'_i using a few gradient steps on $\mathcal{D}_i^{\text{train}}$:

$$\theta'_i = \theta - \alpha \nabla_{\theta} \mathcal{L}_{\mathcal{D}_i^{\text{train}}}(f_{\theta}),$$

where α is the learning rate. An outer loop then optimizes θ across tasks by minimizing the loss on $\mathcal{D}_i^{\text{test}}$ using θ'_i . This nested optimization enables MAML to adapt to diverse tasks, though it requires higher computational resources due to second-order derivatives.

10.1.3 Model-Based Meta-Learning

Model-based meta-learning integrates architectural components, such as memory, to store and retrieve task-specific information rapidly. The probability is modeled as $f_{\theta}(\mathbf{x}, S)$, where the model directly incorporates the support set S [14]. This approach draws from memory-augmented neural networks and rapid adaptation techniques.

Example Architecture: Memory Augmented Neural Networks (MANN) [15] extend Neural Turing Machines (NTMs) with a controller (e.g., LSTM) and an external memory matrix

M_t . For an input \mathbf{x}_t , a key vector k_t is generated and used to read from memory via soft attention:

$$r_t = \sum_i w_t^r(i) M_t(i), \quad w_t^r(i) = \text{softmax} \left(\frac{k_t \cdot M_t(i)}{\|k_t\| \|M_t(i)\|} \right).$$

During training, MANN processes sequences with offset labels (e.g., (\mathbf{x}_t, y_{t-1})), forcing it to store sample-class bindings in memory for prediction, ideal for one-shot learning scenarios like Omniglot.

10.2 Non-Meta-Learning Approaches

Beyond meta-learning, transfer learning provides a simpler theoretical framework for FSL by leveraging knowledge from a pre-trained model on a large dataset (source task) and fine-tuning it on a few-shot task (target task) [13]. The challenge lies in avoiding overfitting with limited data.

Example Architecture: SimpleShot [21] uses a pre-trained deep CNN (e.g., ResNet) to extract embeddings from base classes. For a new task, it applies centering and L2 normalization to these embeddings, then performs nearest-neighbor classification with Euclidean distance. This straightforward approach achieves competitive accuracy (e.g., 64.29% on miniImageNet 5-way 1-shot) without meta-training complexity.

11 Proposed Few-shot Architecture

architecture created for the project is a hybrid model tailored for few-shot learning, combining the strengths of convolutional neural networks (CNNs) and transformers. The model processes input images through two parallel branches—a **convolutional path** and a **transformer-based path**. The CNN branch captures local patterns such as edges and textures, while the transformer branch captures the global context via self-attention mechanisms.

11.1 Convolutional path

To reduce computational overhead, the CNN branch uses depthwise separable convolutions [2], which significantly cut down the number of parameters and operations without sacrificing representational capacity.

11.2 Transformer-based path

Instead of processing an entire image as a sequence, the transformer branch embeds non-overlapping patches (of an arbitrary size, set by default to 2×2), reducing sequence length and computational complexity. Following this approach, an input image consisting of 32^2 pixels would be converted into $(32/2)^2 = \frac{1}{4}32^2$ patches. This balances global reasoning with computational efficiency.

11.3 Fusion layer

The outputs of both branches are merged in a Fusion Layer that learns to combine the complementary strengths of local and global features. It is done through concatenation of the derived features and treating them as an input of a MLP consisting of one dense hidden layer. This results in rich, discriminative embeddings suitable for a variety of classification tasks.

11.4 Training process

Though the architecture itself might be considered in standard image classification, however, due to the data scarcity, the n-way-k-shot method combined with the concept of prototypical networks approach was implemented. That way, during training, the model learns to map images into an embedding space where class prototypes (mean embeddings of support examples) are used for classification via Euclidean distance metric. Cross-Entropy Loss between the logits (negative squared Euclidean distances) and the ground-truth query labels was the task the architecture was trained to minimize.

11.5 Model validation

Even though the trained model requires a support set for its validation on unknown examples (which would be an impossible scenario for class prediction on future images). In order to provide a simulation of such scenario, the entire training set is used for creating a class prototype - and embedded representation of each class. Two most intuitive approaches would be feasible:

1. Every image of a given class would be considered in the support set for this class, requiring only 10 distance computations for each test image embedding,
2. Every image would represent a singleton support set, requiring $|D_{train}|$ distance computations for a test image embedding.

With both of the above being on the rear ends of the regularization spectrum. A hybrid approach was instead chosen: embedding of each training image was considered and the set of embeddings E_c for a given class c was split into k clusters $\{E_{c_1}, \dots, E_{c_k}\}$ using k -Means algorithm. Each cluster was then considered to be its own support set for this class with a mean prototype embedding $e(E_{c_j})$. Classification of a new image could therefore be formalized as:

$$C(\text{Image}) = \underset{c}{\operatorname{argmin}} \{ \min_k \|e(\text{Image}) - e(E_{c_k})\|^2 \}$$

11.6 Experiment settings

Each model was trained using the n -way- k -shot framework with $n = k = 5$, whereas the size q of the query set for each training tasks varied, depending on the size of the training dataset (see Table 13). Each class in the shrunk dataset had approximately the same number of representatives - class imbalance would further hinder the classification task.

Table 13: Size of the query dataset for different fractions p of training data used

p	q
0.001	2
0.002	5
0.005	5
0.01	10
0.02	10
0.05	15
0.1	20

The training of each model took 3000 episodes, each on a randomly sampled subset of the original training data. The decision which 5 out of 10 classes were used for each training task was also randomized. Data augmentation was implemented only to the query set of the data in order to make embeddings closely related to ground truth. Horizontal Flip alongside AutoAugment was used for query set data diversification. After each model’s training, the whole fraction p of the dataset was used for determining the prototypes as mentioned in section 11.5. The number of convolutional neural network channels was set to 32, embedding space dimension to 64, and the number of attention heads for the transformer branch was set to 2. Number of clusters for prototypes was fixed to 3. Further architecture parameter tuning was out of scope of this project and is left as a research opportunities in the future.

11.7 Results

Primary observation that favors this architecture compared to the standard ones is its lightweight structure. The entire model saved as a raw file of `.pth` extension takes up only 199kB of disc space, compared to > 40 MB for the architectures mentioned in earlier sections. Figure 10 illustrates the training loss over episodes for various values of the fraction of data used per episode, denoted as p , in a few-shot learning setup. Notably, it reveals a consistent trend: higher values of p lead to higher training loss throughout the episodic training process. This is contrary to the trend of all

different metric values achieved by the model on the test dataset, denoted in Table 14. One possible explanation is that with a higher p , the model is exposed to more data per episode, allowing it to learn more diverse or representative features - even if the optimization process (as reflected by the loss) progresses more slowly or plateaus at a higher value. In this case, the model may not be minimizing the loss as aggressively, but it is learning features that generalize better to unseen examples, thus resulting in higher accuracy. Another possible reason is that with smaller p , the model may overfit to the very limited data in each episode, leading to lower training loss but poorer generalization, and hence lower accuracy. Essentially, the model gets too good at fitting the small batch, but fails to learn patterns that generalize. Conversely, a larger p provides richer training signals and smoother gradients, which may make the model more robust, even if it sacrifices some optimization efficiency in the process. An obvious comparison to the training process on the entirety of training dataset would be the model performance on test dataset. Even for the highest values of considered p , that basically breach the few-shot assumption of low cardinality of training dataset, both approaches do not achieve nearly as good results. Table 15 shows the results on test dataset of the custom architecture from sec. 4.4 trained on a given fraction of the data p . The few-shot episodic model consistently achieves higher accuracy than the regular CNN across nearly all values of p . This trend persists, with the episodic approach staying above 0.2, while the CNN only surpasses that mark starting around $p = 0.01$. On top of that, for $p = 0.001$, the model achieved a near-randomized classification performance, indicating complete lack of pattern recognition learned throughout the process. Both architectures (custom few-shot and custom CNN from sec. 4.4) show improved accuracy as p increases. However, the pure-CNN model benefits more dramatically from increasing p . Its accuracy nearly quadruples from 0.1123 to 0.4894 between $p = 0.001$ and $p = 0.1$, while the few-shot model’s improvement is more gradual (from 0.2095 to 0.3505). The main cause of that might be that the few-shot episodic framework is explicitly designed to handle low-data regimes, so it performs relatively well even when p is small. On the other hand, it is a very lightweight model that may struggle to learn the required patterns even with more data provided. It is however apparent, that the regular convolutional networks rely heavily on the amount of data they are fed with, increasing their performance drastically once a threshold of the training dataset’s size is met.

Table 14: Results of the custom few-shot learning architecture for different fractions p of training data used

p	Accuracy	Recall	Precision	F1 Score
0.001	0.2095 ± 0.0015	0.2095 ± 0.0015	0.2119 ± 0.0010	0.2080 ± 0.0017
0.002	0.2503 ± 0.0004	0.2503 ± 0.0004	0.2537 ± 0.0002	0.2457 ± 0.0010
0.005	0.2890 ± 0.0011	0.2890 ± 0.0009	0.2766 ± 0.0011	0.2755 ± 0.0010
0.01	0.3283 ± 0.0026	0.3283 ± 0.0026	0.3108 ± 0.0013	0.3093 ± 0.0023
0.02	0.3388 ± 0.0026	0.3387 ± 0.0026	0.3271 ± 0.0013	0.3210 ± 0.0023
0.05	0.3407 ± 0.0026	0.3407 ± 0.0026	0.3204 ± 0.0013	0.3123 ± 0.0023
0.1	0.3505 ± 0.0026	0.3505 ± 0.0026	0.3350 ± 0.0013	0.3274 ± 0.0023

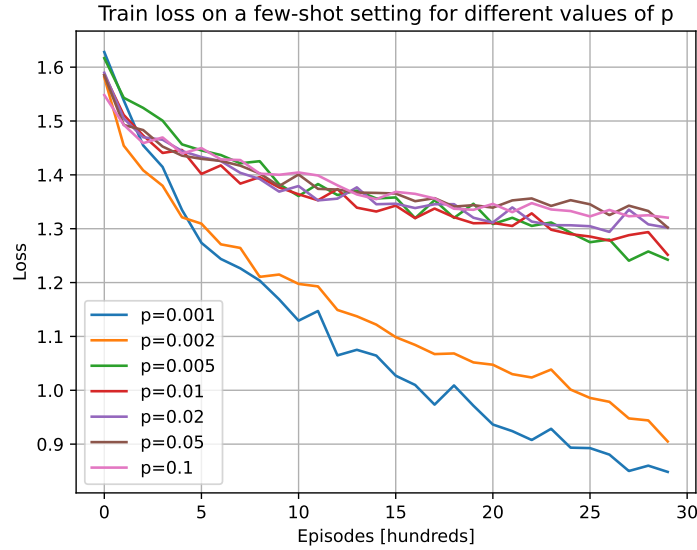


Figure 10: Results of the custom few-shot architecture on different fractions p of the original dataset.

Table 15: Results of the custom non-fewshot learning architecture for different fractions p of training data used

p	Accuracy
0.001	0.1123 \pm 0.0083
0.002	0.1610 \pm 0.0281
0.005	0.1833 \pm 0.0255
0.01	0.2020 \pm 0.0162
0.02	0.3191 \pm 0.0204
0.05	0.3960 \pm 0.0151
0.1	0.4894 \pm 0.0116

12 Summary

This project investigates the application of deep learning techniques for image classification on the CINIC-10 dataset, a more complex variant of CIFAR-10 that combines images from both CIFAR-10 and downsampled ImageNet. The study evaluates several convolutional neural network architectures, including ResNet-18, DenseNet-121, EfficientNet, and a custom-designed CNN, to determine their effectiveness in handling the dataset’s increased variability.

Initial experiments revealed that the custom CNN architecture achieved the highest accuracy (73.29%) among the tested models, demonstrating consistent performance across different augmentation techniques. Data augmentation played a crucial role in improving generalization, with basic transformations such as horizontal flipping and rotation proving effective, while advanced methods like Cutout and AutoAugment provided additional regularization benefits. A hybrid augmentation strategy combining both approaches was ultimately adopted to maximize model robustness.

Hyperparameter optimization was conducted to fine-tune learning rates, optimizers, and regularization techniques. A learning rate of 0.001 with the Adam optimizer yielded the best balance between convergence speed and stability, while dropout and weight decay helped mitigate overfitting. The study also explored a Mixture of Experts (MoE) approach, incorporating specialized networks for vehicle and animal recognition, though further refinement is needed to address overfitting in this framework.

Additionally, a lightweight hybrid CNN-Transformer architecture was proposed for few-shot learning scenarios, demonstrating superior performance in low-data regimes compared to traditional CNNs. This model efficiently combined local feature extraction with global contextual understanding, achieving competitive accuracy while maintaining computational efficiency.

The findings underscore the significance of architecture selection, augmentation strategies, and hyperparameter tuning in optimizing CNN performance for image classification tasks. Future research directions include exploring advanced regularization techniques, refining the MoE framework, and extending the few-shot learning approach to other domains. The project provides a reproducible methodology and insights to guide further advancements in deep learning for computer vision.

13 Reproducibility

All of experiments mentioned in this report can be reproduced using provided Python modules: `training.py` for conventional CNN architectures and `fewshot_training.py` for episode training using custom architecture for this task. Arguments defining the configuration of the training process are listed in Tables 16 and 17 respectively. Both modules assume the datasets are already in the working directory under the `./train`, `./valid` and `./test` catalogues and will automatically save trained models and their progress in appropriate files in the working directory (with the `model_name` parameter as file identifier). All necessary libraries are listed in the `requirements.txt` file.

Table 16: Configuration Parameters for `training.py` module

Argument	Default Value	Description
<code>architecture</code>	<code>custom</code>	Model architecture: [custom, efficientnet, resnet, densenet]
<code>lr</code>	<code>0.001</code>	Learning rate
<code>batch_size</code>	<code>64</code>	Batch size
<code>weight_decay</code>	<code>0.0</code>	Weight decay
<code>optimizer</code>	<code>adam</code>	Optimizer name
<code>dropout</code>	<code>0</code>	Dropout rate
<code>num_trainings</code>	<code>3</code>	Number of training runs
<code>num_epochs</code>	<code>10</code>	Number of epochs per run
<code>model_name</code>	<code>tmp</code>	Model name identifier
<code>double_augment</code>	<code>0</code>	Use double data with one augmented segment [1/0]
<code>fraction</code>	<code>1.0</code>	Fraction of the training data to be used
<code>augmentations</code>	<code>[]</code>	List of augmentations to apply

Table 17: Configuration parameters for `training_fewshot.py` module

Argument	Default Value	Description
<code>num_classes</code>	5	Number of classes (n-way)
<code>cnn_channels</code>	32	Number of CNN output channels
<code>embed_dim</code>	64	Embedding dimension
<code>num_heads</code>	2	Number of attention heads
<code>patch_size</code>	2	Patch size for Vision Transformer
<code>k_shot</code>	5	Number of shots (examples per class)
<code>q_query</code>	15	Number of query samples per class
<code>fraction</code>	0.01	Fraction of the dataset to use
<code>num_trainings</code>	3	Number of training runs
<code>num_episodes</code>	3000	Number of episodes per training run
<code>model_name</code>	'tmp'	Model name identifier
<code>lr</code>	0.001	Learning rate
<code>optimizer</code>	'adam'	Optimizer name
<code>weight_decay</code>	0.0	Weight decay

Therefore, provided that a Python environment is available to a user, a simple command of `pip install -r requirements.txt` followed by `python training.py` will initialize a training process.

References

- [1] H Ali, Summiya Kabir, and Ghufraan Ullah. Indoor scene recognition using resnet-18. *International Journal of Research Publications*, 69(1):7, 2021.
- [2] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.
- [3] Ekin D Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V Le. Autoaugment: Learning augmentation strategies from data. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 113–123, 2019.
- [4] Luke N Darlow, Elliot J Crowley, Antreas Antoniou, and Amos J Storkey. Cinic-10 is not imagenet or cifar-10. *arXiv preprint arXiv:1810.03505*, 2018.
- [5] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. *CoRR*, abs/1703.03400, 2017. URL: <http://arxiv.org/abs/1703.03400>.
- [6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: <http://www.deeplearningbook.org>.
- [7] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017.
- [8] HM Kabir. Reduction of class activation uncertainty with background information. *arXiv preprint arXiv:2305.03238*, 2023.
- [9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 25:1097–1105, 2012. URL: <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>.
- [10] Brian Kulis. Metric learning: A survey. *Foundations and Trends in Machine Learning*, 5(4):287–364, 2013. doi:10.1561/22000000019.
- [11] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi:10.1109/5.726791.

- [12] Keiron O’shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.
- [13] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2009. doi:10.1109/TKDE.2009.191.
- [14] Archit Parnami and Minwoo Lee. Learning from few examples: A summary of approaches to few-shot learning. *ArXiv*, abs/2203.04291, 2022. URL: <https://api.semanticscholar.org/CorpusID:247318847>.
- [15] Adam Santoro, Sergey Bartunov, Matthew M. Botvinick, Daan Wierstra, and Timothy P. Lillicrap. One-shot learning with memory-augmented neural networks. *arXiv preprint arXiv:1605.06065*, 2016. URL: <http://arxiv.org/abs/1605.06065>.
- [16] Jürgen Schmidhuber. Evolutionary principles in self-referential learning. on learning now to learn: The meta-meta-meta...book. Diploma thesis, Technische Universität München, Germany, May 1987.
- [17] Jake Snell, Kevin Swersky, and Richard Zemel. Prototypical networks for few-shot learning. *Advances in neural information processing systems*, 30, 2017.
- [18] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pages 6105–6114. PMLR, 2019.
- [19] Sebastian Thrun and Lorien Pratt, editors. *Learning to Learn*. Kluwer Academic Publishers, USA, 1998.
- [20] Jiyue Wang, Pei Zhang, and Yanxiong Li. Memory-replay knowledge distillation. *Sensors*, 21(8):2792, 2021.
- [21] Yan Wang, Wei-Lun Chao, Kilian Q. Weinberger, and Laurens van der Maaten. Simpleshot: Revisiting nearest-neighbor classification for few-shot learning. *CoRR*, abs/1911.04623, 2019. URL: <http://arxiv.org/abs/1911.04623>, arXiv:1911.04623.