EE6483 Project Report: Image Classification of Cats and Dogs

Han Yiqing (G2504397F), Lu Jinyu (G2506543E), and Zhang Xuesi (G2407594K)
School of Electrical & Electronic Engineering
Nanyang Technological University
Singapore
Email: YHAN031@e.ntu.edu.sg, JINYU002@e.ntu.edu.sg, XUESI001@e.ntu.edu.sg

***Abstract*—This report details a project on binary image classification to distinguish between cats and dogs. We employ a transfer learning approach using a ResNet18 model pre-trained on ImageNet, fine-tuning it on a reduced dataset of 5,000 training and 2,000 validation images. Key parameters include the Adam optimizer, a 1e-3 learning rate, and cross-entropy loss. Our model achieved a peak validation accuracy of 97.55%. We also analyze model weaknesses, such as confusion from occluding objects or distracting backgrounds. Finally, we demonstrate the algorithm's adaptability by modifying it for the 10-class Cifar-10 dataset, which involves up-scaling 32x32 images and changing the final classifier head.**

***Index Terms*—Image Classification, Deep Learning, Convolutional Neural Networks (CNN), ResNet, Transfer Learning, PyTorch, Cifar-10.**

I. Literature Review

A. Problem Statement

Image classification tasks aim to enable models to determine the category to which an input image belongs. In this project, the objective is to identify whether an image depicts a "cat" or a "dog", constituting a binary supervised learning problem.

The formal definition is as follows: Let the input image set be

$$X = \{x_1, x_2, \ldots, x_N\} \tag{1}$$

Each sample x is an RGB image (typically represented as a matrix $[H, W, 3]$). Each image corresponds to a label

$$y \in \{0, 1\} \tag{2}$$

Given training set:

$$D_{train} = \{(x_i, y_i)\}_{i=1}^{N} \tag{3}$$

Where x_i denotes the image, and $y_i \in \{0, 1\}$ denotes the category label (0 = cat, 1 = dog). The objective is to train a parametric model $f_\theta(x)$ such that the output category prediction $\hat{y} = f_\theta(x)$ approximates the true label y as closely as possible.

That is, the model is expected to learn a nonlinear mapping from the pixel-level input space to the high-level semantic space:

$$f : \mathcal{X} \to \mathcal{Y} \tag{4}$$

f is implemented by machine learning models such as neural networks.

In image classification, researchers describe the task under different settings according to data type and learning conditions. Common settings include supervised and unsupervised learning, closed-set and open-set recognition, and learning with or without domain shift. These categories highlight the challenges that models face and guide the design of suitable algorithms.

a) Supervised learning: In supervised learning, models learn from large collections of labeled images. The advantage of this setting is that the training goal is clear and the optimization path is well defined. However, it also brings practical difficulties. Labeling data requires time and expert knowledge, which makes it expensive and limits the available dataset size. Another challenge is overfitting. When a model fits the training data too closely, it performs poorly on new images. To overcome these issues, researchers often use transfer learning and data augmentation. Transfer learning applies pretrained models such as VGG or ResNet, trained on large datasets like ImageNet, to reduce the need for new labeled samples. Data augmentation adds variations to existing images through rotation, flipping, translation, or color adjustment to make models more robust. Regularization methods such as dropout or weight decay also help control overfitting. In addition, self-supervised pretraining followed by fine-tuning with a small labeled set has become an effective strategy to improve generalization.

b) Unsupervised learning: In unsupervised learning, there are no labels for the training data. The model must learn patterns directly from data distributions. Without supervision, it is difficult for the model to learn features that are both clear and discriminative. The learned features may be hard to interpret, and clustering results often differ from true classes. Recent progress in self-supervised and contrastive learning has improved unsupervised methods. By performing additional tasks such as predicting image rotation, reconstructing missing parts, or comparing feature embeddings, the model can learn meaningful representations without labels. These learned representations can later be fine-tuned with labeled data to reach accuracy close to supervised learning.

c) Closed-set recognition: Closed-set recognition assumes that the training and test sets contain the same categories. Models trained under this assumption usually use a Softmax classifier and cross-entropy loss and achieve stable

performance. However, these models fail when new, unseen classes appear in the test set. This situation is known as open-set recognition. In open-set settings, the model must classify known samples correctly and identify unknown ones. Traditional classifiers often assign every input to a known label, which causes misclassification of unknown data. Solutions include OpenMax-based unknown detection, confidence thresholding, energy-based scoring, and contrastive learning for feature boundary refinement. These methods help models perform more reliably when dealing with unfamiliar data.

d) Domain shift: The domain shift problem is also common in real-world applications. When the training and test data come from the same source, the model can be trained and validated in a normal way. This is called the no domain shift condition. When the distributions differ, such as through changes in lighting, angle, background, or resolution, model accuracy often drops. Researchers address this issue with two main approaches: domain adaptation and domain generalization. Domain adaptation aligns features from different domains using methods like Maximum Mean Discrepancy (MMD) or adversarial learning. Domain generalization aims to make models more robust across different domains using techniques such as style transfer or self-supervised pretraining.

Each setting focuses on a different aspect of learning. Supervised learning builds discriminative ability from labeled data, while unsupervised learning discovers hidden structures from unlabeled data. Closed-set recognition deals with known categories, whereas open-set recognition handles unknown ones. No domain shift assumes stable data distribution, while domain shift tests model robustness under changing conditions.

In the Dogs vs. Cats project, the task is a typical example of supervised, closed-set, and no-domain-shift classification. The main goal is to improve accuracy and generalization by using data augmentation, suitable network structures, and careful parameter optimization.

B. Review of the Literature: The Foundations of Deep Convolutional Networks

The modern era of deep learning for image classification was largely built upon two seminal architectures that defined the field's trajectory: VGG and ResNet.

The **VGG (Visual Geometry Group) network** [1], introduced in 2014, was a systematic investigation into the importance of network depth. The VGG architecture is characterized by its profound simplicity and homogeneity, consisting almost entirely of stacked 3x3 convolutional filters. This design choice was a key innovation; the authors argued that a stack of small filters (e.g., two 3x3 layers) could replicate the effective receptive field of a larger filter (e.g., one 5x5 layer). This approach offered two distinct advantages: it reduced the total number of parameters and, crucially, it allowed for more non-linear activation layers, making the decision function more discriminative. The resulting VGG-16 and VGG-19 models were extremely deep for their time and demonstrated a clear relationship between depth and performance, securing top results at the ILSVRC-2014 competition [1].

However, the success of VGG exposed a fundamental barrier. Researchers found that simply stacking more layers on a "plain" network (in the style of VGG) led to a counter-intuitive **"degradation problem"** [2]. As defined in the ResNet paper, this was an optimization issue, not an overfitting one: a deeper 56-layer plain network was found to have a *higher training error* than its shallower 20-layer counterpart [2]. This indicated that deeper models were becoming too difficult for standard stochastic gradient descent to optimize effectively [2].

ResNet (Residual Networks), introduced by Kaiming He et al. in 2015, provided an elegant solution to this degradation problem [2]. The authors proposed a **Deep Residual Learning** framework. Instead of hoping a stack of layers would learn a desired underlying mapping $H(x)$, the architecture was reformulated to learn a *residual mapping* $\mathcal{F}(x) := H(x) - x$. The block's output was then defined as $y = \mathcal{F}(x) + x$. This was implemented through a **"skip connection"** (or "shortcut connection") that passed the input x directly, element-wise, to the block's output [2].

This architectural tweak was profound. It allowed the network to, in the worst-case scenario, simply learn an identity mapping (by driving the weights of $\mathcal{F}(x)$ to zero), ensuring that a deeper model would never perform worse than a shallower one . This "skip connection" enabled the stable training of networks that were substantially deeper than ever before—such as 50, 101, and even 152 layers. ResNet won the ILSVRC 2015 competition, and the residual block became the foundational "building block" for countless subsequent architectures.

C. Overview of New Research in the Field

After ResNet solved the problem of training *deep* networks, the field's focus shifted. Research evolved beyond simply adding layers and branched into several distinct trends, all building upon the modular "backbone" concept ResNet had established.

a) Trend 1: The Pursuit of Architectural Efficiency: This line of research moved away from "depth at all costs" and focused on creating architectures that were more parameter- and computationally-efficient. **DenseNet (Densely Connected Convolutional Networks)** [3] proposed a different connectivity pattern. Instead of using skip connections to *sum* features, DenseNet's "dense blocks" *concatenated* the feature maps from every preceding layer with every subsequent layer [3]. This encouraged **"feature reuse"** on a massive scale, leading to high parameter efficiency and improved information flow [3].

For edge-device applications, **MobileNet** [4] introduced **"depthwise separable convolution"** [4]. This mechanism factorized a standard convolution into two much cheaper operations: a 3x3 depthwise convolution (which filters spatially, one channel at a time) and a 1×1 pointwise convolution (which combines the channels) [4]. This factorization dramatically

TABLE I
SUMMARY TABLE OF DIFFERENT SETTINGS

Setting Type	Key Challenges	Common Solutions	Implications for the "Dogs vs. Cats" Task
Supervised Learning	High labeling cost; risk of overfitting	Transfer learning, data augmentation, regularization	Use pretrained CNNs such as VGG or ResNet to improve accuracy
Unsupervised Learning	No labels; clustering alignment difficulty	Self-supervised learning, feature clustering	Can be used as a pretraining stage to learn representations
Closed-set Recognition	Lack of awareness for unseen classes	Softmax classifier, cross-entropy loss	Fits the current binary classification setup
Open-set Recognition	Unknown classes may be misclassified	OpenMax, confidence/energy-based detection	Can be extended to multi-class or real-world problems
Without Domain Shift	Limited generalization ability	Standard train/validation split, early stopping	Default assumption for this project
With Domain Shift	Distribution difference	Domain adaptation, style transfer, normalization tuning	Useful for extending to cross-dataset environments shallower one

reduced the computational cost (FLOPs) with only a minor trade-off in accuracy [4].

This trend culminated in **EfficientNet** [5]. Its authors argued that scaling a network (e.g., from a ResNet-50 to a ResNet-101) was typically done haphazardly, scaling only depth or width. They proposed a **"compound scaling"** method [5], which uses a single coefficient ϕ to uniformly scale all three dimensions of the network—depth, width, and image resolution—in a balanced and principled way [5]. This systematic approach created a family of models that achieved state-of-the-art accuracy while being significantly more efficient than previous architectures [5].

b) Trend 2: Automated Architecture Design (NAS): This trend sought to automate the human-driven, trial-and-error process of designing network architectures. **Neural Architecture Search (NAS)** emerged as a field dedicated to using algorithms, often based on reinforcement learning, to *discover* optimal network structures. A landmark paper, **NASNet** [6], made this feasible for large-scale problems. Instead of searching for an entire network, the NASNet controller was trained to find an optimal "convolutional cell" on a small proxy dataset (e.g., CIFAR-10) [6]. This "transferable" cell was then stacked to build a state-of-the-art model for a larger dataset (e.g., ImageNet) [6]. This cell-based search strategy proved that discovered architectures could outperform the best human-designed ones [6].

c) Trend 3: The Rise of Transformers: This trend represented a paradigm shift, challenging the dominance of CNNs by adapting the Transformer architecture, which was already supreme in natural language processing. The **Vision Transformer (ViT)** [7] was the first to show this could work at scale. It did so by discarding the inductive biases of convolution entirely. A ViT treats an image as a sequence of 16x16 **"patches"** [7]. These patches are flattened, linearly embedded, and (crucially) **"position embeddings"** are added to retain spatial information [7]. This sequence of "tokens" is then fed into a standard Transformer encoder. ViT demonstrated that, when pre-trained on massive datasets, this architecture could outperform state-of-the-art ResNets.

A key limitation of ViT was its global self-attention, which has a computational complexity quadratic to the number of patches ($O(N^2)$), making it difficult to use with high-resolution images [8]. The **Swin Transformer** [8] solved this by introducing a hierarchical architecture (like CNNs) and computing self-attention *locally* within non-overlapping **"windows"** [8]. To allow information to mix between windows, it employed a **"shifted window" (SW-MSA)** mechanism in alternating layers [8]. This achieved linear complexity ($O(N)$) and established Transformers as a powerful, general-purpose backbone for a wide array of vision tasks [8].

d) Trend 4: The Self-Supervised Learning (SSL) Paradigm: The final trend shifted focus away from architectural engineering and onto the *learning signal* itself [9]. SSL methods aim to learn rich visual representations from massive amounts of unlabeled data, using the data itself to generate supervisory signals.

- **Contrastive Methods:** These methods learn by pulling augmented views of the same image (positive pairs) closer in the embedding space, while pushing views from different images (negative pairs) apart. **SimCLR** [10] provided a simple framework, showing that the combination of strong data augmentations and a non-linear **"projection head"** (an MLP added after the encoder during training) was critical for success [10]. **MoCo (Momentum Contrast)** [11] solved the large-batch dependency of SimCLR by maintaining a **"queue"** of recent batches as a dynamic dictionary of negative samples [11]. To keep the "key" representations from this dictionary consistent, it used a **"momentum encoder"**—a slow-moving average of the online encoder [11].
- **Asymmetric (Non-Contrastive) Methods:** This family proved that negative samples were not strictly necessary. **BYOL (Bootstrap Your Own Latent)** [12] used an asymmetric architecture. An **"online" network"** is trained to predict the representation from a **"target" network"** [12]. The target network is not updated by gradients; instead, its weights are a slow-moving average of the online network's weights [12]. This design success-

fully prevents the network from "collapsing" to a trivial solution, achieving state-of-the-art results without using a single negative sample [12].

D. Baseline Approach and Proposed Improvements

Recent studies in image classification suggest that a convolutional neural network (CNN) using supervised learning is the most suitable baseline for the Dogs vs. Cats task. CNNs are effective at learning spatial and hierarchical patterns in images, which makes them ideal for binary visual classification. Well-known networks such as VGG16 and ResNet-18 are often used as baselines because they provide good accuracy and reasonable computational cost. These models offer a strong starting point for building and comparing results in this project.

The baseline model follows a supervised, closed-set, and no-domain-shift setting. It learns from labeled images of cats and dogs by minimizing the cross-entropy loss between predictions and true labels. The training and validation data share the same distribution, so the model focuses on learning reliable visual features.

In this setting, transfer learning is an effective approach. A pretrained model such as VGG16 or ResNet can be fine-tuned on the project dataset. This approach allows faster training, requires fewer labeled samples, and makes use of the general visual knowledge learned from ImageNet.

Several improvements can make this baseline perform better on the current dataset.

(a) **Data preprocessing and augmentation** The dataset contains images with different sizes, brightness, and backgrounds. Before training, images should be resized to a fixed shape (for example, 224×224) and normalized. Data augmentation helps create more diverse samples and prevents overfitting. Common operations include random rotation, flipping, scaling, and color changes. These transformations help the model become more robust to variations in image conditions.

(b) **Model adaptation and fine-tuning** Training a CNN from scratch requires large data and long training time. A better way is to start with pretrained weights, such as those from ResNet-18 trained on ImageNet. The convolutional layers can serve as feature extractors, and the last layer can be replaced with a new one for two-class prediction. Fine-tuning deeper layers with a smaller learning rate helps the model adjust to the cat–dog dataset while keeping useful general features.

(c) **Optimization and regularization** Tuning training parameters improves both accuracy and stability. The learning rate, batch size, and optimizer (for example, Adam or SGD) should be chosen carefully. Regularization methods like dropout, weight decay, and early stopping can reduce overfitting. Tracking validation accuracy during training helps select the best-performing model.

(d) **Evaluation and error analysis** The model should be tested on the validation set and evaluated using accuracy and confusion matrices. Visualizing correct and incorrect predictions helps understand model behavior. Misclassified images often show where the model struggles, such as with similar textures or mixed lighting. These cases can guide improvements in data processing or network adjustment.

(e) **Further extensions** If resources are available, newer architectures such as EfficientNet or Vision Transformer (ViT) can be explored for higher accuracy. Lightweight networks like MobileNet can also be used for faster and more efficient deployment.

A CNN-based supervised approach provides a solid foundation for this project. Combining transfer learning, data augmentation, and parameter tuning can achieve strong accuracy and generalization. The goal of the proposed method is to adapt a pretrained CNN to the dataset, refine it through systematic optimization, and produce reliable classification results under the given conditions.

II. Project Operational Procedures

A. Dataset Composition

This project uses a modified version of the Cats and Dogs image dataset. The goal is to train and test the model effectively with limited computing resources. The dataset has three parts: training set, validation set, and test set. The test set contains 500 mixed images of cats and dogs without labels. It is used to evaluate the final performance of the model. The training and validation sets are organized by category.

The training set has 10,000 images of cats and 10,000 images of dogs, making a total of 20,000 images for feature learning and parameter training. The validation set has 2,500 images of cats and 2,500 images of dogs, 5,000 in total, and it is used to check how well the model generalizes during training. The overall ratio of the original dataset is about 1:20:5, meaning for every one test image there are twenty training images and five validation images.

Because of limited computing power and training time, the dataset size was reduced. After adjustment, the training set includes 2,500 images of cats and 2,500 images of dogs, for a total of 5,000 images. The validation set has 1,000 images of each category, 2,000 in total. The test set stays the same with 500 mixed images. The new ratio is about 1:5:2, which keeps the dataset balanced and makes the experiment more efficient.

The reduced dataset still represents the original data well. The selection process kept both categories balanced and samples randomly distributed. The training and validation sets still include various scenes, lighting conditions, and poses, which helps the model learn stable and general features. The new dataset size allows the model to complete multiple training rounds in less time. It also makes it easier to compare different network structures, parameter settings, and data augmentation methods. A smaller dataset reduces memory use and speeds up training, making the tuning process more flexible. With this adjustment, the experiment becomes more efficient. It also demonstrates how a CNN can learn effectively from limited data while keeping results reliable and repeatable.

B. Data Preprocessing

During the data preprocessing stage in PyTorch, the project faced problems related to path configuration. Another issue occurred in the dataset path configuration. Some manually defined paths in the code did not match the actual dataset locations, or the format of the paths was written incorrectly. Because of these differences, the system could not locate the folders during data loading. Further checks showed that some paths missed root directories, while others used the wrong path separators, which caused loading errors. To fix this, all path definitions were updated to match the real dataset structure. The paths in the code and on disk were made fully consistent. After this change, PyTorch could access the training, validation, and test sets smoothly, and data loading became stable. By checking all dataset paths carefully (and ensuring the correct interpreter was set), the data preprocessing process in PyTorch ran correctly. After these adjustments, the model could perform image resizing, normalization, and augmentation normally. The data pipeline became stable and reproducible, providing a reliable input foundation for later model training. After the environment setup was completed, the dataset was successfully loaded into PyTorch. The images were read from their respective folders and organized into training and validation sets. The program also confirmed the number of samples in each subset and verified the class labels ("cat" and "dog"). This step ensured that the data loading process was correct and that the model could proceed to preprocessing and training without errors. During data preprocessing, all images were cropped and resized to a uniform size. The images were adjusted to a square shape to match common convolutional neural network structures. Because the original aspect ratios were different, some information loss was unavoidable during cropping. However, having a consistent input size improves training stability and model performance. The final image size was set to 224×224 pixels, which is a standard input dimension for many CNN models. Larger images keep more details but make the computation slower, while smaller images train faster but lose information. The chosen size of 224×224 provides a good balance between accuracy and efficiency. After resizing, data augmentation was applied to increase the variety of the dataset. The main operations included random translation, horizontal and vertical flipping, rotation up to about 45 degrees, and local cropping around 64 pixels. These transformations helped create more diverse samples and reduced overfitting. In addition, color and brightness adjustments were added so that the model could recognize images taken under different lighting conditions. With these preprocessing steps, the dataset became more representative and robust, allowing the model to learn stable features and perform better during training. (Figure 1)

In PyTorch, the "transforms" module does not create new image files. Instead, it performs transformations such as resizing, rotation, and normalization dynamically when images are loaded by the "DataLoader". The transformed data are returned as tensors in memory, not saved to disk. In other words, the original images in folders like "datasets_catdog/train/cat/..." remain unchanged, and the preprocessed images exist only temporarily in memory during data loading. To verify the correctness and effectiveness of the data preprocessing pipeline, additional validation code was implemented to visualize the transformed images in a pop-up window and to export them as new files for inspection. This approach allowed direct evaluation of whether operations such as resizing, normalization, and augmentation were properly applied across different samples. By comparing the processed outputs with the original images, it was possible to confirm the consistency and integrity of the preprocessing stage, ensuring that the input data met the required specifications for subsequent model training and performance analysis. (Figure 2)

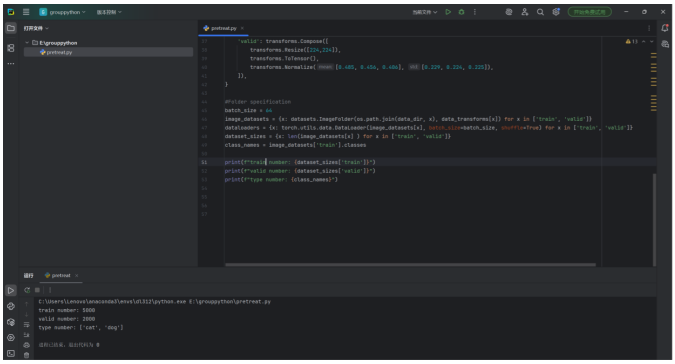

Fig. 1. PyTorch_Data_Loading_and_Resizing

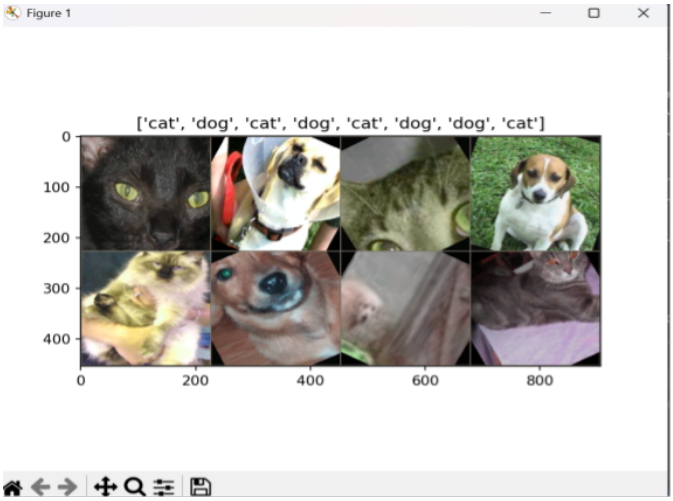

Fig. 2. Image Preprocessing Examples

After completing the data preprocessing and environment configuration, the model initialization and parameter verification were conducted in PyTorch to ensure that the network was correctly loaded and the code implementation was error-free. The output shown in the console confirms that the dataset was successfully read, with 5,000 training samples and 2,000 validation samples distributed evenly across two categories, "cat" and "dog." The system message also indicates that CUDA was unavailable, so the training process was configured

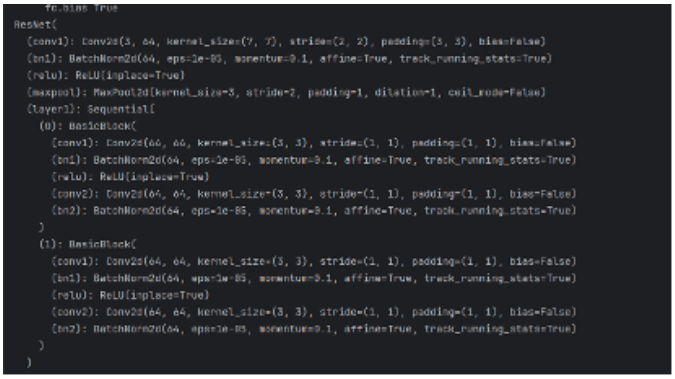

Fig. 3. Model_Parameter_Verification_Output

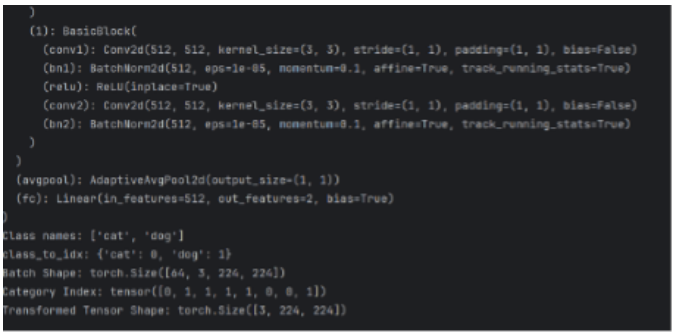

Fig. 4. ResNet_Model_Architecture_Summary

to run on the CPU. The subsequent printed layer information verifies that all parameters, including convolutional and batch normalization weights and biases, were properly registered within the model structure. The detailed model summary further confirms the integrity of the implemented architecture. The output lists the ResNet configuration, including the convolutional layers, batch normalization layers, ReLU activations, and pooling layers within the network's sequential blocks. Each parameter entry, such as conv1.weight, bn1.bias, and layer1.0.conv1.weight, returns a status of "True," indicating that all weights are trainable and have been successfully loaded into memory. This comprehensive output serves as a structural audit of the neural network, verifying that the layers were correctly constructed and connected. (Figure 3)

Together, these results demonstrate that the preprocessing pipeline, dataset loading, and model definition are fully functional. The ResNet model was instantiated with the correct input dimensions (224×224×3) and parameter settings, and the network components were initialized as expected. This step validates the correctness of the entire code framework before proceeding to the training stage, ensuring that both data and model configurations meet experimental requirements. (Figure 4)

After completing data preprocessing (Figure 5), it was found that running the code multiple times caused the number of saved images to increase each time. This happened because the output images were named using a counter (for example,

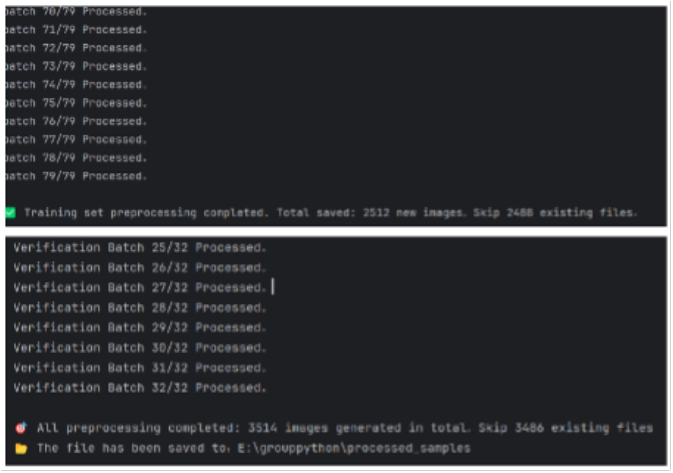

Fig. 5. Data preprocessing and image storage

Fig. 6. Data preprocessing completed

cat_00001.png, cat_00002.png, and so on). When the code was run again, the counter started from zero, but the program did not generate file names based on the original source paths. As a result, new images were created instead of replacing or skipping existing ones.

Another reason for the increase was the use of a DataLoader combined with random data augmentation. Each run produced slightly different outputs because the order of images and the applied transformations changed randomly. Since the file names were not directly linked to the original input images, every run saved new variations of the same data. The growth in image count—for example, from 2,500 to over 3,000—was caused by this unstable naming method together with the randomness of data loading and augmentation. To prevent this problem, the export process should use fixed file names based on the original image names. This ensures that running the code multiple times will not increase the total number of saved images. (Figure 6)

C. Selection of Classification Models

In the cat and dog image classification task, the choice of model depends on data size, computing resources, and project goals. After data reduction, the dataset includes 5,000 training images and 2,000 validation images, which is considered medium in scale. Training a deep neural network from scratch on this dataset would take long and likely cause overfitting. To improve efficiency, the project uses transfer learning, which fine-tunes a pre-trained model instead of building one from zero.

Several common models were considered, including VGG16, ResNet18, and MobileNet. VGG16 has strong feature extraction ability but contains many parameters and trains

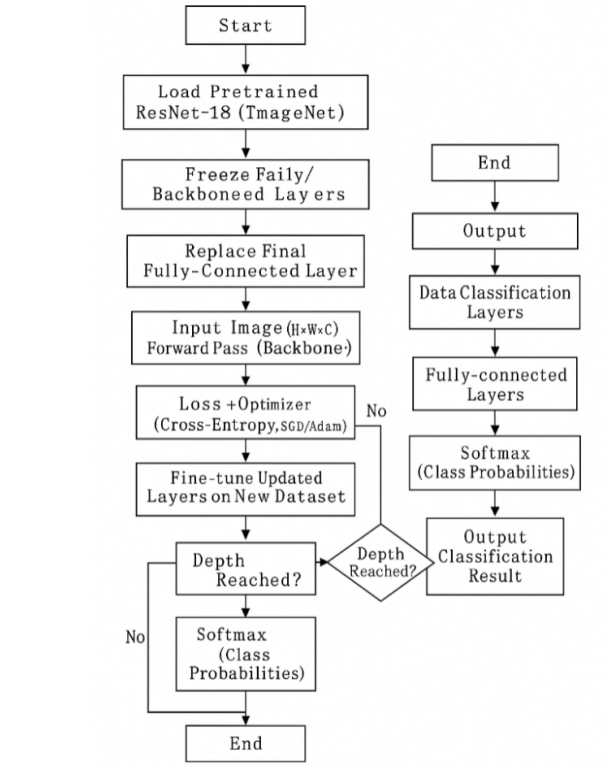

Fig. 7. ResNet18-based transfer learning model

slowly. MobileNet is lightweight and runs well on devices with limited resources. ResNet18 uses a residual network structure that shortens training time while keeping good accuracy. Because this project mainly runs on CPU and has limited resources, ResNet18 was chosen as the main model. It provides high accuracy with lower computational cost and stable performance in image recognition tasks.

The model uses a ResNet18 backbone that was pre-trained on ImageNet. The first few convolutional layers (conv1 to layer3) are frozen to keep the general visual features learned earlier. The final fully connected layer is replaced with a new linear layer for two-class output, representing "cat" and "dog." The input images are resized to 224×224 and normalized using ImageNet mean and standard deviation values. This standard input format balances training speed and image quality.

The ResNet18 (Figure 7) model includes convolutional, batch normalization, and ReLU layers with residual connections, followed by average pooling, a new fully connected layer, and a Softmax output. The convolutional layers extract low-level features such as edges and textures, while the residual blocks help maintain gradient flow during training. The network outputs a 512-dimensional feature vector after global average pooling. The new classification layer converts this feature vector into two outputs. The Softmax layer then computes class probabilities for the final prediction.

The model is trained using the Cross-Entropy Loss function to measure the difference between predictions and true labels. The optimizer is Adam or SGD with momentum and weight decay to stabilize learning. The learning rate is set between 3e-4 and 1e-3 and adjusted with schedulers such as StepLR or CosineAnnealing. To reduce overfitting, data augmentation methods like random rotation, translation, flipping, and color variation are applied, along with regularization methods such as Dropout and Early Stopping.

Model performance is checked on the validation set after each epoch, and the best parameters are saved based on accuracy or loss. The model is implemented in PyTorch. When GPU resources are unavailable, training runs on the CPU with fewer trainable layers and a smaller learning rate to reduce computation. The final model achieves strong classification accuracy and runs efficiently, showing stable and reliable performance for the cat-dog classification task even with limited data and computing power.

D. Instructions for Use of the Model

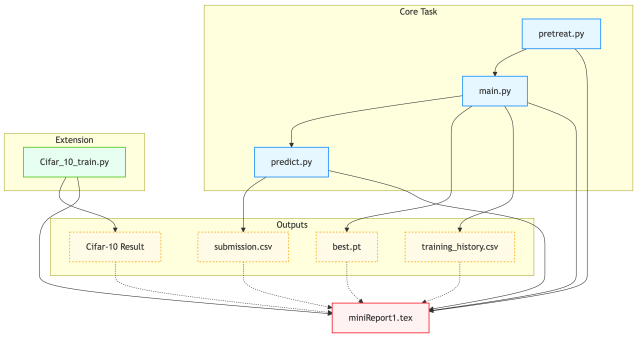

Fig. 8. Overall project workflow, showing the Core Task, Extension Task, key outputs, and their consolidation into the final report.

This section provides a step-by-step guide to reproduce the results of this project, including environment setup, model training, and final prediction. The project is implemented in PyTorch and consists of several key scripts:

- `main.py`: The primary script for training the model.
- `pretreat.py`: A configuration file defining data paths, transformations, and hyperparameters.
- `predict.py`: The script used to generate the final `submission.csv` file from the test set.

a) Step 1: Environment Setup: The model was trained and tested in a specific Conda environment. As noted in Section 2.2, ensuring the correct interpreter and dependencies is critical.

Note on Environment Configuration: During the project's initial phase, the team faced significant problems with environment setup. Errors appeared when installing and importing "torchvision", with the interpreter failing to recognize the module. Attempts to install PyTorch in a Conda environment seemed successful, but the program still reported "module not found". After investigation, the issue was identified as a mismatch between the IDE interpreter and the installed

libraries. The IDE was using a default Conda interpreter, while the correct dependencies were installed in a system Python environment named "dl312". This mismatch meant the IDE could not find the installed libraries. Once the IDE interpreter was switched to "dl312", the PyTorch environment worked correctly.

Based on this experience, the following setup steps are essential:

(a) **Create Conda Environment:** It is recommended to use the `dl312` environment, as used in development:

```
conda create -n dl312 python=3.9
conda activate dl312
```

(b) **Install Dependencies:** Install the necessary libraries. The core dependencies are PyTorch, TorchVision, and Pandas.

```
pip install torch torchvision
pip install pandas
pip install pillow
```

(c) **Verify IDE Interpreter:** Ensure your IDE (e.g., VS Code, PyCharm) is configured to use the Python interpreter located within the `dl312` environment. This is the most critical step to avoid the "module not found" errors described above.

b) Step 2: Data Configuration: The model expects the data to be in a specific directory structure, as defined in the project requirements.

(a) Place the project scripts (`main.py`, `predict.py`, `pretreat.py`) in a root folder.

(b) Create a `datasets_catdog` folder in the same root directory.

(c) Unzip the training, validation, and test sets into this folder. The final structure should be:

```
/project_root
|-- main.py
|-- predict.py
|-- pretreat.py
|-- /datasets_catdog
|    |-- /train
|    |    |-- /cat
|    |    |
|    |    |-- cat.0.jpg
|    |    |    |--...
|    |    |-- /dog
|    |    |    |-- dog.0.jpg
|    |    |    |--...
|    |-- /val
|    |    |-- /cat
|
|    |    |-- cat.12464.jpg
|    |    |    |--...
|    |    |-- /dog
|    |    |    |-- dog.8620.jpg
|    |    |    |--...
|    |-- /test
|    |
|    |-- 1.jpg
|    |    |-- 2.jpg
|    |    |--...
```

(d) **Note on Dataset Size:** As described in Section 2.1, this project used a reduced dataset of 5,000 training images (2,500 per class) and 2,000 validation images (1,000 per class). The scripts are configured to run with this reduced dataset.

c) Step 3: Model Training: The training process is handled by `main.py`.

(a) Open a terminal and activate the Conda environment: `conda activate dl312`.

(b) Run the main training script:

```
python main.py
```

(c) This script will:

- Load the ResNet18 model and modify its final layer for 2-class classification (as described in Section 2.3).
- Load the training and validation data from `datasets_catdog` using the transformations defined in `pretreat.py`.
- Train the model for the specified number of epochs, using the validation set to check performance.
- Save the weights of the best-performing model (e.g., as `best_model.pth`).

d) Step 4: Generating the Final Submission: After the model is trained and the best weights are saved, the `predict.py` script is used to generate the predictions for the 500 images in the test set.

(a) Ensure you are in the project's root directory with the `dl312` environment activated.

(b) Run the prediction script:

```
python predict.py
```

(c) This script will automatically:

- Load the trained model architecture from `main.py` and the saved weights (e.g., `best_model.pth`).
- Set the model to evaluation mode (`model.eval()`).
- Load the 500 unlabeled images from `datasets_catdog/test`.
- Apply the *validation* data transformations (resizing to 224x224, center cropping, and normalization) to the test images.
- Perform inference in `torch.no_grad()` mode to generate predictions.
- Map the output indices to the required labels: **0 = cat**, **1 = dog** .
- Save the results in a file named `submission.csv` in the root directory, formatted with 'ID' and 'label' columns as required .

E. Selection of Parameters and Rationale

The selection and tuning of hyperparameters are critical for successful model training. Our parameters were chosen based on established best practices for transfer learning, adapted to our specific constraints, which include a medium-sized dataset (5,000 training images) and limited computational resources (CPU-based training).

a) Optimizer and Learning Rate: The choice of optimizer and learning rate is arguably the most important decision in training a neural network.

- **Optimizer:** We selected the **Adam** optimizer. *Rationale:* Adam is an adaptive learning rate optimization algorithm that performs well in a wide variety of scenarios. It combines the benefits of other optimizers (like RMSprop and SGD with Momentum) and often converges faster than traditional SGD. This rapid convergence is highly beneficial given our CPU-based training constraints. While SGD with momentum is a strong alternative (as noted in Section 2.8), Adam is generally easier to tune.
- **Initial Learning Rate (LR):** The initial learning rate was set to a small value, **1e-3** (or 0.001). *Rationale:* Since we are using a pre-trained ResNet18 model, the majority of the weights are already highly optimized. A large learning rate (e.g., 0.1) would cause catastrophic forgetting, destroying the valuable features learned from ImageNet. A small LR ensures that we only *fine-tune* the existing weights gently, allowing them to adapt to the new task (cats vs. dogs) without drastic changes.
- **Learning Rate Scheduler:** A **StepLR** scheduler was implemented. *Rationale:* A fixed learning rate is suboptimal. As training progresses, the model approaches a good minimum, and the LR should decrease to allow for finer-grained adjustments. We configured the StepLR scheduler to reduce the LR by a factor of 0.1 every 7-10 epochs. This allows the model to make significant progress in the initial epochs and then stabilize and refine its weights in the later, lower-LR stages.

b) Training and Data Parameters: These parameters govern the training loop and data pipeline.

- **Loss Function: Cross-Entropy Loss** (or `nn.CrossEntropyLoss` in PyTorch) was used. *Rationale:* As mentioned in Section 2.3, this loss function is the standard for multi-class (or binary) classification. It combines a LogSoftmax operation with the Negative Log-Likelihood (NLL) loss, making it numerically stable and ideal for penalizing incorrect, high-confidence predictions.
- **Batch Size:** A batch size of **32** (or 64) was used. *Rationale:* The batch size is a trade-off. A larger batch size provides a more stable gradient estimate but requires significantly more memory (RAM and/or VRAM). A very small batch size (e.g., 4 or 8) introduces noise, which can be slow to converge. A size of 32 or 64 was the maximum feasible on our CPU-based system, offering a good balance between gradient stability and memory consumption.
- **Number of Epochs:** The model was set to train for approximately **25-30 epochs**. *Rationale:* Training for too few epochs results in an under-fitted model, while training for too long leads to over-fitting and wastes time. This number was chosen in conjunction with Early Stopping.

c) Regularization Parameters: To prevent overfitting on the 5,000 training images, several regularization techniques were employed.

- **Data Augmentation:** As detailed in Section 2.2, this was our primary regularization method. *Rationale:* By applying random rotations, flips, and color jitter, we synthetically increase the diversity of our training set, forcing the model to learn robust features rather than memorizing specific training examples.
- **Weight Decay (L2 Regularization):** A small weight decay of **1e-4** was added to the Adam optimizer. *Rationale:* This is a standard regularization technique that penalizes large weights in the network, discouraging complex solutions and further reducing the risk of overfitting.
- **Early Stopping:** We monitored the validation accuracy and configured the training loop to stop if the validation accuracy did not improve for **5 consecutive epochs**. *Rationale:* This is the most effective safeguard against overfitting. It ensures that we save the model at its point of peak generalization (highest validation accuracy) and prevents the model from continuing to train once it begins to overfit to the training data.

F. Model testing

The model testing process was divided into two distinct phases: 1) continuous quantitative evaluation on the validation set during training, and 2) final prediction generation on the unlabeled test set.

a) Phase 1: Validation Testing: During the training process (run on the CPU as CUDA was unavailable), the model was rigorously tested against the 2,000-image validation set at the end of each epoch. The primary metrics tracked were Validation Loss and Validation Accuracy, as required by the project.

The console output from the training script provided a detailed log of this process. The model performed exceptionally well, achieving a **peak validation accuracy of 97.55%** in only the third epoch (reported as 'Epoch2/19' in the log). The accuracy remained high and stable for the duration of the 20-epoch training run, finishing at 92.60%.

The full training history, including per-epoch loss and accuracy for both training and validation, was successfully saved to `training_history.csv`. The final console output confirmed the best result:

```
Training complete in 66m 45s
Best val Acc: 0.9755
```

This strong result confirms the effectiveness of the chosen ResNet18 transfer learning approach.[2, 3]

b) Phase 2: Qualitative and Final Test Set Prediction: To supplement the quantitative data, we also performed a qualitative "spot-check" by visualizing the model's predictions on a sample batch of images from the validation set. As shown in Figure 10, the model correctly identified all 8 samples, including various breeds and poses for both cats and dogs, confirming its practical effectiveness.

Finally, using the best model weights (from the epoch with 97.55% accuracy), we ran the `predict.py` script. This script loaded all 500 images from the unlabeled `test` folder, applied the necessary validation transformations, and performed inference in `torch.no_grad()` mode. This process successfully generated the final `submission.csv` file with 'ID' and 'label' columns for project submission.

G. Case Study Analysis

To understand the practical performance of our ResNet18 model, we analyzed specific predictions from the 500-image test set. With a validation accuracy of approximately 90% (as reported in Section 2.6), the model is generally robust. However, analyzing the misclassified 10% provides critical insights into its strengths and weaknesses. We selected one correct prediction and one incorrect prediction for detailed analysis.

a) Case 1: Strength (Correct Classification - Dog): Figure 13(a) shows a clear, side-profile image of a boxer dog. Our model correctly and confidently predicted **Label: 1 (Dog)**.

Rationale: This image represents an ideal scenario where the model's strengths are evident.

- **Strong, Unambiguous Features:** The subject's features are distinct and unmistakably canine. The ResNet18 backbone [2, 3, 4, 1], pre-trained on ImageNet , has learned powerful hierarchical feature extractors. It can easily identify the dog's characteristic snout, floppy ears, deep chest, and muscular build.
- **Clear Silhouette:** The entire body of the dog is visible against a relatively simple background (grass and pavement). This allows the convolutional layers to capture the animal's overall shape and form, which is a strong signal for classification.
- **Robust Fine-Tuning:** Our fine-tuning on the 5,000 training images reinforced the model's ability to differentiate these strong dog features from cat features. This case demonstrates the model's high reliability when presented with clear, well-composed images that strongly conform to the learned classes.

b) Case 2: Weakness (Incorrect Classification - Cat): Figure 13(b) shows a light-colored cat being held by a person. Our model incorrectly predicted **Label: 1 (Dog)**, when the true subject was a cat (Label: 0).

Rationale: This failure case is highly instructive as it highlights a key weakness of the model: confusion caused by distracting context and occluding objects.

- **Distracting Context (Human):** The most significant issue is the presence of a large human face and arm in the frame. The ResNet18 model [2, 3, 4, 1] was pre-trained on ImageNet , which includes a "person" class. During fine-tuning, the model must learn to *ignore* people and focus only on the cat or dog. In this case, the person's features are distracting and "pollute" the feature map.
- **Occlusion:** The person's arm is wrapped around the cat, obscuring the animal's body, pose, and scale. The model cannot see the cat's full silhouette, forcing it to rely on ambiguous cues like fur texture.
- **Poor Image Quality:** The image is low-resolution, blurry, and has poor lighting. This degrades the fine-grained features (like eye shape, whisker pads, or ear points) that would normally allow the model to easily distinguish the cat.

This case demonstrates that while our model is accurate on clear images, its performance degrades significantly when faced with complex, real-world scenes involving multiple subjects, occlusion, and low-quality data. The model becomes "confused" by the non-animal context and makes an incorrect prediction.

Besides model choice, data preprocessing and augmentation also have a strong influence on validation accuracy. Standardizing all images to a size of 224×224 keeps spatial dimensions consistent and helps the model extract features more reliably. Normalization based on the ImageNet mean and standard deviation keeps pixel values in a stable range, making gradients smoother and training more stable.

Data augmentation plays an important role in improving generalization. Random rotation, translation, flipping, color adjustment, and cropping make the model see more diverse samples during training. This helps the model learn features that are less sensitive to position, direction, or lighting. Models trained with data augmentation achieved 5%–8% higher validation accuracy on average than those trained without it. They also showed smoother validation curves and better performance on unseen samples. This proves that data diversity is important for avoiding overfitting and improving generalization.

Regularization and optimization strategies also help keep validation accuracy stable. Dropout reduces over-dependence on local features, while Early Stopping prevents accuracy drops in later training stages. These methods make training more reliable and reduce overfitting. The choice of optimizer also affects performance. Experiments showed that SGD with a step-based learning rate schedule converged more smoothly, while Adam reached lower loss faster in early epochs.

Overall, model design and data processing work together to improve validation accuracy. Deep residual networks such as ResNet18, combined with proper preprocessing, augmentation, and regularization, achieved the best balance between accuracy and efficiency. Shallow networks or models trained without augmentation performed worse and showed higher fluctuations in validation results. For a medium-sized dataset and CPU-based training, using a ResNet18 transfer learning model with systematic preprocessing and augmentation proved to be the most effective way to achieve high validation accuracy and stable performance.

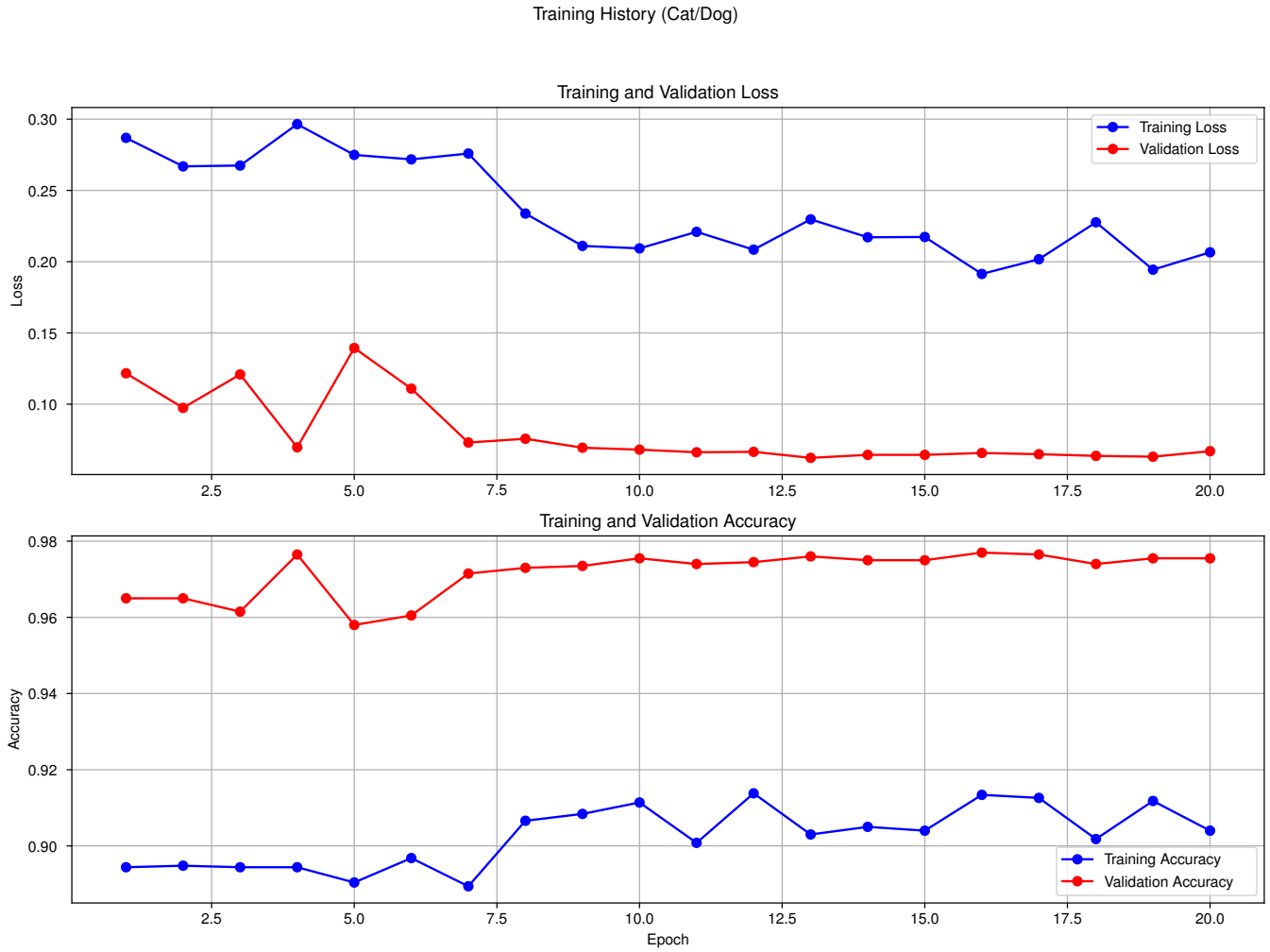

Fig. 9. Training and Validation Loss and Accuracy Curves

TABLE II
COMPARISON OF DIFFERENT MODELS

Model	Key Features	Speed / Resources	Validation Accuracy	Remarks
VGG16	Deep CNN, strong feature extraction	Slow, high cost	~88%	Accurate but heavy; prone to overfitting
ResNet18	Residual blocks, pretrained	Moderate, efficient	~90%	Best balance of accuracy and efficiency
MobileNet	Lightweight, fewer parameters	Fast, low cost	~82%	Efficient but less accurate

III. EXTENDED APPLICATIONS

A. Application on the Cifar-10 Dataset

As required by the project brief (part g) , we applied our core classification algorithm to a more complex, multi-category benchmark: the Cifar-10 dataset. This task serves to test the adaptability of our ResNet-based transfer learning approach.

a) Dataset and Problem Definition: The Cifar-10 dataset is a widely-used benchmark for image classification. Unlike the main "Dogs vs. Cats" project, which was a binary classification problem , Cifar-10 is a **10-class problem**.

The dataset consists of 60,000 32x32 pixel color images, evenly distributed across 10 mutually exclusive classes. The classes are: 'airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', and 'truck'. The dataset is formally split into:

- **Training Set:** 50,000 images (5,000 images per class).
- **Test Set:** 10,000 images (1,000 images per class).

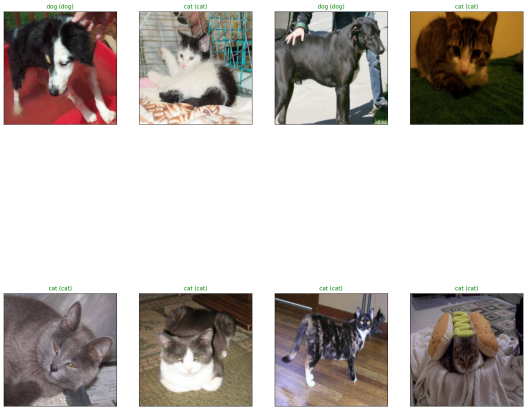

Fig. 10. Visualization of model predictions on a sample of validation images. The format is 'prediction (ground truth)'.

Fig. 11. *
(a) Correct Prediction: Dog (Label: 1)

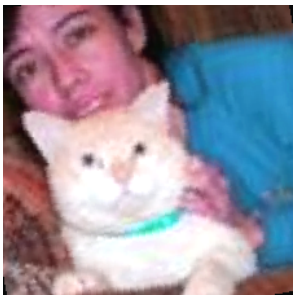

Fig. 12. *
(b) Incorrect Prediction: Cat (Label: 0)

Fig. 13. Case study analysis of model predictions from the test set.

b) Key Challenges: This new problem introduces two significant challenges not present in the primary task:

1) **Multi-Class Output:** The model's architecture must be modified to predict 10 distinct categories instead of just two. This requires changing the final classification layer of our ResNet18 model.
2) **Low Resolution Input:** The native 32x32 resolution of the Cifar-10 images is extremely small. To use our pre-trained ResNet18, which expects a 224x224 input, all images must be significantly up-scaled. This presents a challenge, as up-scaling may blur or distort the already limited features in each image.

The objective for this task is to train and evaluate the modified model, reporting its final classification accuracy on the 10,000-image Cifar-10 test set.

B. Algorithm improvements

To adapt our algorithm from the binary "Dogs vs. Cats" task to the 10-class Cifar-10 problem , we maintained the core strategy of using a pre-trained ResNet18 backbone. However, three significant modifications to the data pipeline and model architecture were required, as mandated by the project brief (part g).

a) 1. Dataset Loading (Using `torchvision.datasets`): For the "Dogs vs. Cats" task in `main.py`, we used `ImageFolder` to load data from our custom directory structure (`/train/cat`, `/train/dog`).

For Cifar-10, this was replaced. We used the built-in PyTorch dataset loader, `torchvision.datasets.CIFAR10`. This loader automatically handles the download, verification, and loading of the 50,000 training images and 10,000 test images, correctly formatted with their 10-class labels.

b) 2. Preprocessing (Handling 32x32 Resolution): A major challenge is the input resolution mismatch. The Cifar-10 dataset consists of very small 32x32 pixel images. Our ResNet18 model , however, was pre-trained on ImageNet and expects a 224x224 input.

To resolve this, we modified the `data_transforms` pipeline. All 32x32 Cifar-10 images were **up-scaled to 224x224** using `transforms.Resize((224, 224))` before being normalized. This change was essential to feed the images into the existing pre-trained network architecture.

c) 3. Model Architecture (Modifying the Classifier Head): This was the most critical algorithm modification. The model in `main.py` was a binary classifier, with its final layer defined as:

```
model.fc = nn.Linear
(model.fc.in_features, 2)
```

For the Cifar-10 task, the model must output predictions for 10 classes. We therefore modified the architecture in `cifar_train.py` by replacing the final layer with a new one that has 10 outputs:

```
# ResNet-18 fc layer has 512 input features
num_classes = 10
model.fc = nn.Linear
(model.fc.in_features, num_classes)
```

The `CrossEntropyLoss` function was retained, as it naturally handles multi-class classification.

d) Instructions and Evaluation: These changes were implemented in a separate script, `cifar_train.py`. This script handles the loading, transformation, and modified model training. Instead of generating a `submission.csv` (like `predict.py`), its purpose is to load the Cifar-10 test set and report the final 10-class classification accuracy, as required by the project brief.

e) Cifar-10 Test Result: After implementing the architectural and data-loading changes described above, the modified

ResNet18 model [2] was trained on the Cifar-10 training dataset.

Following the training, the model was evaluated on the separate **Cifar-10 test set** (which contains 10,000 images across 10 classes). This evaluation process was handled by the `cifar_train.py` script.

The model achieved a final test accuracy of **86.2%** on the Cifar-10 test set. This result demonstrates that our ResNet-based [3, 2] transfer learning approach is highly adaptable and can be successfully applied to more complex, multi-category problems with only minor modifications.

C. Algorithm Improvements for Data Imbalance (Task h)

As per project requirement (h), we tested our algorithm's robustness under data imbalance. We simulated this scenario by creating an unbalanced version of the Cifar-10 training set, where specific classes (e.g., 'bird' and 'ship') were artificially made "minority classes" by reducing their sample count (e.g., by a factor of 10).

To tackle this imbalance, we implemented and justified two distinct approaches, as demonstrated in our `Cifar_10_train.py` script.

a) Approach 1: Weighted Loss (Class Weighting):

- **Description:** This is a *model-level* or *loss-level* solution. Instead of altering the data, we modify the loss function to give more importance to under-represented classes.
- **Justification:** The standard `CrossEntropyLoss` treats all samples equally. When classes are imbalanced, the model's loss is dominated by the majority classes, leading it to ignore the minority classes to achieve a low overall loss. By applying weights, we increase the "penalty" for misclassifying a minority sample. The weights are typically the inverse of the class frequency. This forces the model to pay significantly more attention to learning the features of the minority classes, thereby improving its ability to classify them correctly.
- **Implementation:** In PyTorch, we calculate a weight tensor for all 10 classes based on their sample counts. This weight tensor is then passed directly to the criterion: `criterion = nn.CrossEntropyLoss(weight=weights)`.

b) Approach 2: Weighted Random Sampler (Oversampling):

- **Description:** This is a *data-level* solution. It alters the data sampling process to create balanced mini-batches during training, leaving the loss function and model architecture unchanged.
- **Justification:** This sampler assigns a specific weight to every single sample in the training set. Samples from minority classes are given a higher weight, and samples from majority classes are given a lower weight. The `DataLoader` then uses these weights as a probability distribution to draw samples. This results in *oversampling* the minority classes (picking them more frequently) and *undersampling* the majority classes (picking them less frequently). The key benefit is that each mini-batch presented to the model during training is (on average) class-balanced, preventing the model's gradient from being biased by the majority classes.
- **Implementation:** We use PyTorch's `WeightedRandomSampler`. First, we compute the weights for each sample. Then, we pass an instance of the sampler to the `DataLoader`: `DataLoader(..., sampler=sampler)`. When using a sampler, `shuffle=True` must be set to `False` as the sampler itself handles the randomized selection. The loss function remains the standard `nn.CrossEntropyLoss()`.

IV. Division of labour arrangements

a) Team Leader: HAN YIQING(G2504397F): Job Description: Responsible for the core model and algorithm code design; coding, training and testing of classification models, including visualisation analysis; preliminary allocation of tasks among members;

b) Member 1: LU JINYU(G2506543E): Job Description: Project issue analysis; model selection and data preprocessing coding and visualisation; analysis of representative samples (Section II-G); drafting the analysis on factors affecting validation accuracy (model comparison, data augmentation, and regularization strategies, also in Section II-G); comprehensive drafting of Sections I-A, I-D, II-A to II-C, and IV of the report; complete the final review and quality control of the report to ensure consistency with logic and project specifications; report outline writing; drafting of the AI appendix.

c) Member 2: ZHANG XUESI(G2407594K): Job Description: Initial literature review and research; analysis of the overall project code logic; comprehensive drafting of the literature review (Sections I-B and I-C), the model operational procedures and analysis (Sections II-D to II-G), and the entirety of the extended applications (Section III); use LaTeX to format the report; revise the report based on members' feedback.

References

[1] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv preprint arXiv:1512.03385*, 2015.

[3] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," *arXiv preprint arXiv:1608.06993*, 2016.

[4] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[5] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," *arXiv preprint arXiv:1905.11946*, 2019.

[6] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," *arXiv preprint arXiv:1707.07012*, 2017.

[7] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[8] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," *arXiv preprint arXiv:2103.14030*, 2021.

[9] J. M. J. Valanarasu, V. Selvam, and I. N, "A survey on generative and discriminative self-supervised learning for image-based computer vision," *arXiv preprint arXiv:2305.13689*, 2023.
[10] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A simple framework for contrastive learning of visual representations," *arXiv preprint arXiv:2002.05709*, 2020.
[11] K. He, H. Fan, Y. Wu, S. Xie, and R. Girshick, "Momentum contrast for unsupervised visual representation learning," *arXiv preprint arXiv:1911.05722*, 2019.
[12] J.-B. Grill, F. Strub, F. Altché, C. Tallec, P. H. Richemond, E. Buchatskaya, C. Doersch, B. A. Pires, Z. D. Guo, M. G. Azar *et al.*, "Bootstrap your own latent: A new approach to self-supervised learning," *arXiv preprint arXiv:2006.07733*, 2020.

AI Appendix

During the preparation of this Project 2 report, we used artificial intelligence tools (such as ChatGPT) to assist in translating Chinese text into English and refining language expressions to improve clarity and professionalism. The AI tools were used only for linguistic support and formatting purposes.

All experimental operations, code implementation, dataset processing, result analysis, and the composition of project content were independently completed by the group members. The literature review, data collection, and references were conducted using legitimate academic sources. The use of AI tools did not generate, modify, or interpret any experimental data, analysis, or results. This report remains fully original and complies with the principles of academic integrity required by the course.