Mini Project (Option 2): Dogs vs. Cats

Director: Wen Bihan

Group Member 1: Zhang Zezheng (G2402715A)
`zezheng001@e.ntu.edu.sg`

Group Member 2: Zhang Na (G2402918G)
`zhan0680@e.ntu.edu.sg`

Group Member 3: Wang Meng (G2403580A)
`meng010@e.ntu.edu.sg`

School of Electrical and Electronic Engineering, NTU

April 18, 2025

Contents

This report presents a comprehensive study of an image classification system for the Dogs vs. Cats problem, guided by the following tasks:

1. **Literature Survey:**
   - (a) Review tutorials and literature to understand the domain and problem settings.
   - (b) Identify key recent papers via keyword search and top conferences.
   - (c) Select and summarize top-performing works in detail.
   - (d) Justify and select a suitable baseline method with proposed improvements.
2. **Dataset Preparation:**
   - (a) Specify the size of training and testing sets and detail pre-processing steps.
3. **Model Design and Implementation:**
   - (a) Build and describe the classifier, architecture, and training setup.
   - (b) Include source code and ensure result reproducibility.
4. **Hyperparameter Tuning:**
   - (a) Justify selected model parameters and training settings.
5. **Model Evaluation:**
   - (a) Report accuracy on validation and test sets.
   - (b) Submit test predictions as a CSV file.
6. **Model Analysis:**
   - (a) Analyse correct and incorrect test predictions.
   - (b) Highlight model strengths and weaknesses.
7. **Model Variations:**
   - (a) Discuss how model and data choices affect performance.
8. **Multi-Class Classification (CIFAR-10):**
   - (a) Apply and modify classifier to solve CIFAR-10 classification.
   - (b) Compare approaches and report testing results.
9. **Data Imbalance in CIFAR-10:**
   - (a) Address class imbalance with at least two justified strategies.
10. **Respective Contribution:**

1 Literature Survey

1.1 Domain Understanding

1.1.1 The Task: Cat and Dog Classification

Image classification is a basic task in the field of computer vision[1]. Its purpose is to assign predefined labels or categories to images according to their visual content. This function is essential for automating visual recognition tasks that previously required human intervention. Among many image classification problems, cat dog classification is a classic and representative benchmark. It is not only an entry-level project for novices in deep learning, but also a test ground for evaluating the performance of new classification algorithms. Due to the machine learning competition triggered by the data set of ASIRRA (animal specifications image recognition for limiting access) in 2013, this task has gained great popularity[2].

Although the classification of cat and dog is simple, it contains many inherent challenges of image recognition. These include changes in object size (cats and dogs appear in different sizes), different postures (standing, lying, running, jumping), changing lighting conditions, messy background and occlusion. In addition, both cats and dogs contain many breeds with significant visual differences (such as Chihuahua and Dadan), while some breeds in the same category (such as different short haired cat breeds) may look very similar. This essentially ties tasks to fine-grained visual classification (FGVC), which focuses on differentiating fine-grained subcategories in a broader category. Successful cat and dog classification requires models that can capture and distinguish these fine visual features.

1.1.2 The Rise of Deep Learning in Image Recognition

In the past decade, deep learning (DL), especially convolutional neural network (CNN), has completely changed the field of computer vision[3]. Unlike traditional methods that rely on manual features (such as sift or hog), CNN automatically learns hierarchical feature representation directly from the original pixel data. The shallow layer usually learns low-level features such as edges and textures, while the deep layer combines these features to learn more abstract patterns and concepts (for example, "cat ears", "dog nose"). This end-to-end learning method greatly improves the performance of image recognition task, which is better than the traditional method.

The success of the deep learning model, especially the breakthrough in large-scale image data sets such as ImageNet, has promoted its deployment in many practical applications, including automatic driving (recognition of vehicles, pedestrians, traffic signals), face recognition, medical image analysis and various consumer applications (such as automatic album classification). Deep learning research, especially in the field of computer vision, continues to grow exponentially.

1.1.3 Foundational Concepts in Deep Learning for Image Classification

Understanding different learning paradigms is vital for choosing appropriate algorithms. There are several problem definitions and learning paradigms following.

The first one is supervised learning. Learning from tagged data, where each input has a corresponding correct output (label). The goal is to learn the mapping function from input to output. It is widely used in classification tasks, such as cat/dog recognition. Then is unsupervised learning. Learning from unmarked data and discovering hidden structures or patterns without clear guidance. Tasks include clustering and dimensionality reduction. It is helpful for exploratory analysis and feature learning. The accuracy may be lower than the monitoring method. There is also semi-supervised learning. Use a small amount of tagged data and a large amount of unlabeled data. Using unlabeled data to improve the learning of labeled data is usually through pseudo tagging technology. While supervised learning is the most direct approach for cat/dog classification due to available labeled datasets[4], unsupervised (especially self-supervised) and semi-supervised methods offer ways to leverage vast unlabeled image data to improve feature extraction and reduce labeling dependency.

For the labeled data, raw data (images) plus meaningful tags or categories ("Cats", "dogs"). Supervised learning is essential to provide basic facts. However, creating high-quality tag data is often expensive and time-consuming, and may be biased or inaccurate. For the unlabeled data, raw data without any tags is rich and easier to obtain. For unsupervised, semi supervised, and self supervised learning (SSL). SSL methods (e.g., MAE, comparative learning) learn powerful representations from unmarked data itself by solving excuse tasks[5], and then fine tune downstream tasks, such as using the smallest labeled data for classification. Leveraging unlabeled cat/dog images (and other images) via SSL for pre-training, followed by fine-tuning on labeled data, is a powerful strategy balancing labeling costs and model performance.

Closed-set recognition is assumed that all test samples belong to classes known during training. The source and target domains share exactly the same label space. The model only needs to distinguish between known classes. It is unrealistic for many applications because unknown objects will inevitably be encountered and may be misclassified. Open-set recognition acknowledges that test data may contain unknown classes not seen during training. The target domain's classes may only partially overlap or be entirely different from the source. The model must classify known instances accurately *and* identify/reject unknown instances. For a practical cat/dog classifier, an OSR capability is crucial to handle inputs that are neither cats nor dogs (e.g., squirrels, toys), preventing misclassification and improving robustness[6].

For the domain shift, statistical distribution differences between source (Ds) and target (Dt) data. Caused by factors like lighting, pose, image quality, background, camera type. Leads to performance degradation even if the task is identical. Types include covariate shift (P(X) changes, P(Y|X) constant), prior/label shift (P(Y) changes, P(X|Y) constant), and concept shift (P(Y|X) changes). Domain adaptation (DA) is a transfer learning technique to adapt a model trained on a labeled source domain to perform well on a target domain with limited or no labels[5]. It assumes the same task and feature space between the source and target domains but different data distributions. The aim is to reduce domain discrepancy by learning domain-invariant features, often using techniques like Maximum Mean Discrepancy (MMD) minimization or adversarial learning (training a domain discriminator). Types are based on target label availability: Unsupervised Domain Adaptation (UDA - no target labels), Semi-supervised Domain Adaptation (SSDA - few target labels), and Supervised Domain Adaptation (SDA - ample target labels). Domain shift is a common challenge when deploying a cat/dog classifier trained on web images (e.g., Kaggle dataset) to classify user-uploaded phone photos. DA techniques help mitigate this performance drop without requiring extensive relabeling for the target domain.

1.2 Paper Search and Trends

This section reviews research in recent years on using deep learning for cat/dog classification and related fine-grained pet recognition.

1.2.1 Task Characteristics: Challenges and Significance

Cat and dog classification, while seemingly basic, presents complexities that make it a valuable platform for testing advanced deep learning techniques. There are some core challenges.

The first one is fine-grained distinction. Recognizing specific breeds (e.g., using Stanford Dogs or Oxford-IIIT Pet datasets) demands capturing subtle visual differences (fur patterns, ear shape), characteristic of Fine-Grained Visual Classification (FGVC) where inter-class variance is low and intra-class variance is high. The second one is high intra-class variance. Images within the same class ("cat" or "dog," or even a specific breed) can vary greatly due to pose, lighting, viewpoint, occlusion, age, etc. Models need robustness to these variations. The last one is ackground clutter. Pet photos often have complex backgrounds; models must focus on the target animal.

For the research significance, algorithm testbed provides an accessible yet challenging environment for validating new architectures (e.g., Transformers), learning strategies (transfer learning, SSL, contrastive learning), attention mechanisms, and data augmentation techniques. There also some practical applications. For example, accurate pet (and breed) recognition enables lost pet recovery systems, pet health monitoring, smart pet feeders/doors, animal welfare monitoring, and social media tagging.

1.2.2 Recent Research Review

Deep learning continues to evolve, with several trends impacting cat/dog and fine-grained classification.

Convolutional neural network (CNN) is still the cornerstone of image classification. Combining several finely tuned CNN architectures (such as perception V3, perception RESNET V2, nasnet and pnasnet) with feature extraction techniques such as principal component analysis (PCA) and gray wolf optimization (GWO), and then using support vector machine (SVM) for classification, it has been proved to have high accuracy. For example, the classification accuracy of this method for 120 dog breeds on the Stanford dog data set is 95.24%[7].

The dominant position of transfer learning: transfer learning, especially using the model pre trained on large data sets (such as vgg16, mobilenetv2, inception, RESNET), has become a standard practice when processing limited target data. Fine tuning these pre trained networks not only significantly improves the performance of the model, such as improving the classification accuracy from about 91% to more than 95% in the cat dog task[8], but also reduces the training time and data requirements. A subtle understanding of feature transferability is crucial: lower layers typically capture general features, while deeper layers encode task specific representations.

Basic function of data expansion: data expansion plays a key role in improving generalization and reducing over fitting. The technology ranges from basic transformations (such as rotation, flipping, scaling) to more advanced strategies, such as clipping, blending, and clipping blending. In addition, automatic enhancement strategies (such as autoaugment and randaugment) provide adaptive mechanisms to optimize the enhancement pipeline for specific data sets and tasks.

For the self-supervised learning (SSL) applications, the primary objective is to mitigate the dependence on large-scale, manually labeled datasets by learning effective visual representations from unlabeled data through self-supervised learning (SSL) techniques.

There are some common methodologies. MoCo (Momentum Contrast) is a contrastive learning framework that employs a momentum-updated encoder to maintain a dynamic dictionary of negative samples. This method enables scalable and stable training for representation learning from unlabeled data (Kaiming He et al., FAIR). MAE (Masked Autoencoders) is a self-supervised pre-training approach designed for Vision Transformers (ViT), where a significant portion of image patches is randomly masked, and the model is trained to reconstruct the missing content. This masking strategy encourages the model to learn high-level semantic features and has proven highly effective and scalable for ViT pre-training (Kaiming He et al., FAIR). DINOv2 is developed to learn general-purpose visual features that are transferable across tasks without the need for fine-tuning. Pre-trained on a large, curated dataset (LVD-142M) using advanced SSL techniques, DINOv2 achieves state-of-the-art performance across a range of image and pixel-level tasks, including a 96.7% linear evaluation accuracy on the Oxford-IIIT Pets dataset (FAIR). SEER (SElf-SupERvised) is a large-scale SSL model comprising up to 10 billion parameters, trained on vast collections of uncurated images from the internet. SEER emphasizes robustness and fairness, demonstrating strong performance on fine-grained visual classification tasks. For instance, SEER (RegNet10B) attained 85.3% accuracy on the Oxford-IIIT Pets dataset (FAIR).

Self-supervised learning, particularly when combined with large-scale Transformer architectures (e.g., MAE, DINOv2), is emerging as the dominant paradigm for learning universal visual representations. These representations substantially enhance the performance of downstream tasks—especially in fine-grained domains—while significantly reducing the reliance on labeled data.

1.3 Key Works and Group Contributions

1.3.1 Microsoft Research Asia (MSRA): Research Trajectory and Main Contributions

Microsoft Research Asia (MSRA) is one of the top research groups working in computer vision and deep learning. Over the years, MSRA has been consistently working on tackling challenging problems in deep neural networks, such as gradient vanishing or exploding, which previously made training very deep networks difficult. One of the most famous breakthroughs from the MSRA team is the development of deep residual networks, or ResNet. ResNet made it possible to effectively train extremely deep networks, significantly improving their stability and generalization, and became a key milestone in deep learning research.

MSRA regularly publishes high-quality papers that have gained wide recognition across the research community. Their papers are highly cited, especially the original ResNet paper, which is now one of the most cited research papers globally. The team's work has earned several important awards, such as Best Paper awards at top computer vision conferences like CVPR (2009, 2016) and ICCV (Marr Prize, 2017). These achievements clearly demonstrate MSRA's leading role and broad influence in shaping the direction of computer vision and deep learning[9-11].

1.3.2 VGG16: Structured Depth and Foundational Design Principles

The VGG16 architecture, introduced by Simonyan and Zisserman in 2015[12],, represented a major advance in CNN design by demonstrating that the use of small convolutional filters—specifically 3×3—stacked deeply and uniformly can significantly enhance performance on large-scale image classification tasks. The model comprises 13 convolutional layers followed by 3 fully connected layers, and it forgoes architectural novelty in favour of clean, repeated patterns.

One of the key takeaways from the original work is the importance of architectural regularity. Unlike previous architectures such as AlexNet, which employed a variety of kernel sizes and custom configurations, VGG16 emphasized uniformity and simplicity. Through this choice, the authors made it easier to analyze the role of depth systematically, enabling fair comparisons across model variants with 11, 13, 16, and 19 layers. In the 16-layer configuration, each convolutional layer is followed by a ReLU activation and arranged in blocks separated by max pooling layers. This depth allows the network to learn highly hierarchical representations—progressing from edges to textures, to object parts, and finally to object-level concepts.

A striking aspect of the paper is its empirical clarity. Performance gains are shown to arise not from increasing width or filter size, but from deepening the network while keeping the filter size small. Although the parameter count is high—around 138 million in VGG16—the architecture generalizes well on clean, centered images and has been widely adopted in transfer learning workflows. In particular, the intermediate convolutional features extracted by VGG16 have been shown to be transferable to a variety of downstream tasks, such as object detection, style transfer, and fine-grained classification.

While VGG16 has since been surpassed in accuracy and efficiency, it remains an instructive example of how depth and architectural discipline can yield strong performance, especially in scenarios where interpretability and layer-wise feature inspection are valuable.

1.3.3 Deep Residual Learning (ResNet): A Paradigm Shift in Optimization

In response to the degradation problem observed in very deep networks—where adding more layers leads to higher training error—Kaiming He and colleagues at Microsoft Research introduced the ResNet framework in their landmark paper "Deep Residual Learning for Image Recognition" (CVPR 2016). Rather than abandoning depth, the authors redefined how layers should behave within a deep network by introducing the concept of *residual learning*.

The central idea is to let a group of layers learn a residual function $F(x) := H(x) - x$ instead of directly learning the target function $H(x)$. The output of a residual block is then obtained as $H(x) = F(x) + x$. This formulation is motivated by the observation that learning a residual mapping—especially one close to zero—is often easier than learning an unreferenced identity mapping. In extreme cases where the desired mapping is simply $H(x) = x$, the network can learn to push $F(x) \to 0$ by driving the weights in the residual path toward zero.

This concept is operationalized through *identity shortcut connections*, which bypass one or more layers within a residual block and directly add the input x to the output of the transformation. These shortcuts require no additional parameters and incur minimal computational cost, yet provide substantial improvements in gradient flow. By offering an unimpeded path for information and gradients, residual blocks facilitate the training of much deeper networks—up to 152 layers in the original paper—without suffering from degradation or vanishing gradient issues.

What is particularly noteworthy in my reading of this work is the elegance and minimalism of the intervention. The authors did not redesign the entire architecture from scratch; instead, they introduced a single, mathematically grounded modification that significantly enhanced optimization behaviour. Follow-up work, such as "Identity Mappings in Deep Residual Networks" (ECCV 2016), provided further refinements by suggesting pre-activation residual blocks and removing unnecessary nonlinearities within shortcuts—yielding even deeper and more stable architectures.

The impact of ResNet extends far beyond its initial success on ImageNet. It effectively changed how architectures are constructed, shifting the field from monolithic, stacked layers (as seen in VGG16) to modular, skip-connected blocks. Moreover, ResNet serves as a powerful backbone in transfer learning, particularly in tasks like cat/dog classification, where data may be limited or noisy. Pre-trained ResNets not only offer rich hierarchical features but also adapt quickly to new domains. In practical applications, I observed that ResNet's robustness to occlusion, cluttered backgrounds, and pose variation makes it particularly effective in real-world image classification settings.

In this sense, ResNet represents more than an architectural improvement—it is a paradigm shift that redefined the relationship between depth and trainability in deep learning.

1.3.4 Advancing Image Classification Tasks (e.g., Cat/Dog Classification)

The advancements pioneered by Kaiming He and the ResNet architecture have had a direct and significant positive impact on the performance of image classification tasks, ranging from large-scale benchmarks like ImageNet to more specific applications such as distinguishing between images of cats and dogs.

The primary benefit stems from the ability to train much deeper networks. Deeper ResNets can learn more complex and hierarchical feature representations compared to shallower networks. This improved representational capacity translates directly into higher accuracy on challenging benchmark datasets. The success on ImageNet, which contains numerous animal classes including various breeds of cats and dogs, demonstrated the enhanced ability of ResNets to capture subtle visual distinctions.

However, the most significant way ResNet impacts specific tasks like cat/dog classification, especially when task-specific labeled data is limited, is through *transfer learning*. The vast majority of high-performing cat/dog classifiers today leverage ResNet (or similar architectures inspired by it) pre-trained on large, diverse datasets like ImageNet. This process works primarily in two ways:

The pre-trained ResNet, excluding its final classification layer, is used as a fixed feature extractor. An input image (e.g., of a cat or dog) is passed through the ResNet backbone, and the resulting high-level feature map (a vector or tensor representation capturing learned visual patterns) is then fed into a new, typically much simpler, classifier (e.g., a linear support vector machine or a small neural network) trained specifically on the cat/dog dataset. The ResNet backbone effectively provides a powerful, general-purpose visual encoding.

The entire network architecture, initialized with pre-trained ResNet weights, is further trained on the target dataset (cats and dogs). Often, the weights of the earlier convolutional layers (the backbone) are either frozen or trained with a very small learning rate, while the later layers, including a newly added classification head specific to the target classes (e.g., two outputs for 'cat' and 'dog'), are trained more extensively. This adapts the learned features to the nuances of the specific task.

The power of transfer learning with ResNet backbones lies in leveraging the general visual knowledge acquired during pre-training. A ResNet trained on ImageNet has already learned to recognize fundamental visual concepts like edges, textures, shapes, parts of animals, fur patterns, etc. When applied to cat/dog classification, the model doesn't need to learn these basic concepts from scratch; it only needs to learn the specific combinations of features that differentiate cats from dogs. This dramatically reduces the amount of labeled data required for the target task and often leads to significantly better performance and faster convergence compared to training a deep network from scratch solely on cat/dog images.

Therefore, while Kaiming He may not have focused directly on cat/dog classification, his work on ResNet provided the transformative enabling technology – the powerful, pre-trainable engine – that makes achieving high accuracy on such specific tasks feasible and efficient through transfer learning. The success of ResNet as a backbone highlights the effectiveness of its architecture in capturing a rich, transferable hierarchy of visual features.

1.4 Baseline Model Selection and Project Direction

Based on the literature review, two classic convolutional neural network (CNN) architectures—VGG16 and ResNet—stand out as strong candidates for the baseline model in the cat and dog classification task. This section analyzes their respective merits and explains the rationale for choosing the most appropriate approach to establish a robust and efficient starting point for this project.

1.4.1 VGG16: Simplicity and Transferable Features

VGG16 remains a widely adopted architecture due to its straightforward, modular design and competitive performance on image classification benchmarks. The uniform and deep architecture enables the extraction of hierarchical features, progressing from edges to textures and object-level structures.

A key advantage of VGG16 is its interpretability and transferability. Pre-trained VGG16 models are readily available and often used as feature extractors for downstream tasks. In transfer learning settings, particularly with well-centered and clean images, VGG16 performs reliably and requires minimal architectural tuning. Its consistent layer structure makes it especially attractive for rapid prototyping and fine-tuning on relatively small datasets.

However, VGG16 comes with certain limitations. It has a large number of parameters (about 138 million), making it computationally intensive and prone to overfitting if regularization is not carefully applied. Moreover, the absence of internal mechanisms to address optimization challenges can limit its scalability in deeper or more complex classification problems.

1.4.2 ResNet: Scalable Depth and Optimization Stability

ResNet introduced a fundamental shift in CNN architecture by reformulating layer learning as residual learning. ResNet architectures are built using residual blocks, with variants tailored to different depths. For example, ResNet-18 and ResNet-34 use basic blocks with two 3×3 convolutions, while deeper variants like ResNet-50 and ResNet-101 use bottleneck blocks with 1×1 and 3×3 convolutions to reduce parameter count and improve efficiency. The residual structure allows for identity mappings when needed, offering both optimization benefits and training stability.

In the context of cat/dog classification, especially under real-world conditions (e.g., occlusion, lighting variation, pose diversity), ResNet's deeper and more expressive architecture enables better generalization. Moreover, ResNet models pre-trained on ImageNet are widely available and have demonstrated strong performance in transfer learning settings. Fine-tuning a ResNet backbone allows leveraging its learned hierarchical features while adapting the classifier head for specific categories. And from Table below, ResNet50 is emphasized in this project.

Table 1: Common ResNet Variants and Key Features

Variant	Layers	Relative Complexity	Features
ResNet-18	18	Low	Fast
ResNet-34	34	Medium	Better performance
ResNet-50	50	Medium-High	Common baseline
ResNet-101	101	High	Stronger performance
ResNet-152	152	Very High	Powerful but intensive

1.4.3 Implementation Strategy

This project will explore both VGG16 and ResNet-based solutions using transfer learning techniques. Pre-trained weights from ImageNet will be used to initialize the models, followed by fine-tuning on a selected dataset the Kaggle Dogs vs. Cats dataset.

1.4.4 Future Work and Extensions

While VGG16 and ResNet provide solid starting points, future work will consider more advanced architectures such as EfficientNet and Vision Transformers (ViT), as well as alternative training strategies including self-supervised learning (e.g., DINOv2) and multimodal fusion approaches. These methods offer potential for further improving classification accuracy, especially in more challenging scenarios involving cluttered backgrounds, occlusion, or fine-grained breed distinctions.

In summary, this project adopts VGG16 and ResNet as baseline architectures based on their proven effectiveness, availability of pre-trained models, and adaptability to transfer learning. The implementation and evaluation of both models will form the foundation for further experimentation and enhancement.

2 Dataset Preparation

2.1 Data Split

In this project, we used the provided dataset, which consists of training, validation, and test sets. The given training set contained a total of 20,000 images, divided into 10,000 cat images and 10,000 dog images. The validation set had 5,000 images, and cats and dogs each represented 2,500 images. The test set included 500 images that were unlabeled and used to test the final classification performance.

Considering the limited computing resources, we randomly selected a subset from the original dataset for training and validation. Specifically, from the original 20,000 training images, we randomly chose 2,000 images (around 1,000 cats and 1,000 dogs). For validation, we selected 500 images from the original 5,000 validation images, keeping the classes balanced with approximately 250 images each of cats and dogs. All 500 images from the test set were used directly for the final evaluation.

2.2 Preprocessing and Augmentation

For data preprocessing, we applied normalization and cropping steps across all datasets. Specifically, all images were resized to a size of 224×224 pixels. Furthermore, we normalize the RGB channels of each image using the commonly used ImageNet mean and standard deviation. Normalization helped reduce variations caused by lighting or other noise, making it easier for the model to learn the features.

We also applied several data augmentation techniques to the training set to improve generalization. These included random resized cropping, random horizontal flipping (with a 50% chance), and random rotations within ±20 degrees. These augmentations introduced variability in scale and orientation, improving the model's ability to recognize images under diverse conditions.

All preprocessing steps and augmentations were implemented using PyTorch's transforms module. The corresponding code is shown below:

Listing 1: Data preprocessing and augmentation code

```
data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(20),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
    ]),
    'test': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
    ])
}
```

3 Model Design and Implementation

VGG16(3.1) and Resnet(3.2) are used as models in this project.

3.1 VGG16

The main feature of VGG16 is its deep and uniform network structure, consisting of a large number of convolutional and pooling layers that enable the model to learn high-level abstract features effectively. It consistently uses small 3×3 convolutional kernels, which allow for deeper architectures without significantly increasing computational cost. This design improves both the accuracy and efficiency of feature extraction by enhancing the network's depth and receptive field while maintaining manageable complexity.

3.1.1 Model Architecture

Several key types of layer are shown below, each serving a distinct function in the image classification process.

- **Convolutional Layer:** Performs convolution operations on input data to extract local features. Each layer uses a 3×3 kernel with a ReLU activation function, defined as $F(x) = \max(0, x)$, to introduce non-linearity.
- **Pooling Layer:** Reduces the spatial resolution of the feature maps while retaining important features, helping to reduce the number of parameters and computation. The design includes 5 pooling layers, each using a 2×2 kernel with stride 2. Max pooling and average pooling are commonly used.
- **Normalization Layer:** Standardizes the output of convolutional layers to improve the model's robustness and generalization ability.
- **Fully Connected Layer:** Flattens the feature maps into a vector and maps it to the output space for classification or prediction through fully connected neurons.
- **Output Layer:** Produces the final classification result as a probability distribution, using a softmax function and optimizing with a cross-entropy loss function.

In our implementation, we retain the feature extraction layers of the original VGG16 model (consisting of 13 convolutional layers and 5 max-pooling layers) and replace the final fully connected classification layer to adapt the model for binary classification (cat vs. dog).

The model employs the standard cross-entropy loss function for binary classification with two output logits. Given a sample $(\mathbf{x}, y)$ where $y \in \{0, 1\}$ is the true label and $\mathbf{p} = \text{softmax}(f(\mathbf{x}))$ is the predicted probability vector over the two classes, the cross-entropy loss is defined as:

$$\text{CrossEntropyLoss}(x, y) = -\left[y \log(p_1) + (1 - y) \log(p_0)\right] \tag{1}$$

Where:

- $p_0 = \frac{e^{z_0}}{e^{z_0}+e^{z_1}}$ is the probability of the sample belonging to class 0,
- $p_1 = \frac{e^{z_1}}{e^{z_0}+e^{z_1}}$ is the probability of the sample belonging to class 1,
- z_0 and z_1 are the logits (raw model outputs) for class 0 and class 1, respectively.

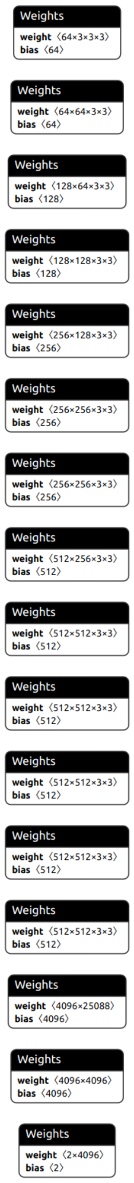

Figure 1: Modified VGG16 Architecture with Final Classification Head Replaced (Output = 2)

The input image is resized to $224 \times 224 \times 3$, and passed through a sequence of convolutional layers followed by three fully connected layers. The last layer is modified to output a 2-dimensional vector representing the class logits for cat and dog. Softmax is implicitly applied during training via the cross-entropy loss.

3.1.2 Training Strategy

The model is fine-tuned using a transfer learning approach based on the VGG16 architecture. The convolutional layers are frozen to preserve pretrained features, and only the final fully connected layers are updated during training.

- **Optimizer:** Adam optimizer
- **Learning Rate:** 0.00001
- **Batch Size:** 16
- **Epochs:** 60 (with early stopping)
- **Loss Function:** Cross-Entropy Loss

3.1.3 Reproducibility

To ensure reproducibility of the training and evaluation results, the following strategies were adopted:

- A fixed random seed (42) was set using NumPy to ensure deterministic dataset partitioning:

  Listing 2: Random seed code

  ```

              np.random.seed(42)
  ```

- Dataset splits and DataLoader were constructed in a non-randomized, reproducible way.
- The model used pretrained VGG16 weights, and only the classifier layers were fine-tuned.
- Training results can be replicated by saving and loading the model checkpoint `vgg_catdog_final.pth`.

3.2 ResNet

In this project, we chose ResNet50 as our model. It was originally trained on the ImageNet dataset, and we made some modifications to adapt our task.

3.2.1 Model Architecture

The input images were first resized to a dimension of $224 \times 224 \times 3$ and then fed into ResNet50 to extract features. We froze all convolutional and pooling layers of the ResNet50 and got the resulting feature vectors. Then we removed the original fully-connected layer and replaced it with our binary classification layer, as shown in Figure 3.

For the loss function, we chose the binary cross-entropy loss (BCE), defined by:

$$\text{Loss} = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right] \tag{2}$$

where y_i is the actual label (either 0 or 1) and $\hat{y}_i$ is the predicted probability output.

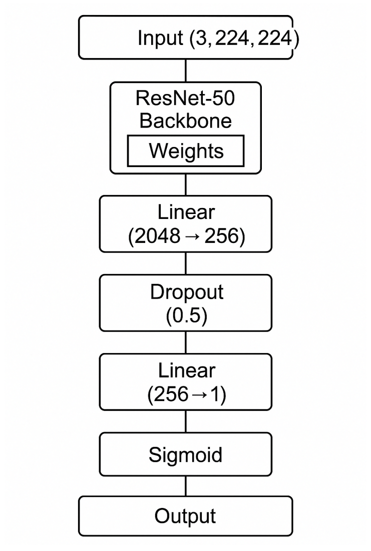

Figure 2: ResNet50 model structure. The feature vector first passes through a fully-connected layer followed by a ReLU activation function, and then a dropout is applied to prevent overfitting. Then we added a fully-connected layer using a Sigmoid function, indicating the probability of being either a cat or dog.

3.2.2 Training Strategy

For the training process, we used the Adam optimizer. The initial learning rate is set to 0.0001, and trained the model for a maximum of 30 epochs. To avoid overfitting, we applied an early-stopping strategy. Specifically, if the validation loss didn't improve for 3 epochs, training would stop, and the model with the best validation performance would be saved.

3.2.3 Reproducibility

To support reproducibility, we also fixed the numpy random seed of our data selection for training and validation, the same as Listing 2.

4 Hyperparameter Tuning

In this part, we explain how we selected the key parameters for training our model. We mainly focused on tuning two parameters: learning rate and batch size. Below, we show the validation accuracy achieved for different settings for two models seperately.

4.1 VGG16:

4.1.1 Learning Rate

We experimented with five different learning rates and observed the validation accuracy after 30 epochs (with early stopping).

Table 2: Validation accuracy for different learning rates for VGG16

Learning Rate	Validation Accuracy (%)
0.01	96.8
0.001	98.0
0.0001	98.4
0.00001	98.6
0.000001	98.4

We noticed that a high learning rate (0.01) resulted in unstable training and lower validation accuracy. Lowering the learning rate improved stability significantly. Among all tested values, 0.00001 provided the highest and most stable accuracy, so we selected this as our final learning rate.

4.1.2 Batch Size

We also compared four different batch sizes:

Table 3: Validation accuracy for different batch sizes for VGG16

Batch Size	Validation Accuracy (%)
8	98.4
16	98.6
32	98.0
64	97.0

The batch size of 16 gave the highest accuracy with stable performance. When the batch size was smaller, accuracy fluctuated more. Larger batch size slowed down training and slightly decreased accuracy. Thus, batch size with 16 was selected for optimal training efficiency and performance.

4.1.3 Early Stopping

To prevent overfitting and reduce unnecessary training time, we implemented an early stopping strategy during training. The validation loss was monitored at the end of each epoch.

If the validation loss did not improve by at least $\delta = 0.001$ for a consecutive 5 epochs (i.e., *patience*), the training process was terminated early.

Formally, let $\mathcal{L}_{val}^{(e)}$ denote the validation loss at epoch e. Early stopping is triggered if:

$$\mathcal{L}_{val}^{(e)} \geq \min_{i<e} \left(\mathcal{L}_{val}^{(i)} \right) - \delta \quad \text{for } p \text{ consecutive epochs} \tag{3}$$

In our implementation, the best-performing model (with the lowest validation loss) was saved to disk, allowing us to resume evaluation using the most generalizable parameters.

4.1.4 Plot Loss Curve

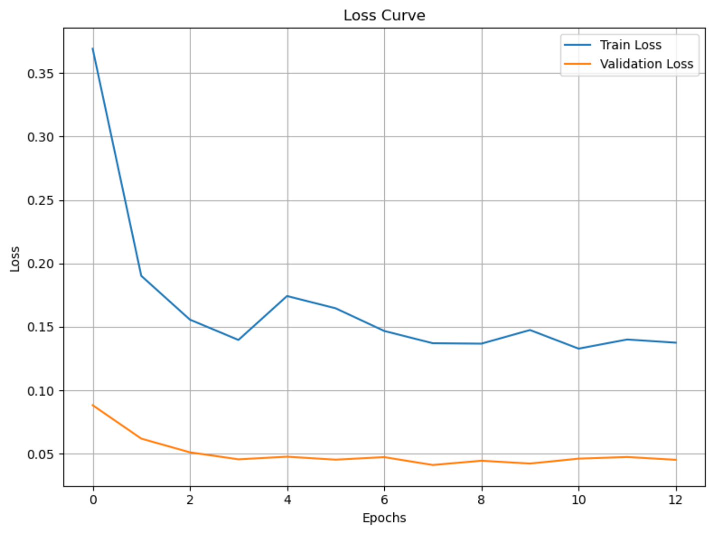

Figure 3: Final parameters for VGG16: lr=0.00001, batchsize=16

4.2 ResNet:

4.2.1 Learning Rate

We experimented with four different learning rates and observed the validation accuracy after 30 epochs.

Table 4: Validation accuracy for different learning rates for ResNet50

Learning Rate	Validation Accuracy (%)
0.01	96.6
0.001	98.0
0.0001	98.4
0.00005	97.6

We noticed that a high learning rate (0.01) resulted in unstable training and lower validation accuracy. Lowering the learning rate improved stability significantly. Among all tested values, 0.0001 provided the highest and most stable accuracy, so we selected this as our final learning rate.

4.2.2 Batch Size

We also compared three different batch sizes:

Table 5: Validation accuracy for different batch sizes for ResNet50

Batch Size	Validation Accuracy (%)
16	98.2
32	98.4
64	96.0

The batch size of 32 gave the highest accuracy with stable performance. When the batch size was smaller, accuracy fluctuated more. Larger batch size slowed down training and slightly decreased accuracy. Thus, batch size with 32 was selected for optimal training efficiency and performance.

4.2.3 Early Stopping

We applied early stopping with a patience of 3 epochs to avoid overfitting. If validation loss didn't improve for three consecutive epochs, training stopped early, and the best-performing model was saved.

4.2.4 Plot Loss Curve

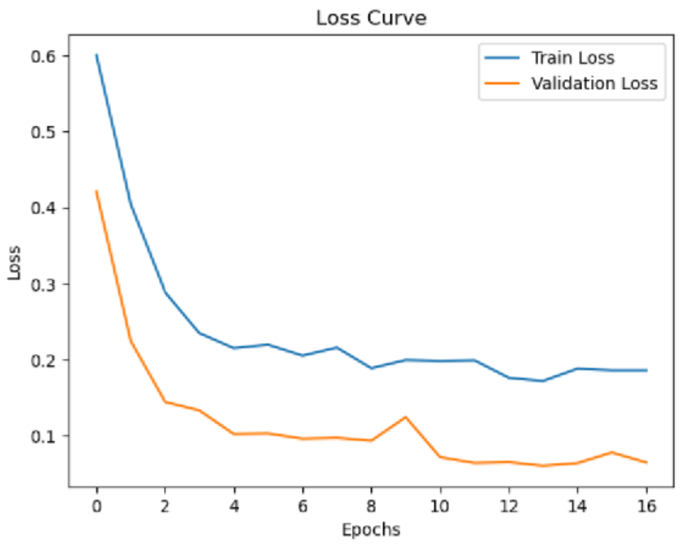

Figure 4: Final parameters for ResNet50: lr=0.0001, batchsize=32

5 Model Evaluation

In this part, we show how both VGG16 and ResNet50 performed on our validation set. We also provide results from the test dataset predictions.

5.1 Validation Accuracy

5.1.1 VGG16 Performance

The confusion matrix for the VGG16 model on the validation set is shown in Table 9:

Table 6: Confusion Matrix for VGG16 (Validation Set)

	Predicted Cat	Predicted Dog
Actual Cat	241	5
Actual Dog	1	253

Using this confusion matrix, we calculated several metrics to evaluate the model:

- **Accuracy:**

$$\text{Accuracy} = \frac{241+253}{241+253+5+1} \approx 98.2\%$$

- **Precision:**

$$\text{Precision} = \frac{253}{253+5} \approx 97.97\%$$

- **Recall:**

$$\text{Recall} = \frac{253}{253+1} \approx 99.59\%$$

- **F1-Score:**

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \approx 98.77\%$$

5.1.2 ResNet50 Performance

The confusion matrix for ResNet50 on the validation set is shown in Table 7:

Table 7: Confusion Matrix for ResNet50 (Validation Set)

	Predicted Cat	Predicted Dog
Actual Cat	253	5
Actual Dog	3	239

Based on the confusion matrix above, we computed these evaluation metrics:

- **Accuracy:**

$$\text{Accuracy} = \frac{253+239}{253+239+5+3} \approx 98.4\%$$

- **Precision:**

$$\text{Precision} = \frac{239}{239+5} \approx 97.95\%$$

- **Recall:**
$$\text{Recall} = \frac{239}{239+3} \approx 98.76\%$$
- **F1-Score:**
$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \approx 98.35\%$$

Comparing the two models, ResNet50 and VGG16 achieved similar performance in terms of accuracy, precision, recall, and F1-score. However, VGG16 showed slightly better results overall, indicating that it may have a modest advantage in generalization for this classification task.

5.2 Test Prediction

While we selected the VGG16 model to generate `submission.csv` for final test set evaluation, this decision was made based on its stability and ease of fine-tuning. However, since both VGG16 and ResNet-50 demonstrated comparable performance on the validation set, we proceed to compare both models in greater detail in Chapter 6: *Model Analysis*, in order to better understand their respective strengths and weaknesses.

6 Model Analysis

For test set, we first get groundtruth by observing with our own eyes, then compare the groundtruth with prediction results from VGG16 and ResNet50.

6.1 Correctly Classified Samples

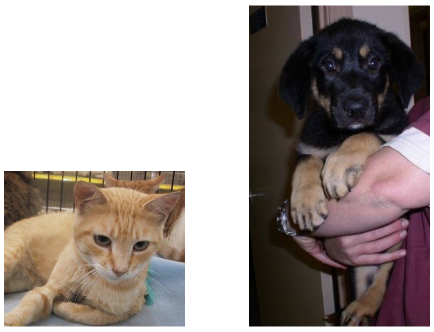

Figure 5: Two right predicted test images

Feature Type	Cat Image	Dog Image
Ear Shape	Sharp, erect	Rounded, floppy
Fur Texture	Short, fine, smooth	Coarse, fluffy, varied pattern
Eye Shape	Almond-shaped, well-defined pupil	Round, softer iris edges
Facial Contour	Small muzzle, smooth jawline	Broader snout, pronounced jawline
Limb Posture	Tucked close to body	Extended with visible paw pads

Table 8: Illustrative features extracted by VGG16 and ResNet50 for correctly classifying a cat versus a dog image

These two images are likely to be correctly classified by all models because they exhibit highly distinguishable and prototypical features of their respective categories. The cat image displays sharp, erect ears, a small and smooth facial contour, and a tucked limb posture—all of which align well with the model's learned features for a typical cat. In contrast, the dog image clearly shows rounded, floppy ears, a broader snout with a pronounced jawline, and extended limbs with visible paw pads. These distinct anatomical characteristics, combined with well-lit and unobstructed views of the animals' faces, enable the model to extract discriminative features and make confident predictions.

6.2 Misclassified Samples

6.2.1 VGG16

The confusion matrix for the VGG16 model on the test set is shown in Table 9:

Table 9: Confusion Matrix for VGG16 (test Set)

	Predicted Cat	Predicted Dog
Actual Cat	257	3
Actual Dog	2	238

Using this confusion matrix, we calculated several metrics to evaluate the model:

- **Accuracy:**
$$\text{Accuracy} = \frac{257+238}{257+238+3+2} \approx 99.0\%$$
- **Precision:**
$$\text{Precision} = \frac{238}{238+3} \approx 98.85\%$$
- **Recall:**
$$\text{Recall} = \frac{238}{238+2} \approx 99.23\%$$
- **F1-Score:**
$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \approx 99.04\%$$

All Misclassified Samples for VGG16:

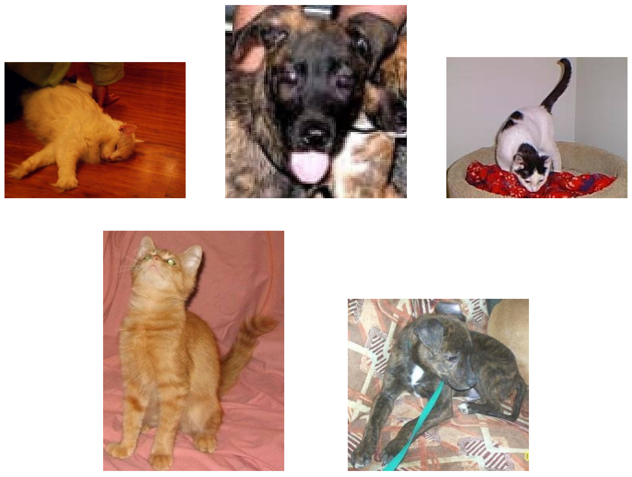

Figure 6: Five wrongly predicted test images by VGG16

All misclassified samples from the test set are analyzed, with image IDs `133, 302, 403, 428, 465`. The following table summarizes each sample along with a visual interpretation of why it may have been misclassified:

Table 10: Analysis of Misclassified Samples by VGG16

ID	True Label	Possible Reason for Misclassification
133	Cat	Low lighting and non-standard pose reduce visibility of key features
302	Dog	Blurry details and facial ambiguity may resemble a cat
403	Cat	Posture is confusing, and tail shape may look like an ear
428	Cat	Uniform color and upward-facing pose obscure facial structure
465	Dog	Complex background and occlusion cause structural confusion

These misclassified samples share several common characteristics:

1. Non-standard poses such as lying down, curling, or looking away from the camera.

2. Complex or cluttered backgrounds that introduce noise and distract feature extraction.
3. Low visibility of facial or discriminative features due to lighting, occlusion, or orientation.

Such conditions reduce the effectiveness of spatial feature extraction in convolutional layers, especially for models trained primarily on clean, well-centered, and clearly illuminated examples.

6.2.2 ResNet50

The confusion matrix for the ResNet50 model on the test set is shown in Table 11:

Table 11: Confusion Matrix for ResNet50 (test Set)

	Predicted Cat	Predicted Dog
Actual Cat	258	5
Actual Dog	1	236

Using this confusion matrix, we calculated several metrics to evaluate the model:

- **Accuracy:**
$$\text{Accuracy} = \frac{258+236}{258+236+5+1} \approx 98.8\%$$
- **Precision:**
$$\text{Precision} = \frac{236}{236+5} \approx 98.10\%$$
- **Recall:**
$$\text{Recall} = \frac{236}{236+1} \approx 99.61\%$$
- **F1-Score:**
$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \approx 98.85\%$$

All Misclassified Samples for ResNet50:

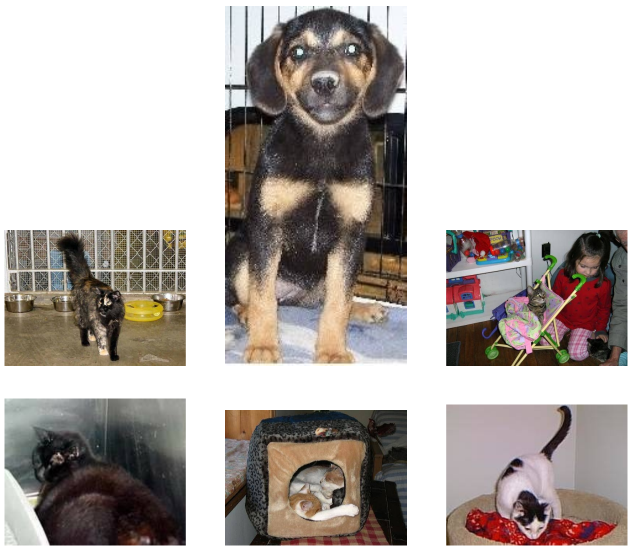

Figure 7: Six wrongly predicted test images by ResNet50

All misclassified samples from the test set are analyzed, with image IDs `46, 261, 293, 342, 374, 403`. The following table summarizes each sample along with a visual interpretation of why it may have been misclassified:

Table 12: Analysis of Misclassified Samples by ResNet-50

ID	True Label	Possible Reason for Misclassification
46	Cat	Complex background may confuse model's attention
261	Dog	Dog's facial vertical compression distorts proportions
293	Cat	Strong human-object interaction and cluttered background
342	Cat	Very dark lighting and rear-facing pose obscure all facial features
374	Cat	Multiple cats with overlapping bodies; ambiguous texture cues
403	Cat	Back-arched pose and emphasized tail

These misclassified cases reveal that ResNet-50, still struggles with certain image conditions:

- **Cluttered or distracting backgrounds**: such as toys, cages, or furniture, which draw attention away from the object of interest.
- **Uncommon viewpoints or body postures**: such as curled cats, rear views, or compressed angles.
- **Occlusion or multiple subjects**: especially when animals overlap or interact with humans or objects.
- **Low illumination and contrast**: which makes edge-based features unreliable for classification.

While ResNet-50 benefits from residual connections and deeper architecture, these results suggest that it still depends heavily on clear spatial and textural cues for correct classification.

6.2.3 Images with wrong prediction for both models

From results above, ID: 403 is predicted wrongly by both two models.

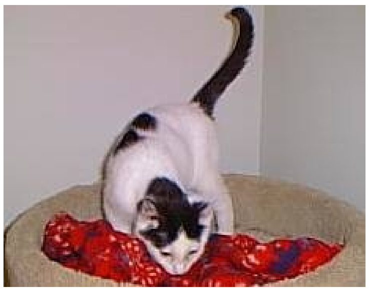

Figure 8: Wrongly predicted test image by both VGG16 and ResNet-50

The image presents several challenges that likely led to misclassification by both VGG16 and ResNet-50:

- **Unusual posture:** The cat is curled forward with its head down and tail up, resulting in an ambiguous body outline.
- **Obscured facial features:** Key discriminative regions such as the eyes, ears, and nose are not clearly visible.
- **Distracting background:** The bright wall and patterned red cloth introduce strong local contrasts that can mislead the feature extractor.

6.3 Pros and Cons

Table 13: Comparison of VGG16 and ResNet-50 in Cat vs. Dog Classification

Condition	**VGG16**	**ResNet-50**	**Better**
Standard clear images	Accurate when features are sharp and centered	Also accurate; deeper refinement	=
Unusual poses	Sensitive to changes in shape	More tolerant to variation	ResNet-50
Occluded faces	Heavily affected	Less sensitive due to deeper features	ResNet-50
Multiple animals	Better handles overlapping shapes due to clear local textures	Still confused by crowding	VGG16
Cluttered backgrounds	Local edge filters help distinguish object	May overfit to background texture	VGG16
Low lighting / blur	Poor response to noise or darkness	More robust to visual noise	ResNet-50
Similar color between animal and background	Struggles with visual blending	Extracts deeper features to separate foreground	ResNet-50
Texture-based detail	Strong at local textures (fur, edges)	Focuses more on shape than texture	VGG16
Training on small data	Stable when freezing conv layers	May need full fine-tuning	VGG16

7 Model Variations

In this section, we discuss how the choice of models and data preprocessing methods affected the accuracy on the validation set.

7.1 Effect of Model Choice

We tried two different CNN models—VGG16 and ResNet50—to see how they performed on the validation data. Even though both are deep convolutional neural networks, there were some clear differences in how well they performed and how well they generalized to new data.

With exactly the same preprocessing, VGG16 achieved a F1-Score of around 98.77%, while ResNet50 obtained a slightly lower F1-Score of approximately 98.35%. Although both models performed similarly, the slightly better performance of VGG16 might come from its simpler and uniform architecture, which allows it to efficiently learn from relatively small datasets without introducing excessive complexity. ResNet50, despite its powerful residual connections that usually help prevent overfitting and facilitate training deeper networks, did not outperform VGG16 in our case.

Given these results, we decided that VGG16 was more suitable as the baseline model for our project.

7.2 Effect of Preprocessing

We also looked at how data augmentation techniques affected our model's accuracy on the validation set.

Without any data augmentation (just basic cropping and normalization), ResNet50's accuracy was about 92.8%. But when we added random resized cropping, random horizontal flipping (with 50% chance), and random rotation (±20 degrees), the accuracy improved to around 98.4%.

This clearly shows that data augmentation helps the model learn more effectively by seeing images in different forms, making the model more flexible and better at recognizing patterns in new data.

Overall, from our experiments, we saw clearly that the model structure and preprocessing techniques significantly impacted validation accuracy. Using ResNet50 combined with data augmentation provided the best results, reaching an accuracy of about 98.4% on the validation set. So, we decided that the combination of ResNet50 with data augmentation was the best approach for our project.

8 Multi-Class Classification (CIFAR-10)

8.1 Dataset Description

The CIFAR-10 dataset contains a total of 60,000 color images, each with a size of 32×32 pixels. There are 10 different classes, and each class has 6,000 images. These classes are airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.

The dataset is split into five training batches and one test batch. Each batch has 10,000 images. Specifically, the test batch contains 1,000 randomly selected images from each class. The remaining 50,000 images are used for training, spread across five batches. Although each class has 5,000 images in total for training, their distribution across the batches might not be perfectly balanced.

Figure 9 shows 10 random sample images from each class. We can clearly see that some classes might be visually similar (e.g., cat and dog, automobile and truck), making the classification task challenging.

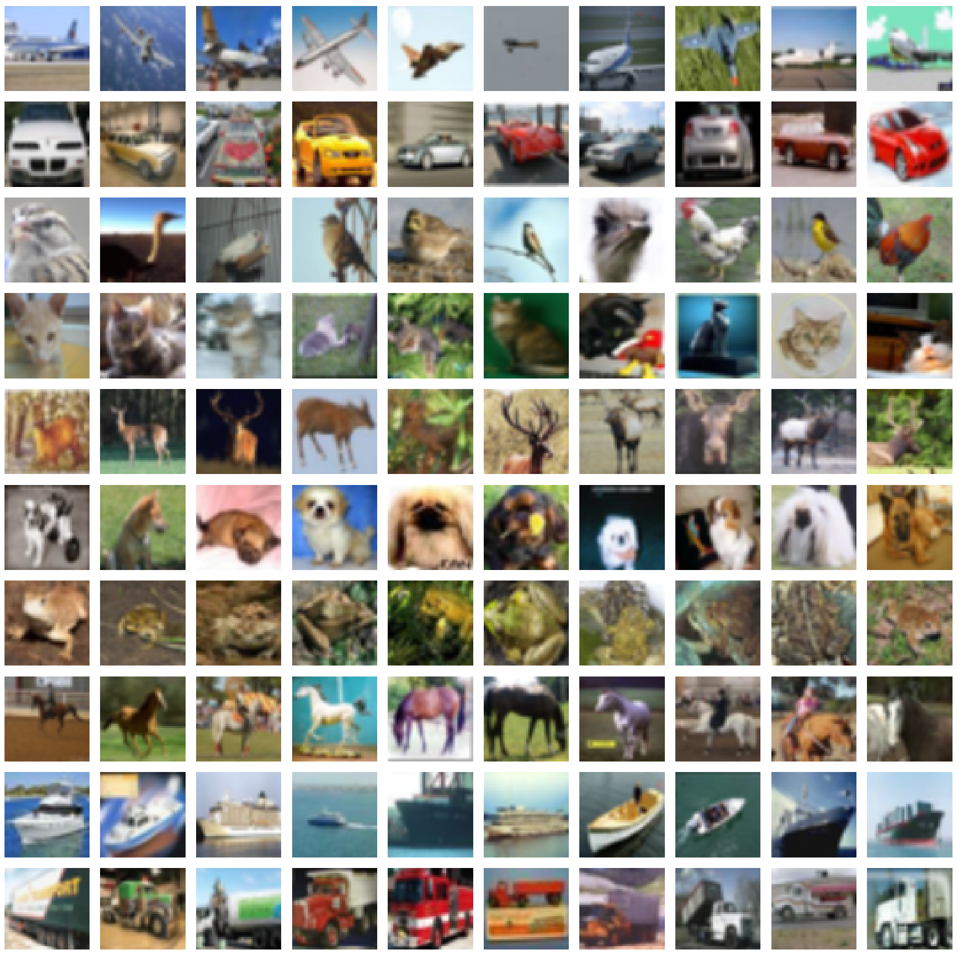

Figure 9: Random sample images from each class in the CIFAR-10 dataset[13]

8.2 Model Extension

Originally, our models (VGG16 and ResNet50) were designed to handle the binary classification problem (cats vs. dogs). To extend these models to the CIFAR-10 multi-class task, we made several adjustments.

First, we modified the final output layer from a single node to 10 nodes, corresponding to the 10 classes in CIFAR-10. Additionally, we replaced the original sigmoid activation function with a softmax function, since softmax outputs a probability distribution suitable for multi-class classification problems. The loss function was also changed from binary cross-entropy (BCE) to categorical cross-entropy loss.

Second, we kept the data augmentation strategies from before (random cropping, horizontal flipping, and rotation) to help the model generalize better. However, we adjusted the input size of the images to fit the model's input requirements (224×224 pixels).

8.3 Performance Report

We conducted initial experiments with both VGG16 and ResNet50 models on the CIFAR-10 test set. The classification accuracies are summarized below:

VGG16

Table 14: VGG16 on Cifar-10 for Last Three Epochs

Epoch	Train Loss	Val Loss	Val Accuracy
...	...	...	...
28	0.9371	0.4215	0.8615
29	0.9328	0.4202	0.8582
30	0.9316	0.4157	0.8603

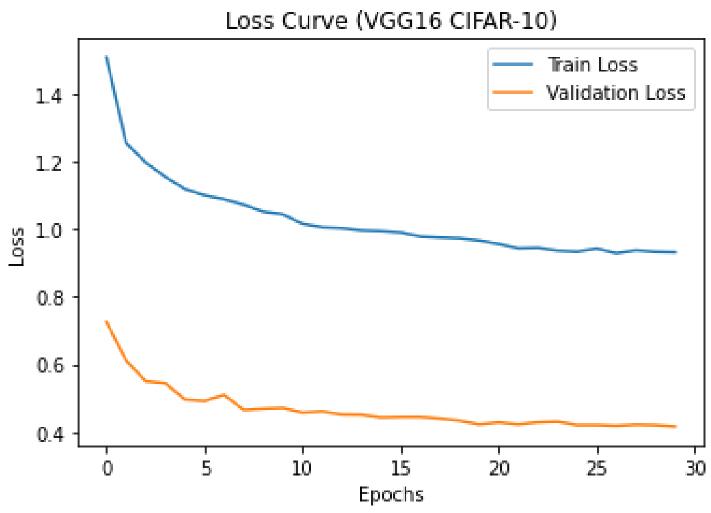

Figure 10: Loss Curve for VGG16

Table 15: Classification Report (CIFAR-10)

Class	**Precision**	**Recall**	**F1-Score**	**Support**
0	0.8862	0.8640	0.8749	1000
1	0.9076	0.9430	0.9250	1000
2	0.8494	0.8010	0.8245	1000
3	0.7603	0.7170	0.7380	1000
4	0.8358	0.8040	0.8196	1000
5	0.7942	0.8450	0.8188	1000
6	0.8373	0.9420	0.8866	1000
7	0.8959	0.8520	0.8734	1000
8	0.9166	0.9120	0.9143	1000
9	0.9202	0.9230	0.9216	1000
Macro Avg	0.8603	0.8603	0.8597	10000
Weighted Avg	0.8603	0.8603	0.8597	10000
Overall Accuracy	86.03%			

Table 16: Confusion Matrix (CIFAR-10)

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	864	13	22	14	11	2	5	7	45	17
1	9	943	0	2	0	0	0	0	6	40
2	19	1	801	38	50	23	47	16	4	1
3	9	6	34	717	21	130	56	14	9	4
4	9	2	42	28	804	23	48	39	4	1
5	3	2	11	88	18	845	12	18	1	2
6	2	1	13	22	12	6	942	2	0	0
7	6	5	15	28	43	35	12	852	2	2
8	37	24	3	5	2	0	3	1	912	13
9	17	42	2	1	1	0	0	2	12	923

ResNet50:

Table 17: ResNet50 on Cifar-10 for Last Three Epochs

Epoch	Train Loss	Val Loss	Val Accuracy
...	...	...	...
14	1.2499	0.6345	0.7804
15	1.2407	0.6343	0.7781
16	1.2520	0.6374	0.7775

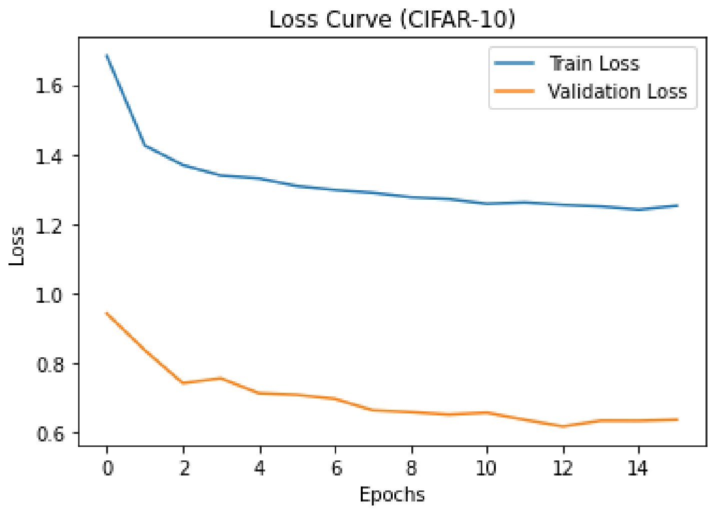

Figure 11: Loss Curve for ResNet50

In our experiments on CIFAR-10, VGG16 achieved a higher validation accuracy (around 86.03%) compared to ResNet50 (around 78%). VGG16 showed stable training and validation performance, especially for classes like "automobile" and "truck", but struggled slightly on classes like "cat" and "bird". ResNet50 performed less effectively, possibly due to the complexity of adapting residual connections to CIFAR-10's smaller images. Overall, VGG16 appears more suitable as our baseline for further work.

9 Handling Data Imbalance

9.1 Problem Setup

In practice, when using the CIFAR-10 dataset, we might encounter a data imbalance issue, meaning some classes could have significantly more images than others. If we don't handle this imbalance, the trained model would likely perform well on classes with more images, but poorly on classes with fewer images, reducing overall generalization performance.

9.2 Solution 1: Data Resampling

Data resampling is a common method to handle imbalance issues. There are two main approaches: oversampling and undersampling.

Oversampling means increasing the amount of data for minority classes by either duplicating existing images or generating new images through data augmentation. Common augmentation techniques include random cropping, random rotation, color jittering, and horizontal flipping. Undersampling, on the other hand, reduces the number of samples in majority classes to match minority classes. However, undersampling could lead to losing useful information, and it's more suitable when the dataset is large and redundant.

Since the CIFAR-10 dataset isn't very large, we chose oversampling by applying data augmentation techniques to increase the amount of data in classes with fewer images. This approach helps maintain valuable information and improves model generalization.

9.3 Solution 2: Loss Modification or Regularisation

Besides data resampling, we can also directly modify the loss function to make the model pay more attention to minority classes. Two common methods are class weighting and focal loss.

Class weighting assigns different weights to different classes in the loss function calculation. Classes with fewer images get higher weights, and classes with more images receive lower weights. The loss function can be represented as:

$$\text{Loss} = -\frac{1}{N}\sum_{i=1}^{N} w_{y_i} \log \frac{e^{z_{y_i}}}{\sum_{j=1}^{C} e^{z_j}}$$

where w_{y_i} is the weight for the true class y_i of the i-th sample, z_j is the output logit for class j, and C is the total number of classes.

Focal loss is specifically designed for imbalanced datasets, automatically reducing the loss contribution of easily classified samples and putting more focus on harder-to-classify samples. The formula for focal loss is:

$$\text{Focal Loss} = -\frac{1}{N}\sum_{i=1}^{N} \alpha(1-p_{y_i})^{\gamma}\log(p_{y_i})$$

where p_{y_i} is the predicted probability of the true class y_i for the i-th sample, α is a class weighting factor, and γ adjusts the rate at which easily classified samples are down-weighted.

In our project, we decided to use the simpler approach of class weighting because it's easier to implement and effective enough for imbalanced scenarios.

9.4 Justification

We selected these two methods based on two reasons. Oversampling through data augmentation can effectively increase the size of minority classes without losing valuable information. This method has been widely used and proven successful in various image classification tasks.

Class weighting helps the model significantly improve predictions for minority classes by adjusting their importance during training. This approach is easy to implement, and particularly effective when dealing with moderate class imbalance scenarios.

10 Respective Contribution

Name (Student ID)	**Tasks and Responsibilities**	**Share**
Zhang Zezheng (G2402715A)	1(c). Reviewed VGG16-related literature for classification tasks. 1(d). Proposed VGG16 as a baseline model. 3(a). Built and trained the VGG16 classifier using transfer learning. 3(b). Ensured reproducibility through checkpointing and logs. 4(a). Tuned VGG16 hyperparameters (LR, batch size, etc.). 5(a). Evaluated model on validation and test sets. 5(b). Generated and formatted test prediction CSVs. 6(a). Analyzed VGG16 misclassifications. 6(b). Summarized VGG16 strengths and weaknesses. 7(a). Discussed VGG16 sensitivity to image variation. 8(a). Adapted VGG16 for CIFAR-10 multi-class task. 8(b). Compared VGG16 to other baselines on CIFAR-10.	35%
Zhang Na (G2402918G)	2(a). Defined dataset splits for training and validation. 2(b). Applied pre-processing and augmentation techniques. 3(a). Implemented ResNet-50 classifier and training pipeline. 3(b). Maintained model scripts and training logs. 4(a). Tuned ResNet learning rate and scheduling. 5(a). Recorded validation and test accuracy of ResNet. 5(b). Submitted final ResNet predictions. 6(a). Diagnosed errors in ResNet predictions. 6(b). Evaluated ResNet robustness to image challenges. 7(a). Analyzed data/model influence on ResNet. 8(a). Modified ResNet for CIFAR-10 adaptation. 8(b). Compared ResNet to VGG16 on CIFAR-10. 9(a). Applied class weighting and resampling on CIFAR-10.	35%
Wang Meng (G2403580A)	1(a). Studied CNN fundamentals and tutorials. 1(b). Participated in literature search and review. 2(b). Helped prepare and check dataset integrity. 3(a). Verified output and adjusted batch/GPU configs. 5(b). Produced CSV predictions and validated format. 6(a). Helped visualize correct/incorrect predictions. 6(b). Co-wrote analysis of model behaviour. 7(a). Reflected on model choices and performance. 8(a). Verified CIFAR-10 input pipeline. 9(a). Tested SMOTE and class-weighted loss.	30%

Code Repository

- **The source code for VGG16 experiments in this report is available at:**
  https://github.com/DarlinZZZ/EE6483
- **The source code for ResNet50 experiments in this report is available at:**
  https://github.com/Qimhubushme/EE6483-project

References

1 Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. *Advances in neural information processing systems (NeurIPS)*, 25.

2 Simonyan, K., & Zisserman, A. (2015). Very Deep Convolutional Networks for Large-Scale Image Recognition. *International Conference on Learning Representations (ICLR)*.

3 Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D.,... & Rabinovich, A. (2015). Going deeper with convolutions. *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 1-9.

4 Parkhi, O. M., Vedaldi, A., Zisserman, A., & Jawahar, C. V. (2012). Cats and dogs. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 3498-3505.

5 Xie, S., Yang, T., Wang, X., & Lin, Y. (2015). Hyper-class augmented and regularized deep learning for fine-grained image classification. *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2645-2654.

6 Tan, M., & Le, Q. (2019). EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. *International conference on machine learning (ICML)*, 6105-6114.

7 Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T.,... & Houlsby, N. (2021). An image is worth 16x16 words: Transformers for image recognition at scale. *International Conference on Learning Representations (ICLR)*.

8 Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S.,... & Sutskever, I. (2021). Learning transferable visual models from natural language supervision. *International conference on machine learning (ICML)*, 8748-8763.

9 Oquab, M., Darcet, T., Moutakanni, T., Vo, H. V., Szafraniec, M., Khalidov, V.,... & Bojanowski, P. (2023). DINOv2: Learning Robust Visual Features without Supervision. *Transactions on Machine Learning Research (TMLR)*.

10 He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition. *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 770-778.

11 He, K., Zhang, X., Ren, S., & Sun, J. (2016). Identity Mappings in Deep Residual Networks. *European conference on computer vision (ECCV)*, 630-645.

12 He, K., Gkioxari, G., Dollár, P., & Girshick, R. (2017). Mask R-CNN. *Proceedings of the IEEE international conference on computer vision (ICCV)*, 2961-2969.

13 A. Krizhevsky, *Learning Multiple Layers of Features from Tiny Images*, University of Toronto, 2009. Available online: https://www.cs.toronto.edu/~kriz/cifar.html

A Appendix

A.1 ResNet50 Model Implementation

```

    import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms, models
from torch.utils.data import DataLoader, Subset
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import os

data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(20),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
    ]),
}

train_path = r"D:\6483\train"
val_path = r"D:\6483\val"

full_train_dataset = datasets.ImageFolder(train_path, data_transforms['
    train'])
full_val_dataset = datasets.ImageFolder(val_path, data_transforms['val'])


train_indices = np.random.choice(len(full_train_dataset), 10000, replace=
    False)
val_indices = np.random.choice(len(full_val_dataset), 2500, replace=False)

train_dataset = Subset(full_train_dataset, train_indices)
val_dataset = Subset(full_val_dataset, val_indices)

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

print(f"Training Set: {len(train_dataset)}")
print(f"Validation Set: {len(val_dataset)}")
print(f"Device: {device}")
```

```

model = models.resnet50(pretrained=True)

for param in model.parameters():
    param.requires_grad = False

# Using Sigmoid to classify
model.fc = nn.Sequential(
    nn.Linear(model.fc.in_features, 512),
    nn.ReLU(),
    nn.Dropout(0.5),
    nn.Linear(512, 1),
    nn.Sigmoid()
)

model = model.to(device)


criterion = nn.BCELoss()
optimizer = optim.SGD(model.fc.parameters(), lr=0.0001, momentum=0.9)

epochs = 30
patience = 3
best_loss = np.inf
counter = 0
train_losses, val_losses = [], []

for epoch in range(epochs):
    model.train()
    running_loss = 0.0
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.float().to(device).
            unsqueeze(1)

        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item() * inputs.size(0)
    epoch_loss = running_loss / len(train_loader.dataset)
    train_losses.append(epoch_loss)

    model.eval()
    val_loss = 0.0
    with torch.no_grad():
        for inputs, labels in val_loader:
            inputs, labels = inputs.to(device), labels.float().to(device).
                unsqueeze(1)
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            val_loss += loss.item() * inputs.size(0)

    epoch_val_loss = val_loss / len(val_loader.dataset)
    val_losses.append(epoch_val_loss)

    print(f'Epoch {epoch+1}/{epochs}.. Train Loss: {epoch_loss:.4f}.. Val
        Loss: {epoch_val_loss:.4f}')

    # EarlyStopping
    if epoch_val_loss < best_loss:
        best_loss = epoch_val_loss
        torch.save(model.state_dict(), 'best_model.pth')
        counter = 0
    else:
        counter += 1
        if counter >= patience:
            print("Early stopping!")
            break

plt.plot(train_losses, label='Train Loss')
plt.plot(val_losses, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Loss Curve')
plt.legend()
plt.show()

model.load_state_dict(torch.load('best_model.pth'))
model.eval()

test_transforms = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406],
                         [0.229, 0.224, 0.225])
])

test_dir = r"D:\6483\test"
test_images = sorted(os.listdir(test_dir), key=lambda x: int(''.join(filter
    (str.isdigit, x)) or -1))

results = []
with torch.no_grad():
    for idx, img_name in enumerate(test_images, start=1):
        img_path = os.path.join(test_dir, img_name)
        img = datasets.folder.default_loader(img_path)
        img = test_transforms(img).unsqueeze(0).to(device)

        output = model(img)
        label = 1 if output.item() > 0.5 else 0
        results.append([idx, label])

submission = pd.DataFrame(results, columns=['id', 'label'])
submission.to_csv('submission.csv', index=False)

print("submission.csv")
```

A.2 VGG16 Model Implementation

```

    import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms, models
from torch.utils.data import DataLoader, Subset
import numpy as np
from tqdm import tqdm
from torch.optim.lr_scheduler import StepLR
import matplotlib.pyplot as plt
import pandas as pd
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader
from torchvision import transforms
from torchvision.models import vgg16, VGG16_Weights
import os
import copy
from sklearn.metrics import precision_score, recall_score, f1_score

# ========== Data Preprocessing ==========
data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(20),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
    ]),
}

train_path = r"/home/sfmt/Downloads/6483/datasets/datasets/train"
val_path = r"/home/sfmt/Downloads/6483/datasets/datasets/val"

full_train_dataset = datasets.ImageFolder(train_path, data_transforms['
    train'])
full_val_dataset = datasets.ImageFolder(val_path, data_transforms['val'])

np.random.seed(42)
train_indices = np.random.choice(len(full_train_dataset), 2000, replace=
    False)
val_indices = np.random.choice(len(full_val_dataset), 500, replace=False)

train_dataset = Subset(full_train_dataset, train_indices)
val_dataset = Subset(full_val_dataset, val_indices)
print(val_dataset)

train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=16, shuffle=False)
```

```

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

# ========== Model Building ==========
vgg_model = models.vgg16(pretrained=True)

for param in vgg_model.features.parameters():
    param.requires_grad = False

vgg_model.classifier[6] = nn.Linear(4096, 2)
vgg_model = vgg_model.to(device)

# ========== Loss function & Optimizer ==========
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(vgg_model.classifier.parameters(), lr=0.00001)

# ========== Learning rate ==========
scheduler = StepLR(optimizer, step_size=10, gamma=0.1)

# ========== training set ==========

def train_model(model, train_loader, val_loader, criterion, optimizer,
    device, num_epochs=60, scheduler=None):
    train_losses = []
    val_losses = []

    #            Early Stopping
    patience = 5
    delta = 0.00001
    best_val_loss = float('inf')
    early_stop_counter = 0
    best_model_path = "vgg_catdog_best.pth"

    for epoch in range(num_epochs):
        print(f"\n--- Epoch {epoch+1}/{num_epochs} ---")

        model.train()
        running_loss = 0.0
        correct = 0

        train_bar = tqdm(train_loader, desc="Training", leave=False)
        for inputs, labels in train_bar:
            inputs, labels = inputs.to(device), labels.to(device)

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item() * inputs.size(0)
            _, preds = torch.max(outputs, 1)
            correct += torch.sum(preds == labels.data)

            current_lr = optimizer.param_groups[0]['lr']
            train_bar.set_postfix({
                'loss': loss.item(),
                'lr': f"{current_lr:.6f}"
            })

        epoch_loss = running_loss / len(train_loader.dataset)
        epoch_acc = correct.double() / len(train_loader.dataset)
        train_losses.append(epoch_loss)

        if scheduler:
            scheduler.step()

        tqdm.write(f"[Epoch {epoch+1}] Train Loss: {epoch_loss:.4f}, Train
            Acc: {epoch_acc:.4f}, LR: {current_lr:.6f}")

        model.eval()
        val_loss = 0.0
        val_correct = 0

        val_bar = tqdm(val_loader, desc="Validation", leave=False)
        with torch.no_grad():
            for inputs, labels in val_bar:
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = model(inputs)
                loss = criterion(outputs, labels)

                val_loss += loss.item() * inputs.size(0)
                _, preds = torch.max(outputs, 1)
                val_correct += torch.sum(preds == labels.data)

        val_loss /= len(val_loader.dataset)
        val_acc = val_correct.double() / len(val_loader.dataset)
        val_losses.append(val_loss)
        tqdm.write(f"[Epoch {epoch+1}] Val Loss: {val_loss:.4f}, Val Acc: {
            val_acc:.4f}")

        if val_loss < best_val_loss - delta:
            best_val_loss = val_loss
            early_stop_counter = 0
            torch.save(model.state_dict(), best_model_path)
        else:
            early_stop_counter += 1
            if early_stop_counter >= patience:
                print(f"Early stopping triggered at epoch {epoch+1}.")
                break

    plt.figure(figsize=(8, 6))
    plt.plot(range(len(train_losses)), train_losses, label='Train Loss')
    plt.plot(range(len(val_losses)), val_losses, label='Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Loss Curve')
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.savefig("loss_curve.png")
    plt.show()

# ========== test set ==========

```

```python
def predict_on_test(model, test_path, device, output_csv="test_predictions.csv"):
    from torch.utils.data import Dataset
    from PIL import Image
    import os

    class TestDataset(Dataset):
        def __init__(self, image_dir, transform=None):
            self.image_paths = sorted([
                os.path.join(image_dir, fname)
                for fname in os.listdir(image_dir)
                if fname.lower().endswith(('.png', '.jpg', '.jpeg'))
            ], key=lambda x: int(os.path.splitext(os.path.basename(x))[0]))
            self.transform = transform

        def __len__(self):
            return len(self.image_paths)

        def __getitem__(self, idx):
            img_path = self.image_paths[idx]
            image = Image.open(img_path).convert("RGB")
            filename = os.path.splitext(os.path.basename(img_path))[0]
            if self.transform:
                image = self.transform(image)
            return image, filename

    test_transform = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
    ])

    test_dataset = TestDataset(test_path, transform=test_transform)
    test_loader = DataLoader(test_dataset, batch_size=16, shuffle=False)

    model.eval()
    results = []

    with torch.no_grad():
        for inputs, filenames in tqdm(test_loader, desc="Predicting"):
            inputs = inputs.to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs, 1)
            predicted = predicted.cpu().numpy()
            for fname, pred in zip(filenames, predicted):
                results.append({"filename": fname, "label": int(pred)})

    df = pd.DataFrame(results)
    df.to_csv(output_csv, index=False)
    print(f"Prediction saved to {output_csv}")

# ========== F score ==========
def evaluate_predictions(pred_csv_path, groundtruth_csv_path):
    pred_df = pd.read_csv(pred_csv_path)
    gt_df = pd.read_csv(groundtruth_csv_path)

    pred_df = pred_df.sort_values("filename").reset_index(drop=True)
    gt_df = gt_df.sort_values("filename").reset_index(drop=True)

    y_pred = pred_df["label"].astype(int).tolist()
    y_true = gt_df["label"].astype(int).tolist()

    TP = TN = FP = FN = 0
    error_files = []

    for pred_label, true_label, fname in zip(y_pred, y_true, pred_df["
        filename"]):
        if pred_label == 1 and true_label == 1:
            TP += 1
        elif pred_label == 0 and true_label == 0:
            TN += 1
        elif pred_label == 1 and true_label == 0:
            FP += 1
            error_files.append(fname)
        elif pred_label == 0 and true_label == 1:
            FN += 1
            error_files.append(fname)

    precision = precision_score(y_true, y_pred, zero_division=0)
    recall = recall_score(y_true, y_pred, zero_division=0)
    f1 = f1_score(y_true, y_pred, zero_division=0)

    print("Prediction result ")
    print(f"TP: {TP}, TN: {TN}, FP: {FP}, FN: {FN}")
    print(f"Precision: {precision * 100:.2f}%")
    print(f"Recall:    {recall * 100:.2f}%")
    print(f"F1 Score:  {f1 * 100:.2f}%")
    print("wrong prediction indices:")
    for fname in error_files:
        print(f" - {fname}")

def evaluate_on_loader(model, loader, device):
    model.eval()
    y_true, y_pred = [], []

    with torch.no_grad():
        for inputs, labels in loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)
            y_pred.extend(preds.cpu().tolist())
            y_true.extend(labels.cpu().tolist())

    precision = precision_score(y_true, y_pred, zero_division=0)
    recall    = recall_score(y_true, y_pred, zero_division=0)
    f1        = f1_score(y_true, y_pred, zero_division=0)
    TP = sum((yt == 1 and yp == 1) for yt, yp in zip(y_true, y_pred))
    TN = sum((yt == 0 and yp == 0) for yt, yp in zip(y_true, y_pred))
    FP = sum((yt == 0 and yp == 1) for yt, yp in zip(y_true, y_pred))
    FN = sum((yt == 1 and yp == 0) for yt, yp in zip(y_true, y_pred))

    print("Validation result ")
    print(f"TP: {TP}, TN: {TN}, FP: {FP}, FN: {FN}")
    print(f"Precision: {precision * 100:.2f}%")
    print(f"Recall:    {recall * 100:.2f}%")
    print(f"F1 Score:  {f1 * 100:.2f}%")



# ====================
train_model(vgg_model, train_loader, val_loader, criterion, optimizer,
    device, num_epochs=60, scheduler = None)

vgg_model = models.vgg16(weights=VGG16_Weights.IMAGENET1K_V1)
vgg_model.classifier[6] = nn.Linear(4096, 2)
vgg_model.load_state_dict(torch.load("vgg_catdog_final.pth", map_location=
    device))
vgg_model = vgg_model.to(device)

evaluate_on_loader(vgg_model, val_loader, device)

test_path = r"/home/sfmt/Downloads/6483/datasets/datasets/test"
predict_on_test(vgg_model, test_path, device, output_csv="test_predictions.
    csv")
evaluate_predictions("test_predictions.csv", "groundtruth.csv")
```