VIETNAM NATIONAL UNIVERSITY
UNIVERSITY OF SCIENCE

Cats and Dogs CNN Classifier

Ho Huyen Trang

Course code: MAT3508
Semester 1, Academic Year 2025-2026

Project Information

Course: MAT3508 – Introduction to Artificial Intelligence
Semester: Semester 1, Academic Year 2025-2026
University: VNU-HUS (Vietnam National University, Hanoi – University of Science)
Project Title: Cats and dogs CNN classifier
Submission Date: 30/11/2025
PDF Report: https://github.com/23001565/Final-project-Cats-and-dogs-CNN-classifier/blob/master/report.p
Presentation Slides: https://github.com/23001565/Final-project-Cats-and-dogs-CNN-classifier/blob/master/AISlide.
GitHub Repository: `https://github.com/23001565/Final-project-Cats-and-dogs-CNN-classifier`
Drive: https://drive.google.com/drive/folders/1eoQ2REdIvF-qhs9Ozx2E0LV29fD3ZTvc?usp=drive$_l ink$

Group Members

Name	Student ID	GitHub Username	Contribution
Ho Huyen Trang	23001565	23001565	All

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Summary

Convolutional neural networks (CNNs) have become a cornerstone of modern machine learning and artificial intelligence, particularly in the field of computer vision. They are designed to automatically learn hierarchical feature representations from raw image data, enabling highly effective solutions for tasks such as object detection, image classification, and visual recognition. While traditional supervised training relies on large labeled datasets, modern deep learning increasingly leverages approaches such as self-supervision, weak supervision, large-scale pretraining, and knowledge distillation to learn powerful representations with far less task-specific labeled data. In this project, I try to train a lightweight CNN model - MobileNetV2 for the task of classifying cats and dogs. Two learning strategies are explored: self-supervised contrastive learning using SimCLR, and a teacher–student knowledge distillation framework with pre-trained ResNet50 teacher.

Experimental results show that a student model trained only through knowledge distillation achieves overall test performance comparable to, and in some cases even surpassing, that of a student initialized with SimCLR-based pretraining. Distillation also takes less training time.

1.2 Problem Statement

Deep convolutional neural networks (CNNs) have achieved remarkable success in image recognition tasks; however, their strong performance typically depends on large-scale labeled datasets and computationally intensive architectures. Lightweight CNNs like MobileNet offer an efficient alternative, but their performance often lags behind larger networks when trained from scratch on limited datasets. To bridge this gap, knowledge transfer techniques have emerged as a powerful strategy. These methods leverage information learned by large, well-trained models—or by related tasks and datasets—to enhance the performance of smaller or data-constrained networks. Approaches such as transfer learning, knowledge distillation, and feature-based adaptation enable lightweight models to inherit useful representations, accelerate convergence, and achieve higher accuracy even when training data are scarce.

Another major challenge lies in data availability. Collecting and labeling massive datasets for every domain is both time-consuming and costly. This has motivated the development of self-supervised learning methods such as SimCLR, which enable models to learn useful feature representations from unlabeled data. Nevertheless, the quality of learned representations can vary depending on the network capacity and initialization, leading to suboptimal results for smaller models.

This project experiments with self-supervised contrastive learning (SimCLR) and classic logits-based knowledge distillation (KD) to improve the performance of a lightweight CNN - MobileNetV2 (trained from scratch), on a classic binary classification task (cats vs. dogs). The aim is to demonstrate the effectiveness of CNN models in general and specifically the lightweight CNN model as well as the two training techniques — SimCLR and KD — particularly in scenarios with limited resources (computation + data, especially labeled data).

Chapter 2

Methods

2.1 CNN Architecture

Convolutional Neural Networks (CNNs) are a class of deep learning models specifically designed for processing grid-like data, such as images. CNNs exploit the spatial structure of images to efficiently learn hierarchical patterns. They achieve this by stacking layers that progressively extract features, from simple edges in early layers to complex textures, shapes, and object-level representations in deeper layers.

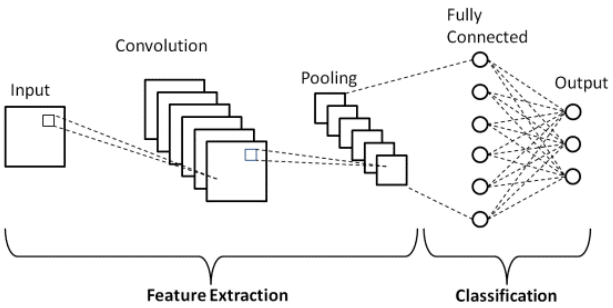

Figure 2.1: CNNs structure

2.1.1 Convolutional Layers and Feature Extraction

A convolutional layer applies a set of learnable filters to an input feature map to extract spatial patterns relevant for visual recognition tasks. In simplified mathematical form, the operation can be written as

$$z = W * x + b,$$

where:

- x is the input tensor (feature map)
- W represents the learnable convolutional kernels (filters)
- $*$ denotes the convolution operator applied spatially over x
- b is the bias term which allows each convolutional filter to shift the output activation (e.g. y = reLU(z)) independently of the weighted sum of inputs, enabling the layer to model patterns that do not naturally pass through the origin. Biases are typically initialized to zero, and like the filter weights, they are updated during training via backpropagation. Each output channel has its own bias value, giving the network additional flexibility in feature extraction.
- z is the resulting output tensor (feature map).

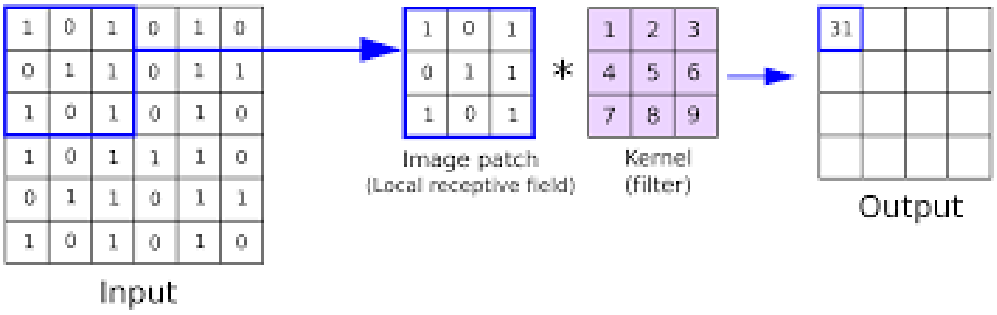

Figure 2.2: Convolution Operation

Filter Initialization

The filters in W are typically initialized randomly using distributions such as Xavier or He initialization. At the start of training, these filters do not encode meaningful patterns; instead, they are simply random tensors. During training, the filters are updated through backpropagation so that they become increasingly effective at capturing the structures present in the training data. Because the learned filters depend entirely on the dataset and optimization process, there are no fixed or universal filters - each network learns its own set of feature detectors tailored to the task.

Multiple filters in a convolutional layer

A single convolutional filter detects only one type of pattern in the input, e.g., a horizontal edge, a vertical edge, or a simple texture. Real images contain many different patterns simultaneously. To capture this diversity, each convolutional layer uses multiple filters, producing multiple output feature maps (channels).

- Each filter acts as a feature detector. They see the same input, but detects different features.
- Together, they allow the network to extract a rich set of complementary features from the same spatial location (capture different types of simple patterns at the same level of abstraction).
- During training, each filter learns a distinct pattern relevant to the task (e.g., edge orientation, color gradient, texture, or object part).

So, a convolutional layer can be thought of as a block of multiple filters applied in parallel, all on the same input, producing a multi-channel output.

Feature Hierarchy in Convolutional Layers

Convolutional layers in CNNs learn hierarchical feature representations, where the type of features captured depends on the layer's depth. This hierarchy arises because each layer builds on the features extracted by previous layers, allowing the network to represent increasingly complex and abstract patterns.

Early layers capture low-level features, such as edges, corners, color gradients, and simple textures. Filters in these layers respond to small, localized regions of the input image. These simple features form the foundation for all subsequent layers, providing basic building blocks for more complex representations.

Intermediate layers combine low-level features into more complex patterns, such as shapes, contours, repeated textures, and object parts (e.g., eyes, ears, wheels). These layers begin to encode spatial relationships between simple features, enabling the network to recognize meaningful components of objects.

Later (deep) layers capture high-level, semantic features that are closely related to the task at hand. For example, filters in deep layers can respond to entire objects (cats, dogs, faces), scene-level patterns, or combinations of multiple object parts into coherent entities. Deeper layers have larger receptive fields, allowing them to integrate information from wider regions of the input image and capture global patterns.

Convolutions are inherently local operations, and stacking multiple layers increases the receptive field of each filter. Early layers "see" only small portions of the image, so they detect simple patterns. As the depth increases, each filter can access a larger portion of the input, allowing the network to combine simple features into more complex, abstract representations that are useful for the final task.

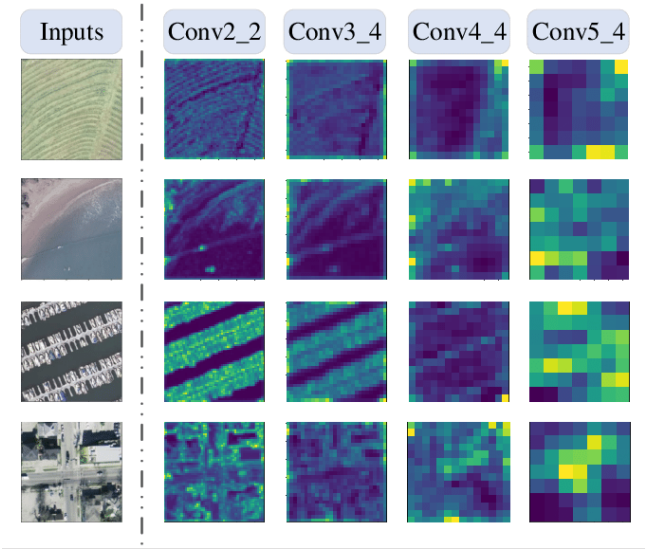

Figure 2.3: Each deep layer sees a more "processed" / compact input so their visualization often looks noisy or incomprehensible. It reflects abstract, distributed representations, not simple edges or colors.

By stacking convolutional layers, CNNs transform raw pixel data into rich hierarchical features. Early layers focus on low-level visual cues, intermediate layers combine these cues into object parts, and later layers encode high-level, semantic information critical for tasks such as classification, detection, and segmentation.

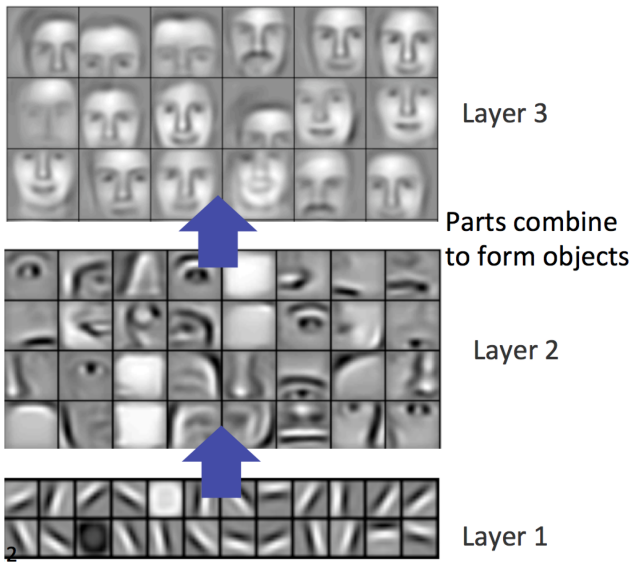

Figure 2.4: Stacking conv layers

2.1.2 Activation Functions

Activation functions are nonlinear operations applied to the output of a neural network layer, such as a convolutional or fully connected layer. Given a pre-activation tensor

$$z = W * x + b,$$

where W is the convolutional kernel, x is the input, and b is the bias, the activation function ϕ produces the output

$$y = \phi(z).$$

Purpose of Activation Functions

The main purpose of activation functions is to introduce **nonlinearity** into the network. Without nonlinear activations, stacking multiple layers would be equivalent to a single linear transformation, regardless of depth. Nonlinear activations allow CNNs to model complex, hierarchical patterns in data, which is essential for tasks such as image classification, detection, and segmentation.

Common Activation Functions

Some widely used activation functions in CNNs include:

- **ReLU (Rectified Linear Unit):** $\text{ReLU}(z) = \max(0, z)$. Introduces sparsity in activations and mitigates vanishing gradient problems.
- **Leaky ReLU:** $\text{LeakyReLU}(z) = \max(\alpha z, z)$ with small $\alpha > 0$, allowing small gradients for negative inputs.
- **Sigmoid:** $\sigma(z) = \frac{1}{1+e^{-z}}$. Compresses outputs to $[0, 1]$; used mostly in output layers for binary tasks.
- **Tanh:** $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$. Maps outputs to $[-1, 1]$, often used in early neural networks.

Nonlinearity in Images

Nonlinearity allows CNNs to capture complex visual patterns that cannot be represented by linear combinations of pixel intensities. For example:

- Edges, textures, and corners combine in nonlinear ways to form object parts.
- Nonlinear activations enable neurons to respond selectively to specific patterns in images rather than just summing input values.
- They allow hierarchical feature composition: simple patterns detected in early layers are combined into increasingly complex representations in deeper layers.

2.1.3 Pooling Layers in Convolutional Neural Networks

Pooling layers are an essential component of Convolutional Neural Networks (CNNs), used to progressively reduce the spatial dimensions of feature maps while retaining important information. By summarizing local regions of the feature maps, pooling layers help reduce computational cost, improve translation invariance, and control overfitting.

Operation

A pooling layer operates on each feature map independently. For a given input feature map $x \in R^{H \times W}$, a pooling function $\mathcal{P}$ computes a downsampled output $y \in R^{H' \times W'}$ over non-overlapping or overlapping windows of size $k \times k$:

$$y_{i,j} = \mathcal{P}\big(x_{m,n} \mid (m, n) \in \text{window}_{i,j}\big)$$

where i, j index the output spatial location, and $\text{window}_{i,j}$ defines the subset of input pixels covered by the pooling operation.

Types of Pooling

Common pooling functions include:

- **Max Pooling:** $y_{i,j} = \max(x_{m,n})$ over the pooling window. Captures the most prominent feature in a local region.
- **Average Pooling:** $y_{i,j} = \frac{1}{k^2} \sum x_{m,n}$ over the pooling window. Computes the average feature intensity.
- **Global Pooling:** Aggregates all values in a feature map into a single number, often used before fully connected layers (e.g., global average pooling).

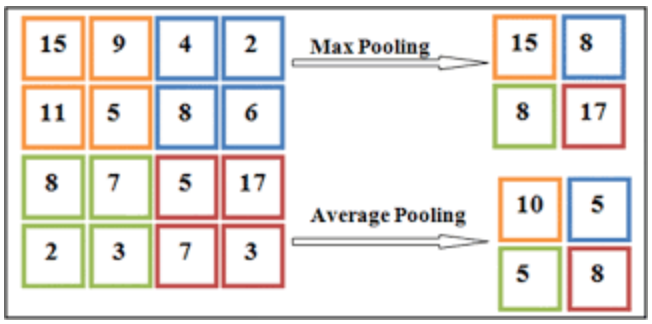

Figure 2.5: Max pooling and avg pooling

Purpose and Benefits

Pooling layers serve several important purposes:

- **Dimensionality Reduction:** Reduces height and width of feature maps, lowering computational cost and memory usage.
- **Translation Invariance:** Summarizes local neighborhoods so that small shifts in the input do not significantly change the output.
- **Feature Emphasis:** Max pooling, for instance, preserves the strongest activation in a region, emphasizing salient features.
- **Overfitting Control:** By reducing feature map size, pooling decreases the number of parameters in downstream layers, mitigating overfitting.

Summary Pooling layers are non-learnable, parameter-free layers that summarize local neighborhoods of feature maps. They are used to reduce spatial dimensions, emphasize important features, improve translation invariance, and facilitate hierarchical feature learning in CNNs.

2.1.4 Fully Connected Layers

Fully connected (FC) layers, also called dense layers, are used to integrate and map features extracted by convolutional and pooling layers to the desired output space. They are typically placed after the last convolutional/pooling layer and receive flattened feature vectors as input.

Linear Operation of FC Layers

An FC layer computes a linear transformation of its input:

$$\mathbf{z} = W\mathbf{x} + \mathbf{b}$$

where

- $\mathbf{x}$ is the 1D input vector ((C, H, W) -¿ (C x H x W)),
- W is the weight matrix connecting all input features to all output neurons,
- $\mathbf{b}$ is the bias vector,
- $\mathbf{z}$ is the output vector, often called *logits* before any activation.

Weight and Bias Initialization

- **Weights (W)** are typically initialized randomly before training. Common methods include:

- **Xavier / Glorot Initialization:** Suitable for tanh or sigmoid activations, scales weights based on the number of input and output neurons.
- **He / Kaiming Initialization:** Suitable for ReLU activations, scales weights based on the number of input neurons to prevent vanishing or exploding gradients.

- **Biases (b)** are usually initialized to zero or small constants and are updated during training.

Proper initialization ensures stable activations and gradients, which facilitates faster and more reliable training of the network.

Roles of the FC Layer:

- **Feature Integration:** Each output neuron receives input from all features, allowing it to detect specific combinations of patterns from previous layers.
- **Dimensionality Transformation:** FC layers can reduce or expand the input vector size to match the output requirements.
- **Task-Specific Linear Mapping:** Map learned features to the target output space, such as class logits for classification, continuous values for regression, or embeddings for similarity tasks.
- **Parameterization / Learning Capacity:** Being fully connected, these layers have a high number of parameters, enabling the network to capture complex correlations among input features.

Activation Functions in FC Layers

An activation function ϕ is applied to the linear output $\mathbf{z}$ to introduce nonlinearity:

$$\mathbf{y} = \phi(\mathbf{z}) = \phi(W\mathbf{x} + \mathbf{b})$$

- **Hidden FC layers:** Use nonlinear activations like ReLU or tanh to increase expressive power.
- **Output FC layers:** Use task-specific activations:
  - Softmax for multi-class classification,
  - Sigmoid for binary or multi-label classification,
  - Linear (no activation) for regression or embeddings.

2.1.5 Weight Updates: Loss and Backpropagation in CNNs

Convolutional Neural Networks (CNNs) learn by adjusting their weights and biases to minimize a loss function that measures the discrepancy between predicted outputs and ground truth labels. This process is known as *training*, and it relies on the principles of *backpropagation* and gradient-based optimization.

- Each training iteration = 1 forward pass + 1 backward pass + 1 parameter update. (an iteration here = processing of 1 batch in an epoch and epoch is a full pass through the entire training set, so we'd have train-set-size/batch-size iterations per epoch)

- Backprop moves layer-by-layer from last to first once.

- Weight updates happen after gradients for all layers are computed. SGD, Adam, etc. use the accumulated gradients and all parameters are updated simultaneously

Loss Function

The **loss function** quantifies how well the network predicts the desired output. Common choices include:

- **Cross-Entropy Loss** for classification tasks:

$$\mathcal{L} = -\sum_i y_i \log(\hat{y}_i)$$

  where y_i is the true label (one-hot encoded) and $\hat{y}_i$ is the predicted probability from the softmax output.

- **Mean Squared Error (MSE)** for regression tasks:

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

The goal of training is to find weights W and biases b that minimize the loss over the training data.

Backpropagation and Gradient Computation

Backpropagation is the algorithm used to compute gradients of the loss with respect to all learnable parameters in the network.

1. Forward Pass: Compute activations of all layers and the loss $\mathcal{L}$.
2. Backward Pass: Propagate the error from the output layer to earlier layers using the chain rule:

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial W}$$

where $z = Wx + b$ is the linear output of a layer.

Weight Update

Once the gradients are computed, weights and biases are updated using an optimization algorithm, typically Stochastic Gradient Descent (SGD) or its variants:

$$W \leftarrow W - \eta \frac{\partial \mathcal{L}}{\partial W}$$

where η is the learning rate, controlling the step size of each update.

Weight updates in CNNs are driven by the loss function and calculated using backpropagation. Gradients of the loss with respect to weights and biases indicate the direction and magnitude of updates. This process enables CNNs to learn complex hierarchical features from input data automatically, making them powerful feature extractors that generalize across tasks and data.

2.2 SimCLR: Simple Framework for Contrastive Learning of Visual Representations

SimCLR is a self-supervised learning framework that learns high-quality visual representations without labeled data. It relies on *contrastive learning*, where the model learns to bring similar representations closer and push dissimilar ones apart.

2.2.1 Main Idea

SimCLR aims to learn representations by maximizing agreement between differently augmented views of the same image (positive pairs) while minimizing agreement between views from different images (negative pairs). No manual labels are needed; the model creates supervision via a *contrastive objective*. Learned representations can be fine-tuned on downstream tasks such as classification or detection.

2.2.2 Key Components

Data Augmentation

Data augmentations generate multiple views of the same image (positive pairs). Typical augmentations include:

- Random cropping and resizing
- Color jittering
- Gaussian blur
- Horizontal flipping

Strong and diverse augmentations are crucial. Weak augmentations lead to trivial solutions; overly strong augmentations can destroy semantic content. Crop + color jitter + blur is particularly effective.

Base Encoder

A neural network $f(\cdot)$ extracts representations $h = f(x)$ from images. A *projection head* (MLP) then maps h to a space suitable for contrastive loss: $z = g(h)$ (they find it beneficial to define the contrastive loss on z rather than h). The projection head is discarded after pretraining; the base encoder is used for downstream tasks. Also, deeper networks (e.g., ResNet-50 or ResNet-152) seem to yield better representations.

NT-Xent or Normalized Temperature-scaled Cross Entropy Loss

For each batch of N samples, we get 2N augmented samples. We do not sample negative examples explicitly. Instead, each sample has 1 positive (its augmented twin) and 2(N 1) negatives (all others).

Let $\text{sim}(u, v) = \frac{u^\top v}{\|u\| \|v\|}$ denote the dot product between ℓ_2–normalized vectors u and v (i.e., cosine similarity). Then the loss function for a positive pair of examples (i, j) is defined as

$$\ell_{i,j} = -\log \frac{\exp(\text{sim}(z_i, z_j)/\tau)}{\sum_{k=1}^{2N} \mathbf{1}_{[k \neq i]} \exp(\text{sim}(z_i, z_k)/\tau)}, \tag{2.1}$$

where $\mathbf{1}_{[k \neq i]} \in \{0, 1\}$ is an indicator function that evaluates to 1 iff $k \neq i$, and τ denotes a temperature parameter.

Interpretation:

- Numerator: similarity with its **positive pair**
- Denominator: similarity with **all other samples** (positives + negatives)
- The log-softmax pushes up similarity with j and pushes down similarity with all negatives, encouraging representations of positive pairs to be close and all others to be far apart.

The final loss is computed across all positive pairs, both (i, j) and (j, i), within a batch.

$$\mathcal{L} = \frac{1}{2N} \sum_{i=1}^{2N} \ell_{i,j(i)}$$

Large batch sizes are important because contrastive loss relies on many negative examples. Small batches reduce negative pairs, hurting performance.

2.2.3 Summary

SimCLR demonstrates that *strong augmentations + a good encoder + contrastive loss* can produce representations rivaling supervised learning. Its simplicity and effectiveness have made it a foundation for many subsequent self-supervised learning methods.

2.3 Knowledge Distillation Loss

Knowledge Distillation (KD) introduced by G. Hinton in 2015, was designed primarily to compress large models, create lightweight, fast, deployable students without losing much accuracy. It transfers knowledge from a large *teacher* network to a smaller *student* network by training the student to match the teacher's softened output distribution.

Let z_t and z_s denote the teacher and student logits (pre-softmax outputs), respectively.

2.3.1 Softened Softmax with Temperature

To reveal the teacher's "dark knowledge" - its belief about class similarities - both teacher and student logits are passed through a *temperature-scaled* softmax:

$$p_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)} \tag{2.2}$$

where p_i is the softened probabilities for class i and T is the temperature. T ¿ 1 produces a softer probability distribution. When T = 1, we have the standard softmax.

2.3.2 Soft Target Distillation Loss

The core KD objective encourages the student to match the teacher's softened output distribution. This is accomplished using the Kullback–Leibler (KL) divergence:

$$\mathcal{L}_{\text{soft}} = \text{KL}\left(p^t \,\|\, p^s\right) = \sum_{i=1}^{C} p_i^t \log \frac{p_i^t}{p_i^s}.$$

This loss transmits rich information about relative class similarities that is not present in one-hot labels.

2.3.3 Hard Label Loss

Optionally, the student is also trained on the ground-truth labels using the standard cross-entropy loss:

$$\mathcal{L}_{\text{hard}} = CE(y_{true}, softmax(z_s))$$

2.3.4 Final Distillation Objective

To properly balance the gradients contributed by the softened KL term, the authors recommend scaling $\mathcal{L}_{\text{soft}}$ by T^2 for stability. The final KD loss is

$$\mathcal{L} = \alpha\, \mathcal{L}_{\text{hard}} \;+\; \beta\, .T^2\, \mathcal{L}_{\text{soft}},$$

where α and β control the tradeoff between fitting ground-truth labels and matching the teacher's distribution.

Chapter 3

Implementation

I train a lightweight model, MobileNetV2, on a binary cats vs. dogs classification task using two different training schemes:

(1) Self-supervised learning following the SimCLR framework, and (2) Knowledge distillation, using ResNet50 as the teacher model.

All experiments were conducted on Google Colab (free tier), with training performed primarily on the available GPU.

I compare the testing performance of the two training schemes—evaluating accuracy, precision, and recall for each class—along with the training time required by each approach. This allows for a clear assessment of both effectiveness and computational efficiency.

3.1 Dataset Description

The dataset used in this project is the *Cats and Dogs Classification Dataset* from Kaggle, consisting of a total of 24,998 images, evenly split into 12,499 cat images and 12,499 dog images.

To simulate a realistic scenario with limited labeled data, the following subsets were constructed:

- **Unlabeled training data:** A pool of 10,000 images was used as the primary unlabeled set. From this pool, two smaller unlabeled subsets of 6,000 and 3,000 images were derived for comparison.
- **Labeled data:** A total of 600 labeled images were used, divided into:
  - 420 images for fine-tuning,
  - 180 images for validation.
- **Test set:** A fixed set of 5,000 images was used to ensure reliable and consistent evaluation across all experiments.

All subsets maintain a balanced class ratio (50:50) of cats and dogs. Thus, the experiments employ three unlabeled training sets of different sizes, while the fine-tuning, validation, and test sets remain identical across all experimental configurations.

3.2 Model Architecture

In this project, I use the lightweight `MobileNetV2` model without any pretrained weights, as well as a `ResNet50` model initialized with `IMAGENET_1K` pretrained weights. Both architectures are obtained from the `torchvision` library.

3.3 SimCLR implementation

For the self-supervised experiments, the `lightly` library is used to implement SimCLR. The library provides built-in support for:

- **SimCLR-style data augmentations:** Random cropping, resizing, color jittering, and Gaussian blur, implemented via the `SimCLRTransform()` function.
- **Contrastive loss:** The normalized temperature-scaled cross-entropy loss (NT-Xent), implemented via `NTXentLoss()`.

Table 3.1: Summary of MobileNetV2 and ResNet50 architectures

Feature	**MobileNetV2**	**ResNet50**
Type	Lightweight CNN	Deep Residual Network
Core Idea	Inverted residual blocks with linear bottlenecks	Residual blocks with skip connections
Initial Layers	3×3 Conv + BN + ReLU6	7×7 Conv + Max Pooling
Main Building Blocks	Inverted residual blocks (expansion → depthwise conv → projection)	Bottleneck residual blocks (1×1 → 3×3 → 1×1)
Pooling	Global Average Pooling	Global Average Pooling
Fully Connected	1 FC layer	1 FC layer
Total Parameters	∼3.5M	∼25.6M
Strengths	Fast, memory-efficient, suitable for limited-resource scenarios	High accuracy, powerful feature extractor, ideal teacher for knowledge distillation

3.4 KD implementation

For the knowledge distillation experiments, the loss functions are implemented as follows:

- **Hard loss:** Standard cross-entropy (CE) loss is used, implemented via `torch.nn.CrossEntropyLoss`.
- **Soft loss:** Composed of multiple functions from `torch.nn.functional`, including `log_softmax`, `softmax`, and `kl_div`, to compute the Kullback-Leibler divergence between teacher and student outputs.

3.5 Training Details

Both training implementations employ the **Adam optimizer** for its fast convergence and suitability for small datasets. The selection of the best model checkpoints as well as other parameters is performed manually by monitoring the training process, including convergence behavior, loss values, and accuracy metrics.

Chapter 4

Results & Analysis

4.1 Accuracy

Table 4.1: SimCLR Results

Train Set Size	Test Acc	PrecisionC	RecallC	PrecisionD	RecallD
10,000	0.78	0.78	0.78	0.78	0.79
6,000	0.76	0.79	0.71	0.73	0.81
3,000	0.72	0.72	0.71	0.72	0.72

SimCLR Despite the large reduction in training data from 10,000 to 3,000 samples, the model's performance remains very stable. The decrease in accuracy is modest, and both precision and recall stay well-balanced across the two classes, indicating no major shift or bias. This stability suggests that the model is able to retain meaningful representations even with substantially fewer labeled examples. Notably, this highlights the effectiveness of SimCLR: its self-supervised pretraining helps the network learn robust, transferable features that reduce reliance on large labeled datasets. As a result, the model maintains strong and balanced class performance even under severe data constraints.

Table 4.2: KD Results

Train Set Size	Test Acc	PrecisionC	RecallC	PrecisionD	RecallD
10,000	0.92	0.94	0.90	0.90	0.94
6,000	0.84	0.93	0.73	0.78	0.94
3,000	0.75	0.79	0.69	0.72	0.81

KD Steeper performance declines as the training set shrinks and bigger gaps between precisions and recalls of both classes. With 10,000 samples, the student network achieves strong accuracy (0.92) and well-balanced precision and recall across both classes, indicating effective knowledge transfer from the teacher. However, at 6,000 samples, the results become skewed: precision and recall for Cat remain high, while those for Dog drop noticeably, I might not run enough epochs in this case. As the data is reduced to 3,000 samples, accuracy drops to 0.75, yet the model maintains relatively consistent behavior between classes—precision and recall decrease . This suggests that KD helps the lightweight student model retain useful decision boundaries even under limited labeled data.

Comparison Comparing the two training strategies, distilled models consistently outperform SimCLR across all training set sizes. The difference is most pronounced with 10,000 samples, where Knowledge Distillation achieves 0.91 accuracy compared to 0.78 for SimCLR, highlighting its strong suitability for scenarios with limited labeled data. At the same time, SimCLR exhibits a more gradual decrease in performance as well as stable balances among metrics of each class and among the two classes as the dataset shrinks, indicating that self-supervised pretraining provides stable feature representations even with very small datasets. Overall, KD delivers higher absolute accuracy, while SimCLR offers more consistent robustness to data reduction.

4.2 Training time

Table 4.3: Total epochs for training

Train Set Size	SimCLR-Pretrain only	Distillation
10,000	62	46
6,000	57	35
3,000	59	25

SimCLR (finetuning exclusive) generally requires more epochs than KD, especially for the largest dataset (62 vs. 46 for 10,000 samples). For smaller datasets, such as 3,000 samples, SimCLR still requires a high number of epochs (59) because the limited data prevents the model from learning sufficiently useful features, causing the loss to decrease very slowly and never achieve the sharp reductions seen with 10,000 samples. In contrast, KD converges faster as the dataset size decreases, dropping from 46 to 25 epochs, since the student model can leverage knowledge from the teacher. This highlights KD's advantage in low-data settings, both in efficiency and in achieving meaningful learning.

Chapter 5

Conclusion

5.1 Conclusion

This experiment demonstrates the effectiveness of CNN models in low-resource scenarios, with a particular focus on lightweight architectures such as MobileNet. We can see that Knowledge Distillation consistently improves accuracy across all training set sizes, indicating its efficiency when labeled data are limited. On the other hand, SimCLR shows very stable performance even with small datasets, highlighting the benefits of self-supervised pretraining. Additionally, in terms of training epochs, we see that KD converges faster, while SimCLR's pretraining can require many epochs on smaller datasets due to slow loss reduction. Overall, the results highlight the effectiveness of the CNNs structure and the two classic training techniques for achieving robust and balanced performance under constrained data and computational resources.

Future Directions: Future work could explore hybrid approaches that combine KD and self-supervised pretraining to further improve performance on extremely small datasets. Investigating adaptive data selection or augmentation strategies could enhance model generalization, while automated architecture search for lightweight models may provide better accuracy-efficiency trade-offs. Finally, extending these methods to diverse tasks and modalities beyond standard image recognition would help assess their general applicability in real-world low-resource settings.

5.2 Future Experiments

For further experiments, I would like to explore in several directions:

- **Hyperparameter Tuning:** Select the best parameters to achieve better performance (the augmentations, the learning rates, the temperature of softmax in KD, ratio of hard loss/soft loss, etc.)
- **Model Variations:** Investigating alternative student or teacher architectures, see if models with similar structures would perform KD better.
- **Further distillation:** transfer more infomations like features learned by teacher to student.
- **Adaptivity:** Especially exploring how the students would learn new classes, how well they would generalize, how to keep them to learn new information without forgetting old ones and also the limits of their learning capabilities, etc.

Bibliography

[1] MIT 6.S191 (2024): Convolutional Neural Networks `https://www.youtube.com/watch?v=2xqkSUhmmXU`

[2] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A Simple Framework for Contrastive Learning of Visual Representations," in *International Conference on Machine Learning (ICML)*, 2020.

[3] SimCLR Lightly doc `https://docs.lightly.ai/self-supervised-learning/examples/simclr.html`

[4] Hinton, Geoffrey E., Oriol Vinyals, and Jeff Dean, "Distilling the Knowledge in a Neural Network." arXiv preprint arXiv:1503.02531, 2015.