**NATIONAL TECHNICAL UNIVERSITY OF UKRAINE**
**“IGOR SIKORSKY KYIV POLYTECHNIC INSTITUTE”,**
**MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE**

**NATIONAL TECHNICAL UNIVERSITY OF UKRAINE**
**“IGOR SIKORSKY KYIV POLYTECHNIC INSTITUTE”,**
**MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE**

Qualifying Scientific

work as a manuscript

XU JIASHU

UDC 004.032.26

**DISSERTATION**

# RESEARCH AND DEVELOPMENT OF SELF-SUPERVISED VISUAL FEATURE LEARNING BASED ON NEURAL NETWORKS

121 Software engineering

12 Information Technology

Submitted for the attainment of the Doctor of Philosophy degree

The dissertation contains the results of personal research. The use of ideas, results, and texts from other authors are accompanied by references to the respective sources.

Scientific Supervisor: Sergii Stirenko, Dr. Tech. Sc., Professor.

Kyiv – 2023

# АНОТАЦІЯ

***Сюй Цзяшу***. Дослідження та розробка самонавчання візуальним особливостям на основі нейронних мереж. - Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня доктора філософії за спеціальністю 121 - Інженерія програмного забезпечення та 12 - Інформаційні технології. - Національний технічний університет України "Київський політехнічний інститут імені Ігоря Сікорського", Київ, 2023.

Ця дисертація присвячена поглибленому дослідженню розробки та впровадження алгоритмів самонавчання, що являються частиною технік неконтрольованого навчання, які функціонують без потреби в маркованих даних. Ці алгоритми особливо вправні у попередньому навчанні моделей неконтрольованим способом, а отримані моделі показують результативність, порівнянну з їх контрольованими аналогами у широкому спектрі застосувань. Цей метод особливо корисний, оскільки він має на меті зменшити залежність від обширного маркування даних, характерного для парадигм глибокого навчання, тим самим підвищуючи ефективність і практичне застосування в різних сценаріях реального світу. Важливість алгоритмів самонавчання особливо підкреслена в області аналізу медичних зображень. У цій спеціалізованій області вимоги до анотування даних є не лише трудомісткими, але й потребують високої точності через критичну природу використовуваних даних. Складність отримання точних анотацій посилюється через дефіцит

спеціалістів, здатних їх забезпечити, що в свою чергу підкреслює трансформуючий потенціал підходів самонавчання в цій сфері.

У цій дисертації представлено новітню методологію самонавчання, що використовує Mixup Feature як мету реконструкції у межах pretext task. Це pretext task засноване на укладенні візуальних представлень через прогнозування Mixup Feature із маскованого зображення, використовуючи ці карти особливостей для вилучення високорівневої семантичної інформації. Дисертація детально розглядає роль Mixup Feature як прогностичної цілі у структурах самонавчання. Це дослідження включало детальну калібровку гіперпараметра λ, що є важливою для функціонування Mixup Feature. Ці налаштування дозволили створити комбіновані карти особливостей, що охоплюють карти детекції країв Sobel, гістограми орієнтованих градієнтів (HOG) та карти локальних бінарних шаблонів (LBP), забезпечуючи багатогранне представлення візуальних даних. Для практичного застосування цього нового методу як головної архітектури був обраний VIT (visual transformer), оскільки він ефективно обробляє складні візуальні вхідні дані та фокусується на важливих регіонах зображення. Цей вибір було додатково посилений висновками, отриманими з підходу Masked AutoEncoder (MAE), який виявив потенціал використання частково видимих вхідних даних для реконструкції повних зображень, таким чином покращуючи прогностичні здібності моделі в контексті самонавчання.

Розроблено модель denoising self-distillation Masked Autoencoder для самонавчання. Ця модель поєднує елементи з мереж Siamese Networks та

Masked Autoencoders, втілюючи трьохчастинну архітектуру, що включає student network у формі маскованого автокодера, проміжний regressor та teacher network. Основним проксі-завданням цієї моделі є відновлення вхідних зображень, які були штучно спотворені випадковими плямами гауссівського шуму. Це стратегічне рішення, призначене для стимулювання моделі вчитися стійким представленням особливостей, відокремлюючи чисті сигнали від шумних вхідних даних. Виконуючи це, модель навчається реконструювати деградоване зображення, ефективно вчиться концентруватися на сутності візуального контенту. Для забезпечення всебічного навчання модель застосовує механізм подвійної функції втрати. Одна функція налаштована на зміцнення глобального контекстуального розуміння зображення, що дозволяє моделі осягнути загальну структуру та конфігурацію сцени. Одночасно друга функція націлена на удосконалення сприйняття складних локальних деталей, гарантуючи, що тонкі візуальні нюанси не втрачаються під час дешумізації та реконструкції. Завдяки цьому інноваційний підхід, модель прагне досягнути делікатного балансу між макроскопічним сприйняттям візуальних сцен та детальною реконструкцією локалізованих деталей, балансу, який відіграє вирішальну роль для складних завдань аналізу зображень в рамках систем самонавчання.

Для оцінки експериментальної продуктивності двох інноваційних алгоритмів самонавчання було здійснено всебічний аналіз, зокрема застосований до трьох стандартних наборів даних: Cifar-10, Cifar-100 і STL-10. Це дослідження мало на меті порівняти ці алгоритми з сучасними передовими

техніками самонавчання, основаними на моделюванні з маскованими зображеннями. Порівняно з іншими сучасними методами самонавчання, що базуються на моделюванні з маскованими зображеннями, змішані картографічні характеристики HOG-Sobel, отримані за допомогою Mixup, показали видатні результати на Cifar-10 та STL-10 після full fine-tuning, з середнім підвищенням продуктивності на 0,4%. Крім того, переднавчена модель denoising self-distillation Masked Autoencoder (DMAE) була піддана ретельній оцінці. Після full fine-tuning, на наборі даних STL-10 ця модель продемонструвала невелику, але вагому перевагу над традиційним Masked Autoencoders (MAE), перевершуючи його продуктивність на 0,1%. Це відкриття підкреслює потенціал DMAE у покращенні точності моделі. Більше того, дослідження виявило, що в порівнянні з традиційними стратегіями самонавчання, які ґрунтуються на контрастному навчанні, метод Mixup Feature виявився ефективнішим. Він надав перевагу у вигляді скорочення часу навчання та усунення необхідності традиційних методів розширення даних, тим самим оптимізуючи процес навчання. В заключенні, два алгоритми самонавчання, введені в цьому дослідженні, сприяють розширенню набору методів для моделювання зображень із застосуванням масок. Їх доведена ефективність на контрольних наборах даних висвітлює їхній потенціал для більш широкого використання, зокрема в більших та складніших наборах даних.

Ефективне розширення застосування цих алгоритмів самонавчання охопило область аналізу медичних зображень. Таке розширення включало

застосування самонавчання з попереднім навчанням на спеціально підібраних наборах медичних зображень. Після фази попереднього навчання, розроблена таким чином модель була застосована для виконання наступних завдань. Емпіричні результати цього дослідження демонструють, що метод самонавчання з попереднім навчанням перевищує ефективність прямих методів навчання. Було спостережено відчутне підвищення точності, перевищуюче 5%, після Full fine-tuning моделі на двох наборах даних для наступних завдань.

Незбалансованість даних є вагомим викликом у аналізі медичних зображень, адже недостатнє представлення окремих станів або характеристик може негативно впливати на ефективність тренування моделей та екстракції ознак. З огляду на це, у дослідженні був розроблений незбалансований набір даних, а також проведено аналіз стійкості самонавчальних попередньо тренованих моделей у контексті незбалансованості даних. Експериментальні результати виділяють перевагу стійкості методів самонавчання з попереднім тренуванням над моделями, навченими з нуля, у подоланні проблем незбалансованості даних. Ці результати засвідчують ефективність наших запропонованих самонавчальних попередньо тренованих моделей у розв'язанні проблем незбалансованості наборів даних. Відчутне покращення стійкості алгоритмів самонавчання розширює їх можливості як ефективних інструментів у аналізі медичних зображень, натякаючи на перспективне збільшення точності в системах інтелектуальної підтримки діагностики.

***Ключові слова***: Self-supervised learning, Реконструкція зображення, Видобуток особливостей, Виявлення краю зображення, Маскований автоенкодер, Візійні трансформатори, Мережі Сіамських, Аналіз медичних зображень.

## ABSTRACT

***Xu Jiashu***. Research and development of self-supervised visual feature learning based on neural networks. - Qualified scientific work on the rights of the manuscript.

Dissertation for the degree of Doctor of Philosophy in the specialty 121 - Software Engineering and 12 - Information Technologies. - National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, 2024.

This Dissertation focuses on in-depth exploration into the design and development of self-supervised learning algorithms, which are a subset of unsupervised learning techniques that operate without the need for labeled datasets. These algorithms are particularly adept at pre-training models in an unsupervised manner, with the resultant models demonstrating performance on par with their supervised counterparts across a range of downstream applications. This method is particularly advantageous as it aims to mitigate the over-dependence on extensive data labeling that is typical within deep learning paradigms, thereby enhancing efficiency and practical utility in diverse real-world scenarios. The pertinence of self-supervised learning algorithms is especially highlighted within the realm of medical image analysis. In this specialized field, the requisites for data annotation are not only laborious but also require a high degree of precision due to the critical nature of the data involved. The difficulty of obtaining accurate annotations is compounded by the scarcity of specialists capable of providing them, which in turn underscores the transformative potential of self-supervised learning approaches within this domain.

In this dissertation, a cutting-edge self-supervised learning methodology is delineated, which employs the Mixup Feature as the reconstruction target within the

pretext task. This pretext task is fundamentally designed to encapsulate visual representations by the prediction of Mixup features from masked image, utilizing these feature maps to extracting high-level semantic information. The dissertation delves into the validation of the Mixup Feature's role as a predictive target in self-supervised learning frameworks. This investigation involved the meticulous calibration of the hyperparameter λ, integral to the Mixup Feature operation. Such adjustments allowed for the generation of amalgamated feature maps that encompass Sobel edge detection maps, Histogram of Oriented Gradients (HOG) maps, and Local Binary Pattern (LBP) maps, providing a rich, multifaceted representation of visual data. For the empirical application of this novel method, the visual transformer was selected as the principal architecture, due to its proficiency in handling complex visual inputs and its emphasis on critical image regions. This choice was further reinforced by the insights derived from the Masked AutoEncoder (MAE) approach, which illuminated the potential of utilizing partially visible inputs to reconstruct full images, thus enhancing the model's predictive capabilities in a self-supervised context.

A denoising self-distillation Masked Autoencoder model for self-supervised learning was developed. This model synthesizes elements from Siamese Networks and Masked Autoencoders, incorporating a tripartite architecture that includes a student network in the form of a masked autoencoder, an intermediary regressor, and a teacher network. The underlying proxy task for this model is the restoration of input images that have been artificially corrupted with random Gaussian noise patches. This is a strategic choice designed to encourage the model to learn robust feature representations by distilling clean signals from noisy inputs. In doing so, the model is

trained to reconstruction of the degraded image, effectively teaching it focus on the essence of the visual content. To ensure comprehensive learning, the model harnesses a dual loss function mechanism. One function is calibrated to reinforce the global contextual understanding of the image, thereby enabling the model to grasp the overall structure and scene configuration. Concurrently, the second function is tailored to refine the perception of intricate local details, ensuring that fine visual nuances are not lost in the process of denoising and reconstruction. Through this innovative approach, the model aspires to achieve a delicate balance between the macroscopic comprehension of visual scenes and the meticulous reconstruction of localized details, a balance that is pivotal for sophisticated image analysis tasks in self-supervised learning frameworks.

An exhaustive analysis was executed to assess the experimental performance of two innovative self-supervised learning algorithms, specifically applied to three benchmark datasets: Cifar-10, Cifar-100, and STL-10. This study aimed to benchmark these algorithms against existing advanced self-supervised techniques grounded in Masked Image Modeling. In comparison to other state-of-the-art self-supervised methods based on Masked Image Modeling, the mixed HOG-Sobel feature maps obtained using Mixup showed outstanding performance on Cifar-10 and STL-10 after full fine-tuning, with an average performance improvement of 0.4%. Additionally, the pre-trained model of the Deep Masked Autoencoder (DMAE) was subjected to a rigorous evaluation. When full fine-tuned on the STL-10 dataset, this model demonstrated a modest yet significant edge over the conventional Masked Autoencoder (MAE), exceeding its performance by a margin of 0.1%. This finding

shed light on the potential of DMAE in enhancing model accuracy. Moreover, the study revealed that in comparison to traditional self-supervised learning strategies reliant on contrastive learning, the Mixup Feature method emerged as more efficient. It offered the advantage of shortened training durations and negated the requirement for conventional data augmentation methods, thus streamlining the learning process. In conclusion, the two self-supervised learning algorithms introduced in this research contribute to the expanding repertoire of methods for masked image modeling. Their demonstrated effectiveness on benchmark datasets illuminates their potential for broader applications, particularly in larger and more complex datasets.

The application of these self-supervised learning algorithms was effectively expanded to encompass the domain of medical image analysis. This extension involved the utilization of self-supervised pre-training on specifically curated medical image datasets. Following this pre-training phase, the model thus developed was then employed for the downstream tasks. Empirical results from this study illustrate that the approach of self-supervised pre-training surpasses the efficacy of direct training methodologies. A notable enhancement in accuracy, exceeding 5%, was observed upon the Full fine-tuning of the model on the two downstream datasets.

Data imbalance poses a substantial challenge in medical image analysis, as inadequate representation of specific conditions or features can negatively impact the efficacy of model training and feature extraction. Considering this, the study developed an imbalanced dataset and delved into the robustness of self-supervised pre-trained models in the context of data imbalance. The experimental findings underscore the superior robustness of self-supervised pre-training methods over from

scrath trained models in addressing data imbalance issues. Particularly notable is their performance in scenarios with a positive to negative sample ratio of 1:8, where they exhibit enhanced robustness compared to traditional supervised Convolutional Neural Network (CNN) pre-trained models. These results affirm the effectiveness of our proposed self-supervised pre-trained models in tackling dataset imbalance challenges. The notable improvement in the robustness of self-supervised learning algorithms augments their potential as powerful tools in medical image analysis, suggesting a prospective enhancement in accuracy within intelligent assisted diagnostic systems.

***Keywords***: Self-supervised learning, Image reconstruction, Feature extraction, Image edge detection, Masked Autoencoder, Vision Transformers, Siamese Networks, Medical image analysis.

## LIST OF PUBLICATIONS BY THE AUTHOR

*Scientific publications in which the main research findings of the dissertation are published:*

1. **Jiashu Xu** and Sergii Stirenko, (2023) "Mixup Feature: A Pretext Task Self-Supervised Learning Method for Enhanced Visual Feature Learning," in IEEE Access, vol. 11, pp. 82400-82409, IEEE, ISSN: 2169-3536, DOI: 10.1109/ACCESS.2023.3301561 (Scopus Q1, WoS Q2).
2. **Jiashu Xu**, Sergii Stirenko, (2023) "Denoising Self-Distillation Masked Autoencoder for Self-Supervised Learning", International Journal of Image, Graphics and Signal Processing (IJIGSP), Vol.15, No.5, pp. 29-38. MECS Press, ISSN:2074-9074, DOI:10.5815/ijigsp.2023.05.03 (Scopus)
3. **Jiashu Xu**, Sergii Stirenko, (2022) "Self-Supervised Model Based on Masked Autoencoders Advance CT Scans Classification", International Journal of Image, Graphics and Signal Processing (IJIGSP), Vol.14, No.5, pp. 1-9. MECS Press, ISSN:2074-9074, DOI:10.5815/ijigsp.2022.05.01 (Scopus)
4. **Jiashu Xu**. (2021) "A review of self-supervised learning methods in the field of medical image analysis." International Journal of Image, Graphics and Signal Processing (IJIGSP) 13, no. 4: 33-46. MECS Press, ISSN:2074-9074, 10.5815/ijigsp.2021.04.03 (Scopus).
5. Yahu Yang, Hu Zhang, **Jiashu Xu**, Shenmin Song, (2023), “MATEKG: A Large-scale Multi-class Equipment Knowledge Graph for Military Auxiliary

Tasks.” 2023 IEEE 6th International Conference on Information Systems and Computer Aided Education (ICISCAE), Dalian, China. (Scopus).

6. Yang, Ya-Hu, **Jiashu Xu**, Yuri Gordienko, and Sergii Stirenko. (2021). "Abnormal Interference Recognition Based on Rolling Prediction Average Algorithm." Advances in Computer Science for Engineering and Education III. ICCSEEA 2020. Advances in Intelligent Systems and Computing, vol 1247. Springer, Cham. https://doi.org/10.1007/978-3-030-55506-1_28 (Scopus)
7. **Jiashu Xu**, Sergii Stirenko, (2020), "FACIAL EXPRESSION RECOGNITION SYSTEM BASED ON GAN NETWORK DATA AUGMENTATION", The International Conference on Security, Fault Tolerance, Intelligence 2020, pp. 144-149.

# CONTENTS

# LIST OF ABBREVIATIONS

**SSL – Self-Supervised Learning**

**MIM – Masked Image Modeling**

**CV – Computer Vision**

**NLP – Natural Language Processing**

**ViT – Vision Transformer**

**NT-Xent – Normalized Temperature-Scaled Cross-Entropy**

**MAE – Mask Auto-Encoders**

**DMAE – Denoising Mask Auto-Encoders**

**EMA – Exponential Moving Average**

**CT – Computerized Tomography**

**AUC – Area Under the Curve**

**CNN – Convolutional Neural Networks**

**RLHF – Reinforcement learning from human feedback**

# INTRODUCTION

## Relevance of the Topic.

In recent years, self-supervised learning (SSL) has made remarkable strides in the field of deep learning, even being referred to as "the dark matter of intelligence" [1]. Traditional unsupervised learning methods face challenges in extracting high-quality feature representations. To address the challenge of learning universal feature representations from a large amount of unlabeled data, the approach of self-supervised learning emerged. These methods use the inherent features of data to design proxy tasks that generate pseudo-labels, enabling models to learn generic visual representations for further transfer learning and reinforcement learning based on human feedback (RLHF) [93]. The fundamental concept of SSL is shown in Figure 1.

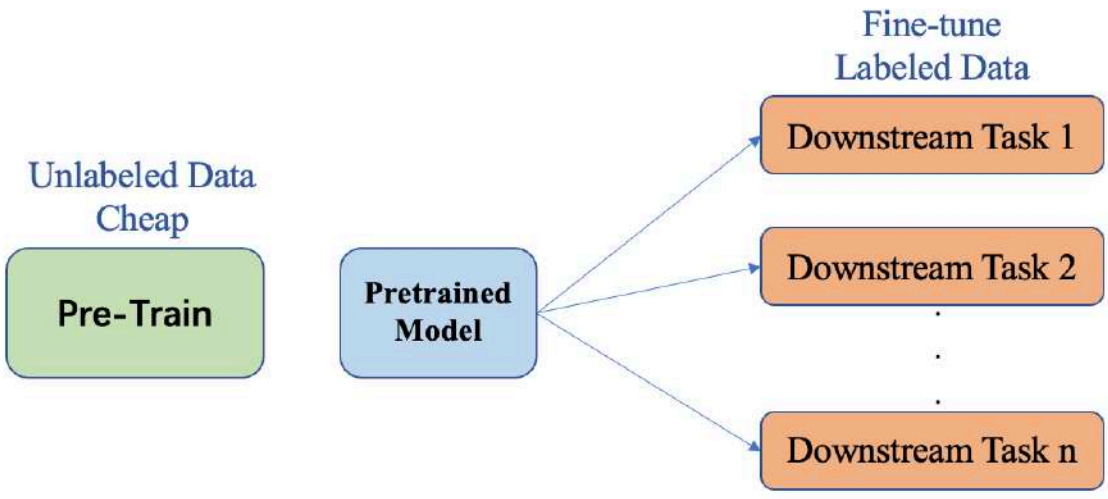



Figure 1 — The fundamental concept of SSL.

In the field of natural language processing (NLP), SSL methods have been employed to train large-scale models on vast, unlabeled text corpora from the internet,

leading to the emergence of notable language models such as ChatGPT [2], PaLM [3], and Claude. A common proxy task in NLP involves masking words in the text and predicting the masked words, enabling the models to capture relationships between words without relying on explicit labels. In computer vision, large-scale CV models like SEER [4] and SAM [5] can learn from unlabeled images available on the internet and achieve state-of-the-art performance on a range of computer vision benchmarks. These advancements owe to the success of self-supervised learning methods. By utilizing abundant unlabeled data for training, SSL facilitates the learning of general representations that can be applied to various tasks and even enables cross-domain transfer learning. Notably, SSL is particularly valuable in domains such as medicine, where labeling costs are prohibitively high.

Although SSL has made tremendous progress, the development of universal large-scale models in the computer vision (CV) field is still in its early stages, and self-supervised learning for image understanding remains challenging. The focus of this research is to develop novel self-supervised learning methods that complement existing algorithms in the visual domain, aiming to drive advancements in self-supervised learning for visual feature extraction.

**The connection of the work with scientific programs, plans, and topics.**

The topic of the dissertation is included in the scientific work plan approved by the Department of Computer Science at Igor Sikorsky Kyiv Polytechnic Institute, taking into account the decree of the Cabinet of Ministers of Ukraine dated December 2, 2020, No. 1556-p on the approval of the Concept of Artificial Intelligence Development in Ukraine. The methods proposed in the dissertation were used in the

scientific research work of the National University of Defense of Ukraine "Science for Human and Society Security" - the project "Artificial Intelligence Platform for Remote Automated Detection and Diagnosis of Human Diseases" project registration number: 2020.01/0490.

**The purpose and objectives of the research.**

The aim of this research is to develop a self-supervised learning framework for extracting generic visual features without requiring labeled data. By leveraging pretraining models through self-supervised learning, these models can be applied to downstream tasks such as medical image analysis and diagnostics, thereby enhancing their accuracy.

To achieve this goal, the following tasks need to be addressed:

- Designing self-supervised learning algorithms is the most critical challenge. It enables learning without labels, including self-supervised learning of local features and global structures in images. Ensure that the designed self-supervised models can learn generalizable visual features.
- Design appropriate model architectures that can effectively learn from unlabeled data. Utilize ViT architectures［6］and autoencoder models.
- Define effective self-supervised tasks and loss functions to drive model learning.
- Apply transfer learning and fine-tune the pre-trained models by transferring them to downstream medical image analysis and diagnosis tasks.

- Perform performance evaluation and visualization analysis to analyze the learned features for their generalizability and efficacy through visualization and quantitative metrics. Evaluate model performance on downstream tasks.
- Investigate interpretability through ablation studies to analyze important factors affecting the results, including self-supervised tasks, loss functions, model architectures, etc.

**Research object**

The research objective of this dissertation is to develop innovative self-supervised learning methodologies that empower the resulting models to acquire visual features with robust generalizability, particularly suited for downstream tasks in the context of medical image classification.

**Research topic**

The research topic focuses on the development of self-supervised visual feature learning algorithms based on deep neural networks.

**Research Methodology.**

This research will focus on developing a novel self-supervised learning algorithm for visual feature learning, in order to advance self-supervised representation learning in the field of computer vision. The key methodology of this research will involve designing self-supervised learning algorithms for unlabeled image data, constructing appropriate model architectures, defining effective loss functions, applying transfer learning, evaluating learned representations, conducting ablation studies, and comparing the developed method with state-of-the-art self-supervised learning algorithms through comprehensive evaluations. The core

emphasis will be on the development and evaluation of a self-supervised learning framework.

**Scientific Novelty of the Obtained Results:**

- A novel self-supervised learning methodology leveraging the Mixup Feature function is introduced. This method involves the pre-training of visual representations by predicting Mixup features from masked images, which stand in as proxies for advanced semantic information. The approach is poised to potentially bolster the aggregate efficacy of the model.

- A masked autoencoder model for self-supervised learning is proposed, featuring novel mechanisms for noise suppression and self-distillation. The architecture utilizes a masked autoencoder in conjunction with a teacher network to facilitate the reconstruction of corrupted image segments afflicted with random Gaussian noise. This model extends the utility of self-supervised techniques in restoring visual data.

- For the first time, a model is proposed that, by combining losses at the pixel level and feature level, enables the extraction of deep semantic characteristics of the image. This complements existing techniques for modeling masked images and also increases the robustness of self-supervised learning models when working with unbalanced data sets.

**The Practical Significance of the Obtained Results.**

The present study proposes innovative approaches and novel models in the field of computer vision. The proposed self-supervised learning method has practical

applications in downstream tasks, such as medical image processing, and has achieved results comparable to those of supervised learning experiments, indicating its potential for practical use. This method can supplement existing self-supervised learning methods for masked image modeling and may be applied to larger datasets. Overall, the results of this study demonstrate the potential of self-supervised learning in the field of medical image classification and provide new methods and models that offer important insights and guidance for future research in computer vision.

**Contributions of the author.**

This dissertation is a product of independent scientific research, integrating two self-supervised learning methods proposed by the author and applying the self-supervised pre-trained models to medical image processing. The scientific methodologies and key findings presented in the dissertation were independently derived by the applicant throughout their research endeavors. In the works where the applicant is a co-author, the publications include:

[1] Development of a feature learning self-supervised approach, "Mixup Feature: A Pretext Task Self-Supervised Learning Method for Enhanced Visual Feature Learning".

[2] Development of a denoising self-supervised method, "Denoising Self-Distillation Masked Autoencoder for Self-Supervised Learning".

[3] Development of a self-supervised model leveraging masked autoencoders for enhanced CT scan classification.

[4] Conducted the first comprehensive survey of the application and research of self-supervised learning algorithms in the medical imaging field, filling the gap in this area.

[5] Development of an anomaly interference detection method utilizing rolling prediction average algorithms.

**Approbation of the results of the dissertation.**

The primary outcomes of this research have been disseminated and deliberated in international and IEEE journals, as well as at IEEE scientific conferences, specifically in venues such as: the "IEEE Access" journal; the "International Journal of Image, Graphics and Signal Processing (IJIGSP)"; the "IEEE 6th International Conference on Information Systems and Computer Aided Education (ICISCAE)", held in Dalian, 2023; the "3rd International Conference on Computer Science, Engineering, and Education Applications (ICCSEEA)", held in Kyiv, 2020.

**Publications.**

Based on the findings of the study, 7 scientific papers have been published, and they have been indexed in the following international citation databases: Scopus - 6, Web of Science - 1, and EI Compendex - 2.

**Structure and scope of work.**

The paper consists of an introduction, four chapters, a conclusion, a list of 94 references, and appendices. The total number of pages in the paper is 168, with 101 pages for the main part and two appendices totaling 35 pages. It includes 30 figures, 16 equations, and 10 tables.

# CHAPTER 1. SELF-SUPERVISED LEARNING ALGORITHMS FOR VISUAL FEATURES

Since 2020, the emergence of ultra-large datasets, coupled with the availability of high-performance, high-memory GPUs, has led to a resurgence of interest in Self-Supervised Learning (SSL) methods. However, SSL is not a recent development but can be traced back to the early stages of deep learning. Today's SSL methods are built upon the knowledge derived from the experiments of early researchers.

In this section, the main concepts underpinning Self-Supervised Learning (SSL) prior to 2020 are succinctly delineated. While many of these specific methods are no longer mainstream due to their inability to deliver state-of-the-art performance on benchmark problems, the ideas within these papers have nonetheless contributed significantly to the formation of many modern methods. The fundamental principle of SSL algorithms is to devise a proxy task that leverages the data itself to generate pseudo-labels. Based on various proxy tasks, existing self-supervised learning algorithms can be broadly classified into the following categories:

- **Information restoration**. Training neural networks to recover missing or damaged image information. For example, one common strategy is to convert an image to a grayscale and predict the original RGB values as a proxy task [7, 8]. Another approach involves masking or removing a portion of the image and then reconstructing the missing pixel values as a proxy task [9].

- **Learning spatial context information**. This category of proxy tasks involves training models to comprehend the relative positions and orientations of objects within images. For instance, the model is subjected to random rotations of the original images and is then tasked to predict the rotation angles [10]. Another approach is to partition the images into disjointed blocks, randomize their arrangement, and employ a 'jigsaw' method to predict the relative positions of each block [11], An example is shown in Figure 1.1.
- **Clustering for images**. Clustering semantically similar images together can facilitate the acquisition of rich features. Drawing inspiration from the classical machine learning algorithm, K-means clustering, we can implement SSL through neural models. Deep clustering [12] involves performing k-means in the feature space to assign Pseudo-label to the images.
- **Generative methods.** Generation-based approaches capitalize primarily on the feature learning capabilities of autoencoders, such as denoising autoencoders [13], deep canonically correlated autoencoders [14], and Split-brain autoencoders [15]. Generative adversarial networks (GANs) [16] consist of an image generator and a discriminator. This model is trained in an unsupervised manner and can learn feature representations useful for transfer learning. Early GANs SSL [17] attempted to use GAN components for downstream image classification.
- **Multi-view invariance**. In recent years, many SSL methods create proxy tasks using contrastive learning, encouraging the model to output similar

feature representations for simple image transformations. This maximizes the mutual information between feature representations of the image under different views [18].

Based on these classifications, I have grouped these SSL categories into four major families: the contrastive learning family, the masked image modeling family, the self-distillation family, and the canonical correlation analysis family. Each of these families will be discussed in the following subsections.

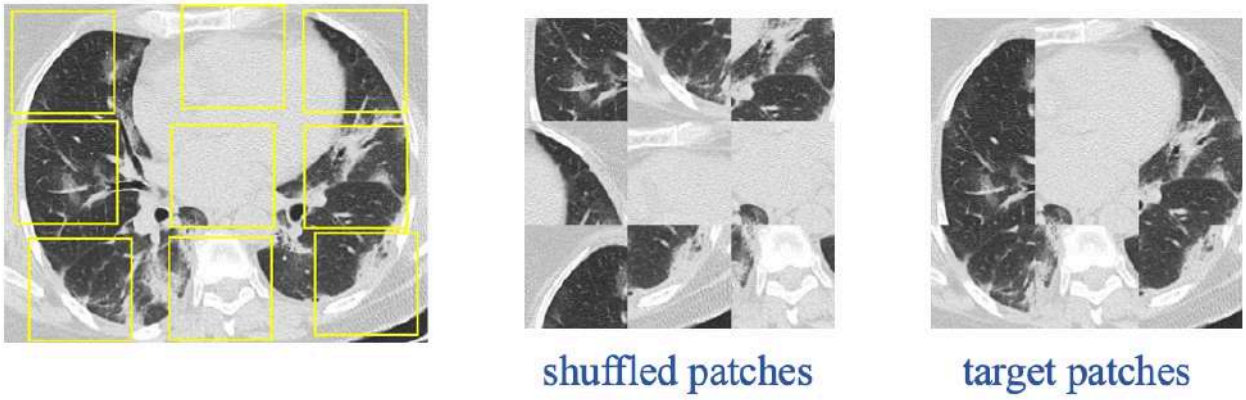



Figure. 1.1 An example of the “jigsaw” pretext task on an CT image.

## 1.1 Contrastive Learning Family

Contrastive learning methods aim to learn a representation learning model by automatically constructing pairs of similar and dissimilar instances, such that similar instances are projected close to each other in the embedding space while dissimilar instances are projected far apart. For instance, consider a set of four images, as depicted in Figure 1.2. The first two images belong to the 'dog' category, while the subsequent two images pertain to different categories. Taking the first image as an exemplar, our objective is to maximize its similarity with the second image, which

also belongs to the 'dog' category, while minimizing its similarity with the third and fourth images, which belong to distinct categories.

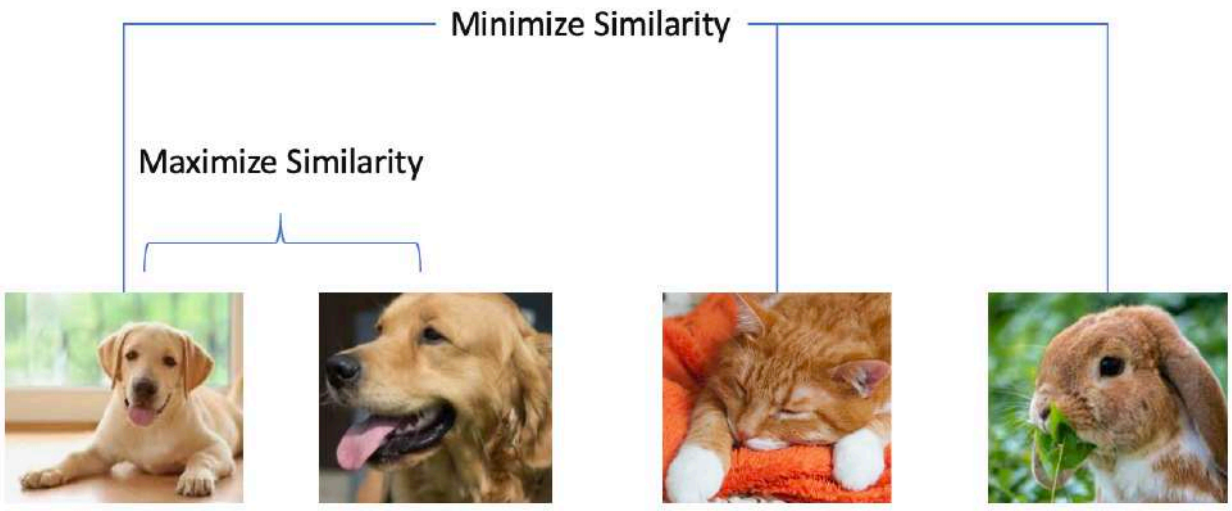



Figure. 1.2 Contrastive learning attempts to teach models to distinguish between similar and dissimilar entities.

The contrastive loss was first introduced in 1993 [19] and was later formally defined in 2006 [20]. Since the data is unlabeled, we often use data augmentation to preserve the semantic information and form positive samples for a single instance, while different samples are considered negative. The contrastive loss sets a margin parameter $m$, requiring that the distance between negative samples should be greater than $m$. Similar to the contrastive loss, triplet loss calculates the similarity between samples by optimizing the distance between the anchor example and the positive example to be less than the distance between the anchor example and the negative example [21]. Compared to contrastive loss, triplet loss only requires that the final optimization objective brings the anchor and positive closer while pushing the anchor and negative examples further apart, i.e., the similarity difference between the anchor-positive and anchor-negative pairs is greater than a margin $m$.

When selecting sample pairs, triplet loss can only consider a single negative sample pair, which limits its ability to distinguish between samples from other classes. This can result in instability and slow convergence. To address this issue, N-pair loss selects multiple negative sample pairs by combining a positive sample pair with all samples from different classes to form negative sample pairs [22]. For a dataset with N classes, each positive sample pair corresponds to N-1 negative sample pairs. A similar approach is taken in contrastive predictive coding (CPC) loss, which has led to the emergence of contrastive learning in the field of self-supervised learning (SSL) [23]. CPC has been extended to the image domain [24], with a key element being the introduction of InfoNCE loss [25], which has become a core element of SSL.

SimCLR [27] is one of the seminal works in the field of self-supervised learning (SSL), focusing on contrastive learning. SimCLR employs a specific approach by maximizing the similarity between two augmented views of a sample to learn visual representations. The augmented views are created using common data augmentation techniques, such as random resizing, cropping, color jittering, and random blurring, among others. Its framework is illustrated in Figure 1.3. Consider an arbitrary image $X$, referred to as the Original Image. Firstly, data augmentation is applied to $X$, resulting in two augmented images, $x_i$ and $x_j$. Subsequently, the augmented images $x_i$ and $x_j$ are fed into an Encoder, where the two Encoders share the same parameters, producing respective representations $h_i$ and $h_j$. Further, these representations $h_i$ and $h_j$ are passed through a Projection Head, which also shares same parameters. The Projection Head, often implemented as a Multi-Layer Perceptron (MLP) followed by ReLU activations [28], maps the initial embeddings to another space. In this space, a

contrastive loss is applied to maximize the similarity between the representations $z_i$ and $z_j$ obtained from the same image, aiming to encourage enhanced agreement between $z_i$ and $z_j$. After completing the pre-training phase, the Projection Head is discarded, while retaining the feature extraction capability of the Encoder, which is then utilized for fine-tuning on downstream tasks.

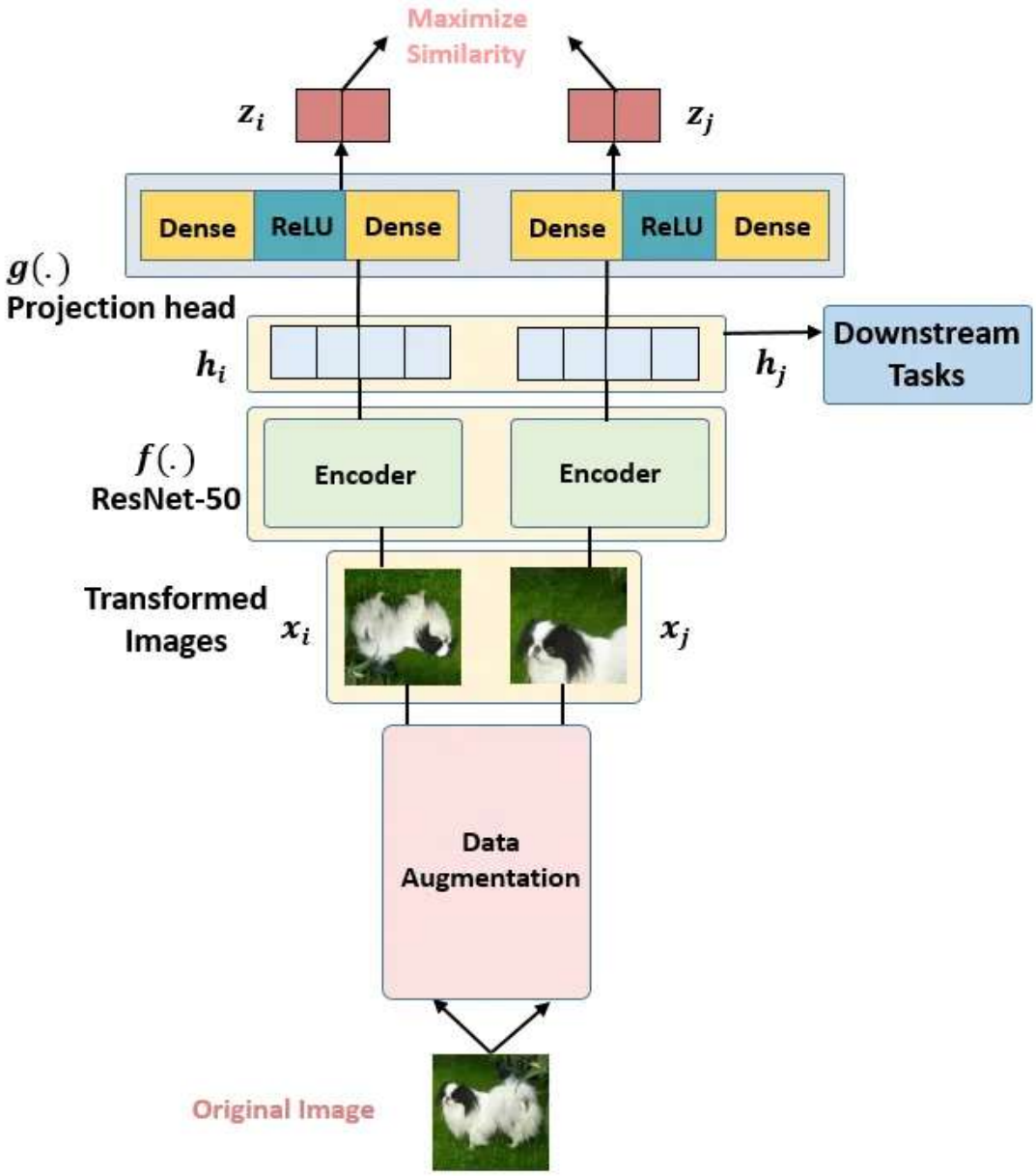



Figure. 1.3 The framework of SimCLR, the picture is sourced from [26].

SimCLR defines the similarity between representations using Cosine Similarity and transforms it into a Loss Function that can be optimized. SimCLR employs a

contrastive learning loss function called NT-Xent loss [27], which utilizes Non-parametric SoftMax [29] as a key ingredient. For SimCLR, the number of negative samples determines the model's ultimate feature learning capability, thus relying on a relatively large batch size. However, due to computational limitations, the batch size cannot be set too large, making it difficult to incorporate a large number of negative samples. To address this dilemma, a larger memory bank is employed to store the representations of all samples [29]. However, the updating speed of the memory bank is slower than that of the encoder, leading to inconsistency. MoCo [30] utilizes momentum (moving average updating of model weights) and a queue (dictionary) to effectively address the inconsistency problem and avoid the issue of insufficient negative samples.

In further work, researchers attempted to perform contrastive learning using only positive pairs, which is intuitively unrealistic since the model is likely to collapse all representations to a single constant value by only minimizing the distance between positive pairs without increasing the distance to negative samples. This would result in zero loss but the loss of any information in the representation. To avoid representation collapse, an asymmetric self-distillation structure was employed, enabling effective contrastive learning with only positive pairs. Several methods for self-distillation-based contrastive learning are described in the 1.3 subsection.

## 1.2 Masked Image Modeling Family

“What I cannot create, I do not understand.” — Richard Feynman

In the early days of the CV field, many self-supervised pre-training proxy tasks applied degradation techniques to training images, such as decolorization [7] and noise [13]. Additionally, context encoders were used for the restoration of occluded images, where the majority of pixel values in an image were replaced with white, training autoencoders to recover the original pixels overlaid with white. Context encoders were early attempts at masked image modeling, based on convolutional neural network (CNN) backbone networks, prior to the emergence of visual transformer architectures. These methods, limited by network architecture constraints, did not achieve competitive performance on downstream tasks. In the NLP field, the BERT method [31] has achieved significant success in downstream tasks through the use of Masked Language Modeling (MLM) for self-supervised pre-training.

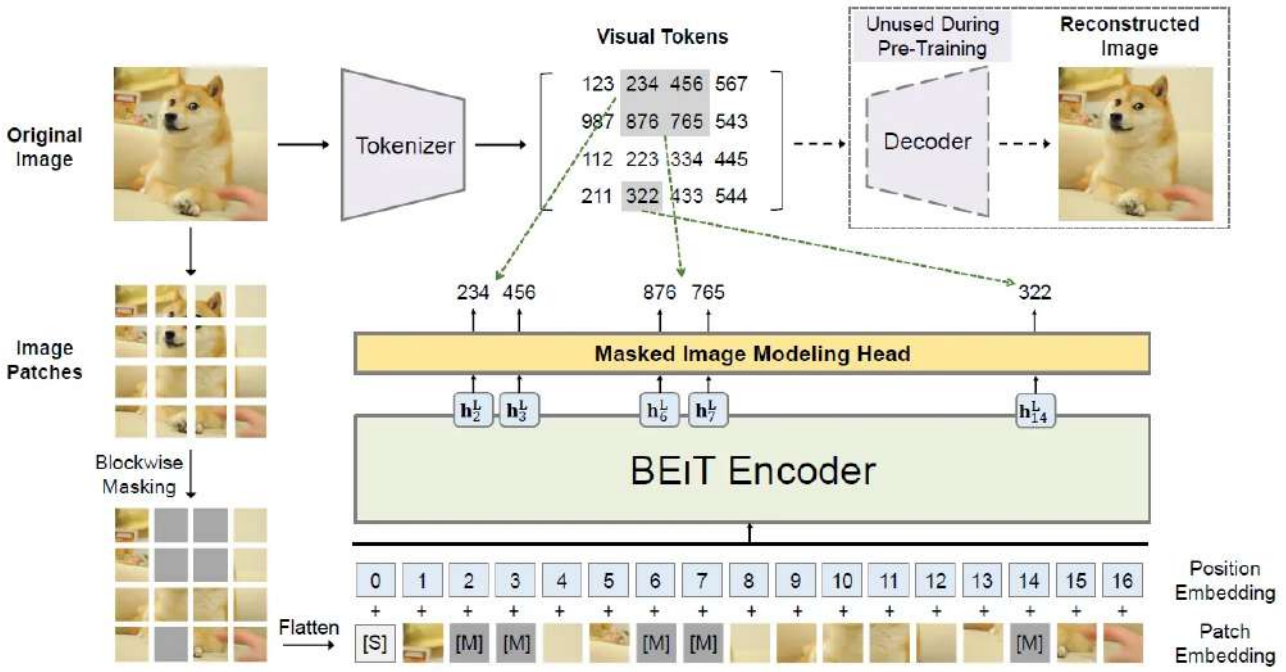



Figure. 1.4 Overall framework of BEIT [32].

Inspired by BERT, a visual Transformer architecture called ViT [6] has emerged in the CV field, where models are trained to directly predict missing pixel values, but

it was found that the effectiveness of this pre-training strategy is significantly lower than that of supervised pre-training. Since there is no large vocabulary in visual tasks, directly applying the BERT strategy to images is challenging. BEiT [32] regards MIM as a regression problem, where the Encoder input is image patches, and the output is visual tokens. The aim is to make the visual tokens output by the masked positions as close as possible to the true visual tokens, which are obtained through additional training of discrete variational autoencoder (dVAE). Compared with previous supervised and self-supervised baselines, BEiT has achieved significant improvements in downstream image classification and semantic segmentation performance. However, its training pipeline is complex because it requires an additional dVAE to convert image patches into visual tokens. The overall structure of BEIT is illustrated in Figure1.4.

In subsequent work, both MAE [33] and SimMIM [34] have simplified the MIM pre-training process. Unlike BEiT, which requires dVAE to extract additional visual tokens, MAE directly reconstructs the masked image patches, using the error between the predicted results and the actual image patches as a loss. Experiments on downstream tasks such as image classification, semantic segmentation, and object detection have shown that these pre-training strategies perform better than BEiT. Direct reconstruction of the original image can also yield impressive results in self-supervised learning.

The MAE method is akin to Denoising Auto-Encoders (DAE), in which the input signal is disrupted, and the model learns to reconstruct the original, undisturbed signal. The encoder and decoder structure in MAE differs and is asymmetrical. The

encoder encodes the input into a latent representation, and the decoder rebuilds the original signal from this latent representation. MAE, like ViT, divides images into regular, non-overlapping patches. Some patches are then randomly selected, and the remaining patches are masked, following a uniform distribution. By utilizing a sufficiently high mask ratio, the redundancy of the patch information is significantly reduced, making the reconstruction of images in this context less straightforward. The MAE self-supervised learning architecture is shown in Figure 1.5.

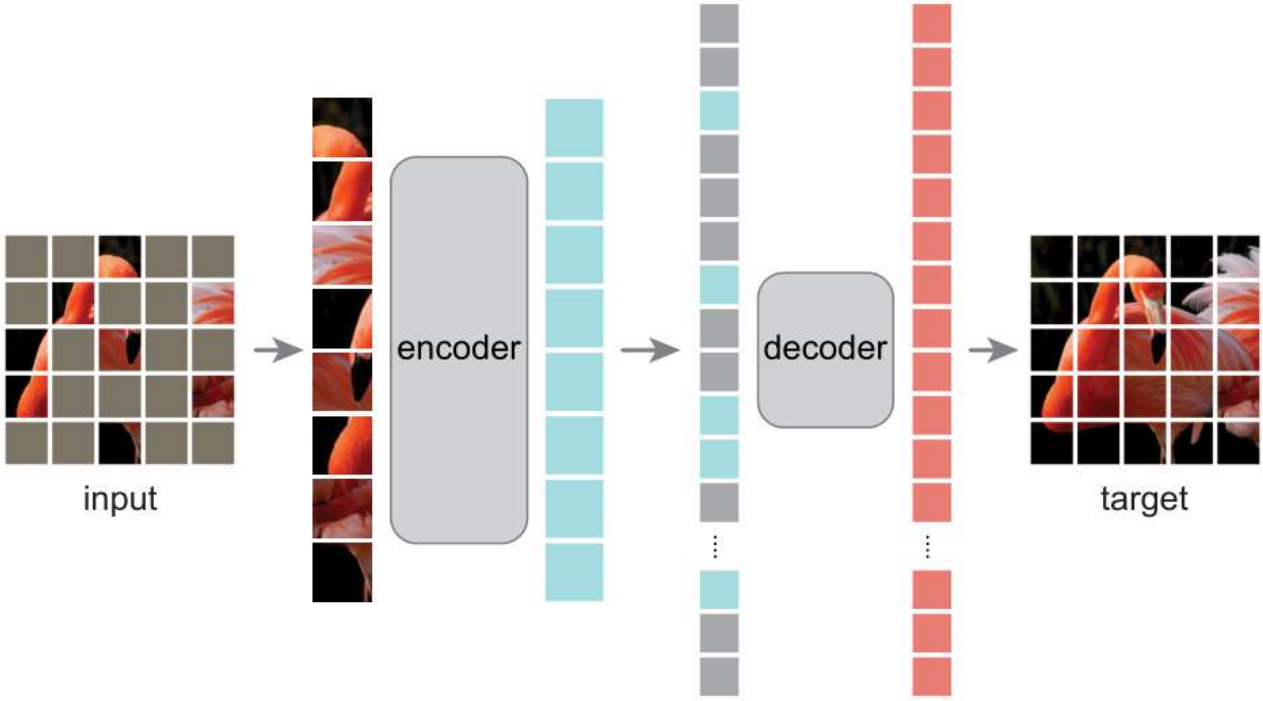



Figure. 1.5 MAE self-supervised learning architecture [33].

The masking strategy of SimMIM contrasts with that of MAE. Instead of discarding the masked patches as MAE does, SimMIM, akin to BEiT and BERT, replaces masked patches with a learnable mask token vector and trains them along with the network. The basic unit of masking remains the image patches. For the ViT model, the size of the masked patch is 32×32. The SimMIM and MAE methods share a common approach of randomly masking image patches and directly regressing the

original pixel RGB values for prediction. Additionally, the decoder model is lightweight. Inspired by MAE and SimMIM, MIM-based self-supervised learning has achieved competitive performance in various visual downstream tasks [35,36] and even in visual language representation learning [37].

Overall, MIM-based self-supervised learning methods have achieved competitive experimental results. Some methods combine masked image modeling with self-distillation, which will be discussed in the next subsection on the self-distillation family. Our proposed Mixup Feature method [67], also inspired by MIM modeling, will be detailed in Chapter 2.

### 1.3 Self-Distillation Family

Self-distillation methods, such as BYOL [38], SimSIAM [39], DINO [40], and their variants, employ a simple mechanism to achieve self-supervised learning. This mechanism involves providing two different sample views to two encoders and mapping the output of one encoder to the output of another encoder through a predictor. To prevent the encoders from failing by predicting a constant value for any input, these methods adopt various techniques. One common approach is to update the weights of one encoder by using the running average of the weights of the other encoder, thereby preventing the encoders from collapsing.

BYOL is a self-supervised learning method that employs self-distillation to avoid training collapses. It is an improvement over the MOCO method that eliminates the use of negative samples. As shown in Figure 1.6, BYOL shares a similar front-

end network architecture with MOCO, except that BN layers are added to $g_\theta$, and it consists of two networks: the online network and the target network. The key difference lies in that the online network first obtains $Z_\theta$ after the projection layer, and then incorporates a predictor (composed of 1 or 2 fully connected layers) to map to the feature $Z'_\xi$ extracted by the target network, which is essentially a regression task. The loss function is computed using mean squared error (MSE), taking into account that both $Z_\theta$ and $Z'_\xi$ are L2 normalized. Each network receives different views of the same image, obtained through random image transformations such as resizing, cropping, color jittering, and brightness changes. The network structure aims to maximize the similarity between the positive pairs of extracted features, without using negative samples. During the entire training process, the online network is updated using gradient descent, while the target network is updated through the exponential moving average (EMA) of the online network's weights. The asymmetry caused by the slow update of the exponential moving average is crucial for BYOL's success.

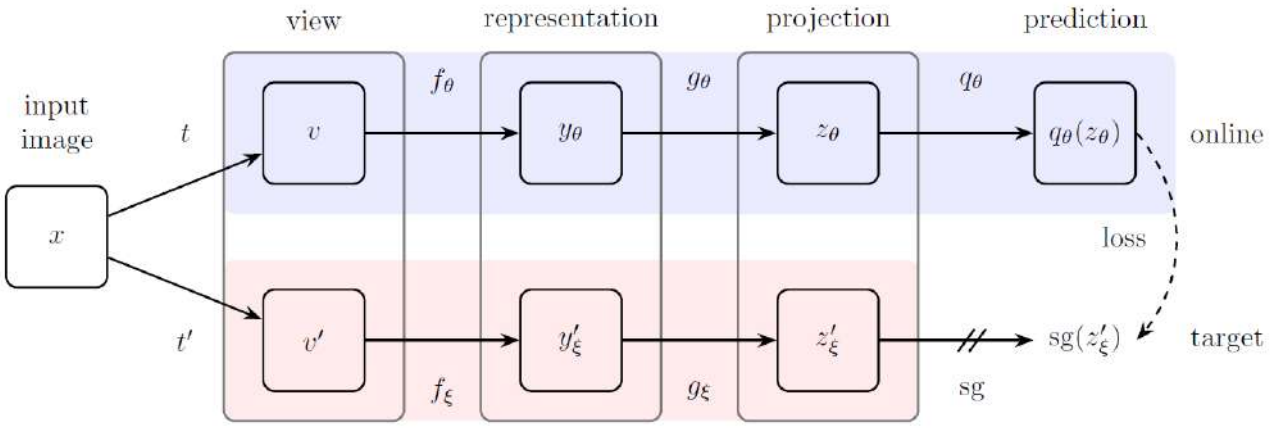



Figure. 1.6 BYOL architecture [38].

SimSIAM [39] proposes a new contrastive learning method based on MoCo [30], SimCLR [27], and BYOL [38], which addresses the issues of model collapse and difficulty in constructing negative samples in contrastive learning with a simple structure. SimSiam demonstrates that EMA is not necessarily required in practice, even though it can lead to a slight improvement in performance. It also investigates the reasons for the absence of model collapse and concludes that the operation of the stop gradient plays a significant role in avoiding model collapse. The model framework is straightforward as shown in the Figure 1.7, the two views of a sample $x$ are processed by an encoder network $f$, which consists of a backbone (e.g., ResNet [41]) and a projection MLP head $h$. The encoder $f$ shares weights between the two views. The $h$ maps the output of one view to the other.

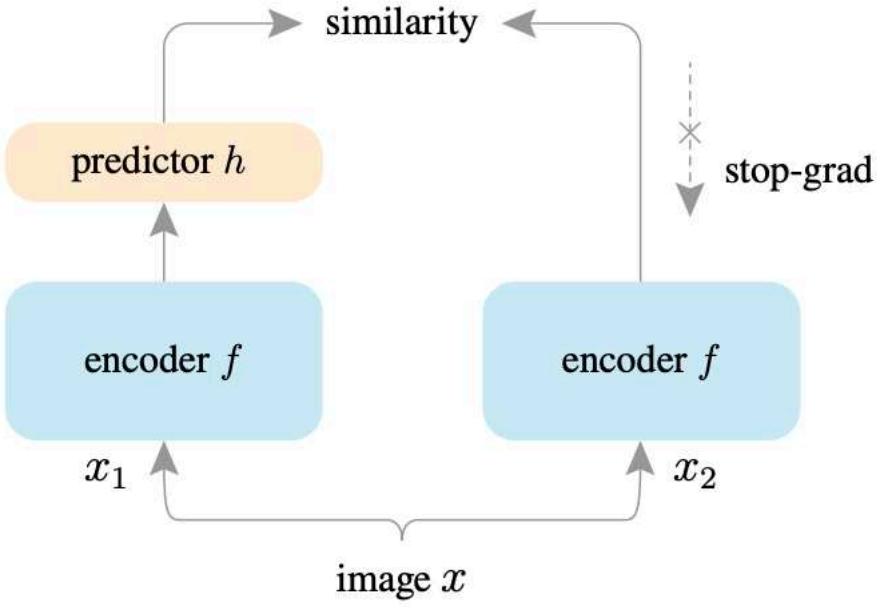



Figure 1.7. SimSIAM architecture [39].

DINO shares similarities with BYOL in that it employs a teacher-student network architecture, where both networks have identical model structures. The teacher network's parameters are updated by computing the moving average of the

student network's weights. The outputs of both networks are normalized using a SoftMax layer, and cross-entropy serves as the loss function for updating the model parameters via backpropagation.

A novel approach called iBOT [42] has been developed by integrating masked image modeling and self-distillation into the DINO framework. The model aims to extract high-level semantic information from image patches using a visual tokenizer, thereby avoiding the learning of redundant details. The iBOT tokenizer possesses two key properties: 1) the ability to represent continuous image content completely, and 2) high-level semantic information, similar to the tokenizer used in natural language processing (NLP). The process of predicting the masked image sequence using the Transformer is modeled as a process of knowledge distillation, where knowledge is obtained from the tokenizer. The target network, such as ViT [6], inputs masked images, while the online tokenizer receives the original images. The main objective is to enable the target network to restore each masked patch token to its corresponding token. The iBOT network framework is shown in Figure 1.8.

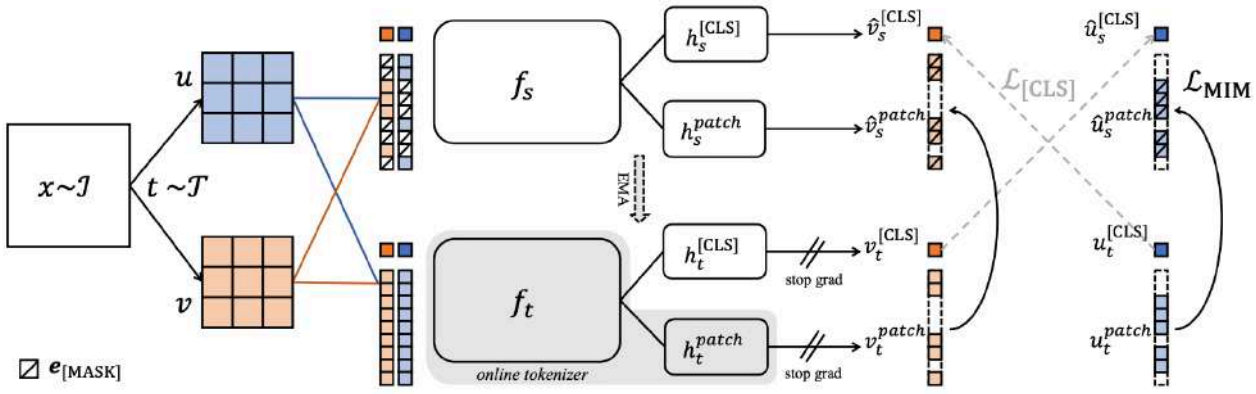



Figure 1.8. overview of the iBOT framework, which utilizes an online tokenizer to perform masked image modeling [42].

DINOv2 [43] builds on the iBOT framework by further refining the training scheme and architecture, resulting in a significant improvement in its performance on both linear and k-NN evaluations. Additionally, DINOv2 plans to utilize an even larger pre-training dataset, consisting of 1.42 billion images. While there are numerous works within the self-distillation family, this paper only provides a brief overview of the classic methods. These self-distillation works serve as inspiration for the development of the self-supervised learning algorithm proposed in this paper, which will be thoroughly discussed in Chapter 2.

## 1.4 Canonical Correlation Analysis Family

Several classic self-supervised learning methods based on the canonical correlation analysis (CCA) framework have been proposed, including VICReg [44], Barlow Twins [45], SWAV [46], and W-MSE [47]. CCA's fundamental principle involves inferring the relationship between two vectors by analyzing the covariance matrix of two variables. Specifically, these methods optimize the correlation between the embeddings of two views using the concept of canonical correlation analysis (CCA), which provides feature-level regularization.

VICReg transforms different views of the same image into embedding vectors using an encoder and constructs a loss function by utilizing three regularization terms to prevent constant or non-informative embedding vectors, thereby improving self-supervised image representation learning. VICReg framework as shown in the Figure.1.9 The first regularization term, Invariance, minimizes the Euclidean distance

between two embedding vectors of the same image. The second regularization term, Variance, uses hinge loss to maintain the standard deviation of each dimension of the embedding vectors (within a batch) above a given threshold, which forces the embedding vectors of different samples in a batch to be different. The third regularization term, Covariance, attracts the covariance (within a batch) of each pair of embedding vector variables to approach zero, which reduces the correlation between embedding vector variables and increases the information content of the embedding vectors. The encoder and extender are trained by minimizing the loss function constructed by the three regularization terms to obtain a model with high-quality embedding vectors.

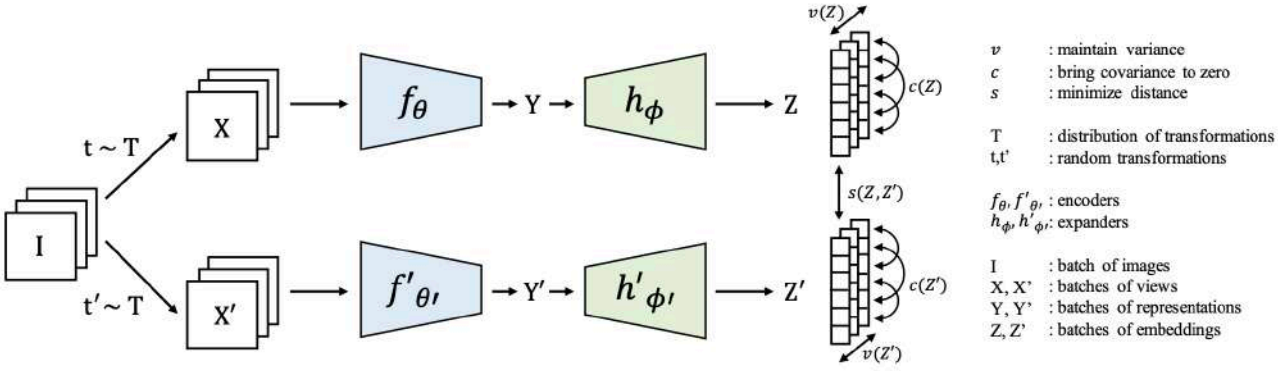



Figure 1.9. overview of the VICReg framework [44].

The Barlow Twins method achieves self-supervised learning by reducing the redundancy of embedding vectors of input samples under noise perturbations. The objective function of this method measures the cross-correlation matrix of outputs from two identical networks and minimizes the distance between the cross-correlation matrix and the identity matrix, thereby minimizing the redundancy of embedding vectors. Through this approach, the Barlow Twins algorithm can learn feature representations that are both robust and interpretable.

The W-MSE method is a self-supervised learning method based on feature whitening operation, which projects feature vectors onto a spherical distribution to prevent all sample features from collapsing into a single point. Compared to traditional contrastive loss methods, this method does not require contrasting positive and negative samples, allowing for multiple positive pairs to be extracted from the same image instance and avoiding the need for large numbers of negative samples. Through feature whitening operation, the W-MSE method is able to scatter the feature representations in a batch, thereby avoiding degenerate solutions. As a potential alternative to contrastive loss methods, the W-MSE method offers a conceptually simple and computationally efficient approach to self-supervised representation learning.

### 1.5 ViT Architecture

The introduction of ViT [6] aims to extend the success of Transformer models in natural language processing (NLP) to the field of computer vision.

The rise of ViT architectures in self-supervised learning can be attributed to the following aspects:

Scalability: ViT models can process images of different sizes, enabling their application to diverse self-supervised tasks like image classification, object detection, semantic segmentation, etc. Their scalability also allows handling large-scale self-supervised datasets for improved performance.

Adaptability: The Transformer design of ViT models can adaptively process input data and leverage self-attention to capture relationships between inputs for better feature representations. This adaptability results in remarkable performance on self-supervised tasks.

Interpretability: ViT models adopt self-attention to extract features, offering improved interpretability into the feature extraction process and further enhancing model performance.

Compared to conventional convolutional neural networks (CNNs), the ViT architecture has the following key differences:

Feature Extraction: CNNs extract image features through convolutional and pooling layers, with the former effectively capturing local features and the latter reducing feature map size and quantity for lower complexity. In contrast, ViT uses Transformer encoders for feature extraction, where self-attention captures global relationships for holistic input understanding.

Input Form: CNNs typically take fixed-size images as input, limiting model ability on varying image sizes. ViT divides input images into fixed-size patches and represents them as vectors, enabling handling of different image sizes.

Computational Efficiency: Large CNNs often require substantial compute and parameters for large images, causing overfitting and low efficiency. ViT incorporates optimizations like removing fully connected layers and lightweight Transformer encoders to improve efficiency and generalization.

Interpretability: CNNs are often black-box models, providing little insight into feature extraction. ViT's self-attention mechanism for feature extraction offers improved interpretability into the model's inner workings.

## 1.6 Conclusion of Chapter 1

In the present chapter, a comprehensive review has been conducted on the evolution of self-supervised learning algorithms utilized in visual feature learning. Although select methodologies may no longer be considered competitive within the contemporary landscape, such early self-supervised techniques have indelibly shaped the foundation upon which current strategies are erected. These algorithms have been broadly classified into four principal categories: contrastive learning, masked image modeling, self-distillation, and canonical correlation analysis.

Contrastive learning approaches ambitiously strive to draw positive sample pairs into closer proximity whilst simultaneously driving negative pairs apart within the embedding space. Noteworthy instances encompass SimCLR, which optimizes the conformity between augmented views, and MoCo, which preserves a negative sample dictionary. Masked image modeling methods endeavor to restore corrupted images, with MAE and BEiT applying BERT's masked language modeling paradigm to images. Self-distillation methodologies such as BYOL and SimSiam distill knowledge between networks to circumvent collapse. Lastly, CCA-based methodologies like VICReg and Barlow Twins leverage correlations existing between views for regularization purposes.

The transformer-based architecture known as the Vision Transformer (ViT) has emerged as a significant player within the realm of self-supervised learning. ViT offers scalability to accommodate fluctuating image sizes, adaptability through self-attention to model relationships, and heightened interpretability when compared to Convolutional Neural Networks (CNNs). ViT-based self-supervised methodologies, including MAE and iBOT, have demonstrated robust performance by capitalizing on advantages such as global modeling.

In conclusion, this chapter has provided an extensive review covering a wide array of ideas, stretching from early techniques to recent ViT-based approaches that have collectively propelled the development of self-supervised visual representation learning. While numerous specific techniques may be deemed obsolete in the current context, the core principles continue to exercise a profound influence over contemporary algorithms. These recent methodologies ingeniously amalgamate these fundamental principles to deliver cutting-edge self-supervised capabilities. The forthcoming chapter will introduce our innovative approach which resourcefully builds upon these trailblazing works.

# CHAPTER 2. ADVANCEMENT IN SELF-SUPERVISED VISUAL FEATURE LEARNING TECHNIQUES THROUGH THE IMPLEMENTATION OF MASKED IMAGE MODELING

This chapter presents two novel self-supervised learning algorithms centered on Masked Image Modeling, predicated on the results of the research conducted.

1. Mixup Feature Algorithm

This work proposes a novel pretext task tailored specifically for self-supervised visual feature learning [67]. This algorithm integrates conventional visual feature maps as the reconstruction targets for Masked Image Modeling. Notably, the feature maps utilized include Sobel edge feature maps [50], Histogram of Oriented Gradients (HOG) feature maps [48], and Local Binary Pattern (LBP) feature maps [49].

2. Denoising Self-Distillation Masked Autoencoder

The work introduces a unique approach that amalgamates the principle of self-distillation with Masked Autoencoders. This synergistic model presents a robust methodology for denoising in the domain of self-supervised visual feature learning.

In the subsequent sections, an articulate exposition of the algorithmic principles inherent to each of the two methods will be provided in a respective manner.

## 2.1 Mixup Features

Inspired by the Masked Autoencoder (MAE) framework, a novel proxy task methodology has been designed for predicting mixed feature maps, termed the Mixup

Feature algorithm. As depicted in Figure 2.1, the algorithm entails a series of specific steps.

In the initial stages of the proposed methodology, the encoder of the Vision Transformer (ViT) backbone network is deployed to extract visual feature representations. During the pre-training phase, the image patches that remain visible post-masking are supplied as input to the encoder. The information supplied to the decoder comprises the encoded details of these visible image patches, coupled with mask tokens. The primary role of the decoder is to reconstruct the Mixup Feature map. The Mixup Feature map, an amalgamation of Sobel edge feature maps, and HOG feature maps, exhibits a novel, mixed structure that can provide more complex and comprehensive visual data representations.

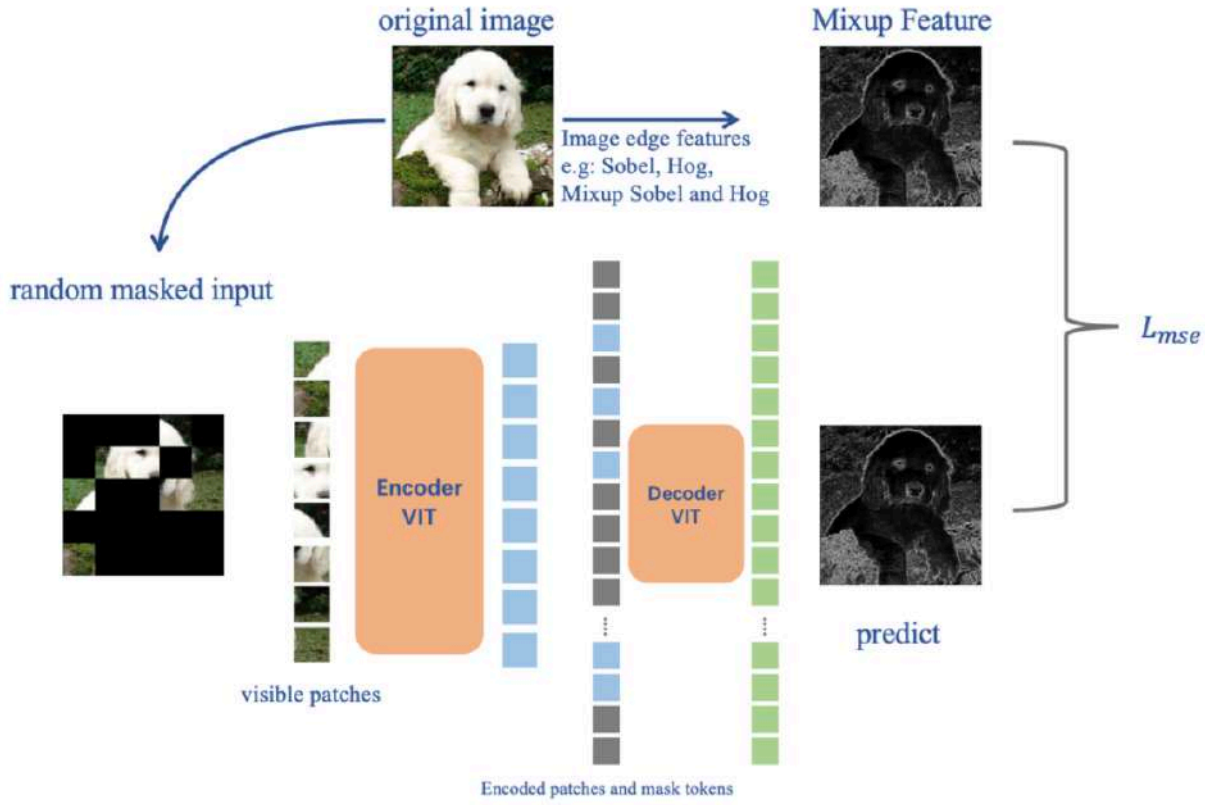



Figure 2.1. Mixup Feature self-supervised learning architecture.

Following the pre-training phase, the encoder is repurposed for subsequent downstream tasks, while the decoder is disregarded.

### 2.1.1 Masking Strategy

In adopting the principles of ViT architecture, images are dissected into non-overlapping segments based on a predetermined patch size, yielding a collection of patches denoted as $X = \{x_1, x_2, \ldots x_n\}$. Subsequently, the ensemble of patches $X$ is subjected to a process of random sampling, implementing a set masking ratio that ensures certain patches are obscured.

The adopted random sampling strategy is simplistic yet effective, operating under the conditions of uniform distribution and without replacement. A significant level of redundancy is eradicated by executing a high masking ratio during the random sampling process. The uniform distribution aids in countering potential central biases, thus providing an even representation across the sampled patches.

In the following subsection, the objective of reconstructing the Mixup Feature will be elaborated.

### 2.1.2 Reconstruction of the Designated Target Mixup Feature

For the purpose of self-supervised learning pretraining, the selection has been made to employ three traditional image feature maps: Local Binary Patterns (LBP), Histogram of Oriented Gradients (HOG), and Sobel edge features. The research encompasses the examination of nine unique feature map pretraining schemes.

In the first instance, the study explores the possibility of using a single traditional feature map as the solitary target signal for the proxy task. This approach gives rise to three different methodologies.

Subsequently, the investigation delves into the strategy of pairing the selected feature maps, with the aim of deriving the sum of feature maps. This method results in an additional set of three schemes.

In the final approach, pairwise combinations are executed to obtain feature maps after the Mixup process, leading to the last three schemes of the study. This exhaustive investigation, encompassing a wide array of feature map pretraining schemes, ensures a comprehensive understanding of the potential capabilities of each approach.

The specific procedure entailed in the manipulation of Mixup features is graphically depicted in equation (2.1). Herein, $\lambda$ serves as a hyperparameter, encompassing a value range that extends from 0 to 1. The two variables $f_1$ and $f_2$ denote the pair of feature maps that are subjected to the Mixup process.

$$F_{mixuped\ feature} = \lambda f_1 + (1 - \lambda) f_2 \tag{2.1}$$

The Local Binary Pattern (LBP) operates as a cardinal method for feature extraction within the domains of computer vision and image processing. The manifold applications of LBP span across distinct areas, encompassing image classification, facial recognition [51], and texture analysis [52], demonstrating its broad-ranging utility. Several key benefits distinguish the LBP descriptor. Notably, it boasts computational simplicity, rotation invariance, and grayscale invariance, factors that contribute to its widespread deployment in the field of computer vision and

image processing. With these capabilities, the LBP descriptor presents an efficient tool for elucidating the texture features intrinsic to various regions of an image. The LBP feature map, a visual representation of the LBP descriptor, is typically generated through the extraction of LBP features from an image, followed by the mapping of each pixel's LBP value to a grayscale equivalent. This process offers a granular view of the texture characteristics inherent to different image regions. Furthermore, the LBP feature map holds significant potential in the realm of image restoration, particularly in the reconstruction of obscured image patches from a limited subset of visible patches. Therefore, LBP can serve as a reconstruction target for pre-text tasks.

Edge detection is the process of identifying and locating sharp discontinuities in an image. Serving as a pivotal aspect of computer vision, it assists in discerning notable attributes of an image such as object boundaries and texture variations. The Sobel edge detector [53] employs a pair of $3x3$ masks to calculate the horizontal and vertical gradients, represented as $G_x$ and $G_y$ respectively. These are then amalgamated to deduce the absolute magnitude and the direction of the gradient at each discrete point, as detailed in Equation (2.2). Utilizing the Sobel operator, the resultant edge feature map serves as the target signal in the prediction of the proxy task. By facilitating the prediction of the masked original image's edges, we equip the model to glean valuable information, thereby diminishing the impact of noise. This process culminates in enhancing the model's comprehension and processing capability of images.

$$|G| = \sqrt{G_x^2 + G_y^2} \tag{2.2}$$

The Histogram of Oriented Gradients (HOG) method generates features through the computation and aggregation of gradient orientation histograms within specified local regions of an image. The modus operandi of HOG entails an initial calculation of image gradients, followed by the image's segmentation into a grid. For each pixel, the gradient direction and magnitude are ascertained, leading to the construction of gradient orientation histograms for all pixels encompassed within each grid cell. These individual histograms are subsequently concatenated to compose the final feature histogram. HOG effectively encapsulates local shape information, while simultaneously mitigating the influence of lighting and color variations on the image, thus reducing the dimensionality of the requisite representation data. The inherent relationships between local pixels are aptly represented, rendering HOG suitable for predicting masked image patches and for self-supervised pretraining. Contrary to the direct usage of Histogram of Oriented Gradients (HOG) features, the adopted approach forecasts HOG feature maps for masked images, wherein specific patch regions remain obscured from sight.

### 2.1.3 The encoder in the Mixup Feature method

The encoder in the Mixup Feature method is ViT [6], following the MAE strategy applied only to unmasked patches. Similar to the standard ViT, our encoder embeds patches by incorporating a linear projection of positional embeddings, subsequently processed through a sequence of transformer blocks.

As depicted in Figure 2.2, the ViT architecture is comprised mainly of two parts within the Transformer encoder, Multi-head Self-Attention (MSA), and Multilayer Perceptron (MLP).

The mathematical formulation of the single-layer Encoder structure in ViT is as follows:

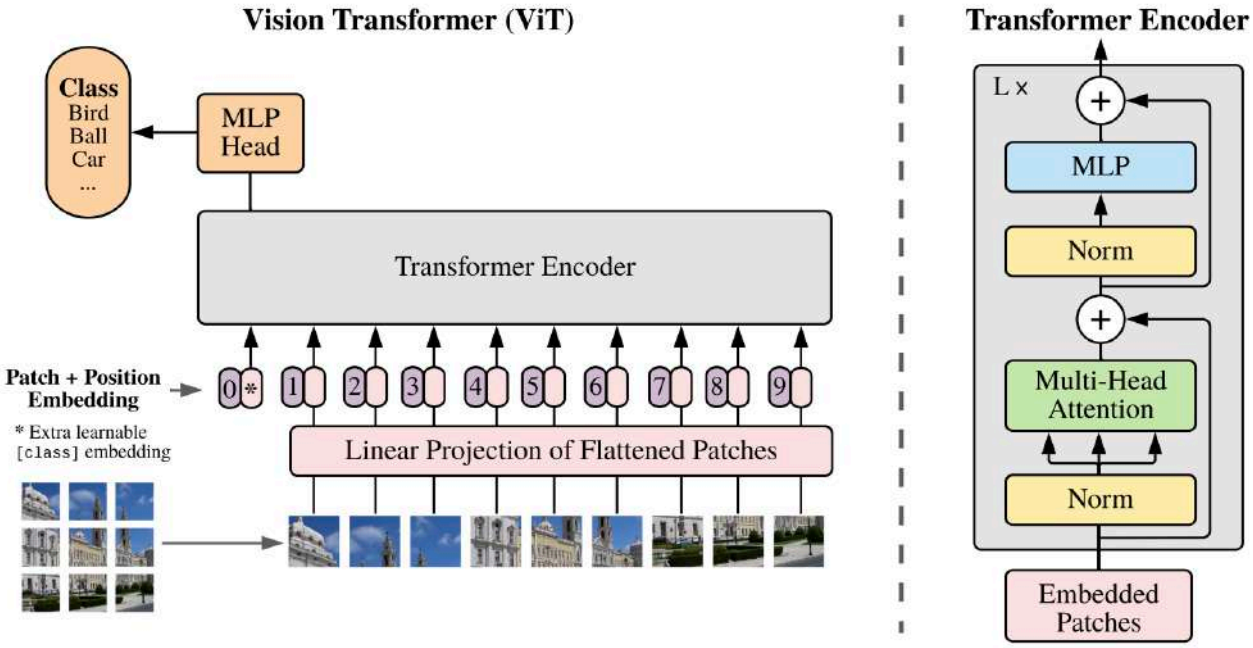



Figure 2.2. ViT architecture [6].

$X_{in}$ denotes the input data, from which we derive three vectors—$Q$ $(query)$, $K$ $(key)$, and $V$ $(value)$—through linear transformation. This is illustrated in equations 2.3, 2.4, and 2.5, with $PE$ indicating positional encoding.

$$Q = W^q(X_{in} + PE) \tag{2.3}$$

$$K = W^k(X_{in} + PE) \tag{2.4}$$

$$V = W^v(X_{in} + PE) \tag{2.5}$$

Subsequently, the similarity between Q and K is calculated, then multiplied by V to yield the ultimate self-attention output. Finally, a linear fully-connected layer is

utilized to map the mid-level feature dimensions to output feature dimensions. As illustrated in equation 2.6, LN stands for LayerNorm. $d_k$ represents the dimension.

$$X_{out} = LN\left(linear\left(softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V\right) + X_{in}\right) \quad (2.6)$$

Upon completion of the Self-Attention process, a feed-forward network (FFN) layer is introduced to manage the $X_{out}$, as depicted in equation 2.7.

$$Y = LN(FFN(X_{out}) + X_{out}) \quad (2.7)$$

### 2.1.4 The Decoder in the Mixup Feature Method

The Mixup Feature method's decoder receives a complete token set, which includes the encoded visible patches and the masked tokens. Please refer to Figure 2.1. Each masked token is a shared learned vector [31], indicating the presence of a missing patch to be predicted. Adding position embeddings to all tokens in the full image patch set, the positional encoding indicates the location of each patch within the image. Without such embeddings, the masked tokens wouldn't possess information about their location in the image.

The decoder is used only during pre-training for the image reconstruction task. Therefore, the decoder's architecture can be designed flexibly, independent of the encoder's design. Experiments were conducted with a decoder that is narrower and shallower than the encoder. Relative to the encoder, this standard decoder processes a lower computational cost for each token. With this asymmetric approach, there is a marked reduction in pre-training duration, thereby enhancing efficiency.

### 2.1.5 Mixup Feature Self-Supervised Pre-Training Process

In the self-supervised pre-training process, the loss function is defined as the L2 loss between the reconstructed mixed feature map and the original mixed feature map obtained. The study also explored two distinct reconstruction types: one that normalizes the mixed feature map and another that does not. The experimental results suggest that using the non-normalized mixed feature map as the reconstruction target yields better performance in downstream tasks. More details will be provided in the following experimental chapter.

**Algorithm 1** Pseudocode for Mixup Feature Pre-training

***Input*** *:* $\boldsymbol{D = \{x^1, x^2, \dots, x^n\}}$

***Output****: Pre-trained Model*

***L****:loss,* ***Optim****: optimizer*

$\boldsymbol{\lambda}$: $\boldsymbol{Mixup\ hyperparameters}$, $\boldsymbol{r}$: $\boldsymbol{masked\ rate}$

*Initialize Model_Masked_Autoencoder*

***For*** *epoch* ***in*** *epochs :*

    *Initialize L*

    ***For*** *d* ***in*** *Dataloader(D):*

        $\boldsymbol{f_1 = get\_feature\_1(d)}$

        $\boldsymbol{f_2 = get\_feature\_2(d)}$

        $\boldsymbol{Mixup\_f = Mixup(f_1, f_2, \lambda)}$

        *Predict_f, mask = Model_Masked_Autoencoder(d,r)*

        *L = Loss( Predict_f , Mixup_f )*

        *L.backward()*

        *Optim.step()*

    ***End For***

***End For***

*Save(Model_Masked_Autoencoder)*

Pseudocode for the Mixup feature self-supervised pre-training stage is illustrated in Algorithm 1.

Where $\boldsymbol{f_1}$ and $\boldsymbol{f_2}$ are two types of feature maps, and the $Mixup()$ function corresponds to the operation in Equation 2.1.

The Loss function is defined as follows:

$$L = min\ \frac{1}{n}\sum_{i=1}^{n} \left\| f_{predict} - f_{mixup} \right\|_2^2 \tag{2.8}$$

Above is the outline of the algorithmic method I've introduced. A comprehensive discussion of the experimental setup and results will be featured in Chapter 3.

## 2.2 Denoising Self-Distillation Masked Autoencoder

Compared to traditional contrastive learning experiments, MAE has effectively pre-trained large models and showcased superior performance when fine-tuned on ImageNet [54]. Like MAE, the majority of MIM objective functions only calculate the MSE loss of the masked regions in the reconstructed image at a pixel level. This proxy task has limitations and might lead to the inability to learn abstract semantic information in images, which is a crucial component of image understanding. To address this issue, we propose a denoising self-distillation Masked Autoencoder model [85] that considers both feature-level regression and pixel-level restoration.

As depicted in Figure 2.3, the approach is designed to remove random Gaussian noise from images and project encoded features onto target features. This method draws inspiration from the MAE concept and also adopts the self-distillation

framework. Instead of gradient-based updates, the teacher network employs the Exponential Moving Average (EMA) for parameter updates [38]. The architecture of the student network includes an encoder, regressor, and decoder. Features are extracted from the input by the encoder, where the input is made up of noise patches inserted at random. Using the representations of the noise patches, the regressor forecasts the representations of the target blocks. In the end, the decoder projects the anticipated noise patcher features onto the original image patches.

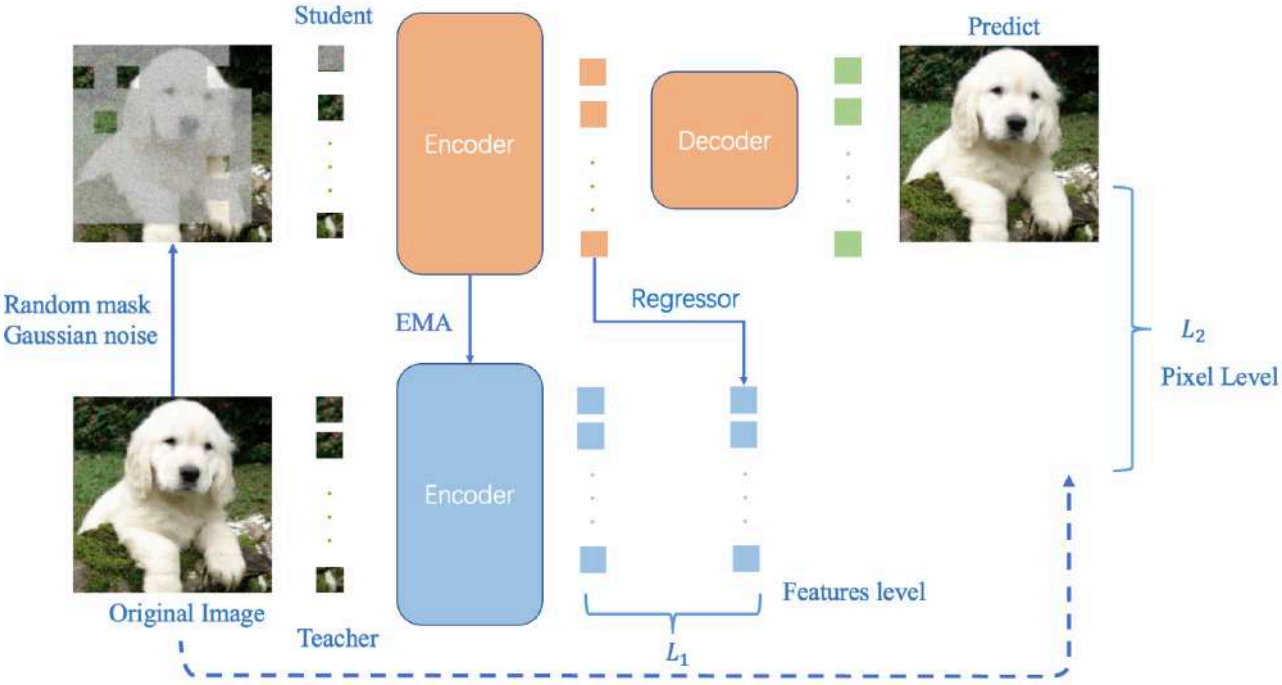



Figure 2.3. Denoising Self-Distillation Masked Autoencoder Framework [85].

### 2.2.1 Random Mask Gaussian Noise

The image is segmented into non-overlapping patches according to a predefined patch size, generating a patch set denoted as $X = \{x_1, x_2, \ldots x_n\}$. Subsequently, Gaussian noise is added to a random sample of the patch set X based on a set mask

noise ratio. The random sampling approach used is straightforward and efficient, carried out with a uniform distribution and without any replacements.

During each iteration of the pre-training phase, a mini-batch $B$ comprising images is sampled. For a given index $i$ within $B$, the $i-th$ image in the mini-batch is denoted as $x_i$. This image, $x_i$, is partitioned into a collection of patches, represented as $x_{io}$, based on a predetermined patch size. Subsequently, Gaussian noise is introduced to a subset of these patches, chosen randomly in accordance with a predefined masking noise ratio $r$, yielding $x_{in}$. The teacher network employs $x_{io}$ as its input, whereas the student network uses $x_{in}$. The pretext task is centered around reconstructing the pristine patches from their noisy counterparts, essentially executing a denoising operation.

### 2.2.2 The Encoder of the Denoising Self-Distillation MAE

The role of the encoder $f_{\theta_s}(x_{in})$ in the student network is to map the noisy block $x_{in}$ to the latent representation $Z_n$, as demonstrated in Equation 2.10. This procedure encompasses all patches in the image $x_{in}$, that is, including patches without added Gaussian noise, $x_{in}$ is derived from $x_{io}$, as demonstrated in Equation 2.9. The encoder utilizes the ViT architecture, beginning with patch embedding and incorporates position embedding to preserve spatial information. Subsequently, the combined embedding is processed through a Transformer encoder, finally generating $Z_n$.

The parameters of $f_{\theta_s}(x_{in})$ are updated by means of gradient-based optimization. Likewise, the encoder $f_{\theta_t}(x_{io})$ in the teacher network and the encoder

$f_{\theta_s}(x_{in})$ in the student network share a similar network architecture to accomplish the mapping from $x_{io}$ to the latent representation $Z_o$, as demonstrated in Equation 2.11. The parameters of $f_{\theta_t}(x_{io})$ are updated by employing the exponential moving average of the parameters of $f_{\theta_s}(x_{in})$.

$$x_{in} = f_{Random\ Mask\ Gaussian\ noise}(x_{io}) \tag{2.9}$$

$$Z_n = f_{\theta_s}(x_{in}) \tag{2.10}$$

$$Z_o = f_{\theta_t}(x_{io}) \tag{2.11}$$

### 2.2.3 The Decoder of the Denoising Self-Distillation MAE

The purpose of the decoder is to map the latent representation $Z_n$ to the denoised patch $Y_n = \phi(Z_n)$. The decoder also adopts the VIT architecture, but it exhibits an asymmetric structure in comparison to the encoder. The decoder only requires a few layers of VIT, significantly reducing the parameters. The input to the decoder only contains the latent representation and position embedding of the noise block. This procedure is mainly for image denoising, constituting pixel-level image restoration.

### 2.2.4 The Regressor of the Denoising Self-Distillation MAE

The main function of the Regressor is to map the features $Z_n'$ output by the student network encoder to the features $Z_o'$ output by the teacher network encoder. $Z_n'$ and $Z_o'$ represent the features of patches excluding those where Gaussian noise has not been added. Regression is performed solely on the noise patches for feature alignment. I implement the Regressor using a VIT structure without a linear head.

### 2.2.5 Pixel-level Restoration

The objective function of Denoising Self-Distillation MAE consists of two parts, pixel-level restoration, and feature-level regression. Under pixel-level restoration, we consider the original image $x_{io}$ as the recovery target, that is, the original image. Through minimizing the L2 loss of the restored image $Y_{in}$ and the original image $x_{io}$, the model gains the capacity to capture local feature information. The pixel-level restoration loss is illustrated in Equation 2.12.

$$\begin{aligned} L_p &= min\frac{1}{n}\sum_{i=1}^{N} D\left(\phi\left(f_{\theta_s}(x_{in})\right), x_{io}\right) \\ &= min\frac{1}{n}\sum_{i}^{N}\sum_{j\in P_i}\left\|Y_{in_j} - x_{io_j}\right\|_2^2 \end{aligned} \tag{2.12}$$

P in Equation 2.12. represents the patches set. $\phi$ represents the decoder.

### 2.2.6 Feature-level Regression

Through feature-level regression, the study further maximizes the mutual information $I(Z_n{}', Z_o{}')$. The process involves predicting the original, undamaged feature representations $Z_o{}'$ in the teacher encoder based on the noisy view representations $Z_n{}'$ in the student encoder, denoted as $F_Z : Z_n{}' \rightarrow Z_o{}'$. The optimizer's goal for the regressor is to maximize the feature cosine similarity between $\phi_r\ (Z_n')$ and the output $Z_o{}'$ of the teacher encoder, as shown in Equation 2.13. The problem of maximizing feature cosine similarity is then transformed into a minimization problem, as indicated in Equation 2.14.

$$L_z = \max log q_{\phi_r}(Z_n', Z_o') \approx max \frac{\sum_{i=1}^{n} \phi_r(f_{\theta_s}(x_{in})\, sg\left[f_{\theta_t}(x_{io})\right] m^i}{\sqrt{\sum_{i=1}^{n} m^i \left(\phi_r\left(f_{\theta_s}(x_{in})\right)\right)^2} \sqrt{\sum_{i=1}^{n} m^i \left(sg\left[f_{\theta_t}(x_{io})\right]\right)^2}} \quad (2.13)$$

$$L_f = \min(1 - L_z) \quad (2.14)$$

In Eq. 2, $sg[\cdot]$ represents the stop gradient operation, $m^i$ represents the index of the noise patches.

The final loss of denoising self-distillation MAE is the weighted sum of the two losses $L_p$ and $L_f$, as shown in Equation 2.15.

$$L = \min\left(\lambda L_p + (1 - \lambda) L_f\right), \lambda \in (0,1) \quad (2.15)$$

The value of $\lambda$ will be discussed in the experimental chapter.

### 2.2.7 Distillation Strategy

The main purpose of self-distillation learning and the EMA (Exponential Moving Average) update strategy is to leverage the knowledge of the powerful teacher model to guide the student model [94].

Denoising self-distillation MAE updates the parameters $\theta_s$ of the student network through backpropagation to minimize the final loss function described in Equation 2.15. On the other hand, the teacher network is updated using an Exponential Moving Average (EMA) in a momentum-based manner. Specifically, the parameters $\theta_s$ of the student encoder are used to update the $\theta_t$ parameters of the teacher encoder, as indicated in Equation 2.16.

$$\theta_t = \eta\theta_t + (1 - \eta)\theta_s \quad (2.16)$$

Here $\eta \in [0,1)$ is the momentum hyperparameter, which is generally set at 0.99. The cosine scheduler is configured to update $\eta$. This scheduler, a commonly utilized method for learning rate reduction, when applied to the momentum hyperparameter within the Exponential Moving Average (EMA) strategy, facilitates progressive adjustment of the momentum value to meet the needs of different training stages. Upon completing a batch's training, the newly updated EMA momentum $\eta$ is utilized to refresh the model's parameters.

The pseudocode for the denoising self-distillation MAE self-supervised pre-training stage is illustrated in Algorithm 2. In the pseudocode, *D* represents the dataset, and $final\_loss$ corresponds to Equation 2.15.

**Algorithm 2** Pseudocode for denoising self-distillation MAE

***Input*** *:* $\boldsymbol{D = \{x_o^1, x_o^2, \dots, x_o^n\}}$

***Output****: Pre-trained Model*

$\boldsymbol{L}$*: Final_Loss,* ***Optim****: optimizer*

$\boldsymbol{\lambda}$: $\boldsymbol{final\ loss\ function\ hyperparameter}$, $\boldsymbol{r}$: $\boldsymbol{masked\ noise\ rate}$

$\boldsymbol{\eta}$: $\boldsymbol{momentum\ hyperparameter}$

*Initialize student_network (·;* $\boldsymbol{\theta_s}$*), with parameters* $\boldsymbol{\theta_s}$

*Initialize teacher_network (·;* $\boldsymbol{\theta_t}$*), with parameters* $\boldsymbol{\theta_t \leftarrow \theta_s}$

***For*** *epoch* ***in*** *epochs :*

*Initialize L*

***For*** *d* ***in*** *Dataloader(D):*

$\boldsymbol{d_n, mask_{index} = f_{Random\ Mask\ Gaussian\ noise}(d, r)}$

$\boldsymbol{Z_n = student\_encoder(d_n)}$

$\boldsymbol{Y_n = student\_decoder(Z_n)}$

$\boldsymbol{Z_o = teacher\_encoder(d)}$

$\boldsymbol{Z_p = student\_regressor(Z_n \cdot mask_{index})}$

*L = Final_Loss(* $\boldsymbol{Y_n, d, Z_p, Z_o, \lambda}$*)*

$\boldsymbol{L.backward()}$ *// Gradient update for student_network parameters* $\boldsymbol{\theta_s}$

*Optim.step()*

***End For***

*// Non-gradient (exponential moving average) update for teacher_network parameters* $\boldsymbol{\theta_t}$

$\boldsymbol{\theta_t \leftarrow \eta\theta_t + (1-\eta)\theta_s}$

***End For***

*Save(student model)*

## 2.3 Conclusion of Chapter 2

Chapter 2 presents two novel self-supervised learning algorithms centered on masked image modeling that advance visual feature learning.

The first algorithm, Mixup Feature, proposes a new pretext task of reconstructing a Mixup of traditional image features like Sobel, HOG, and LBP as the target for a masked autoencoder. This unique mixed feature target provides more complex visual representations for pretraining.

The second algorithm, Denoising Self-Distillation Masked Autoencoder, combines self-distillation with masked autoencoders for robust denoising. It considers both pixel-level image restoration and feature-level regression. Gaussian noise is randomly added to image patches as a pretext task for the student network to denoise. The teacher network guides the student through exponential moving average parameter updates. An asymmetric decoder further enhances efficiency. The loss function balances pixel reconstruction loss and feature alignment loss.

In summary, this chapter presents two novel masked image modeling algorithms for self-supervised visual feature learning. Mixup Feature explores mixed traditional

feature targets to provide richer representations. Denoising Self-Distillation Masked Autoencoder combines self-distillation and denoising to learn robust features efficiently. Both methods demonstrate unique innovations in advancing self-supervised techniques through masked image modeling objectives. The experimental setup, results, and analyses of both algorithms are further discussed in depth in the following chapter.

# CHAPTER 3. EXPERIMENTS AND RESULTS ON MIXUP FEATURE AND DENOISING SELF-DISTILLATION MASKED AUTOENCODER ALGORITHMS

In the realm of self-supervised visual feature learning, rigorous empirical validation is pivotal to ascertain the efficacy and robustness of proposed methodologies. In this chapter, we delve into a comprehensive evaluation of two novel methods, namely the Mixup Feature technique and the Denoising Self-Distillation Masked Autoencoder.

This chapter begins with a comprehensive explanation of the chosen pre-training datasets, detailing their complexities and significance. Subsequently, the experimental configurations for the two proposed methodologies are delineated, offering clarity on the operative mechanisms and settings involved.

The subsequent segment concentrates on model evaluation. It enumerates the primary metrics and their respective outcomes and engages in a visual representation of the pre-training experimental results, which enhances the interpretability and comprehensibility of the findings.

In pursuit of a holistic understanding of the algorithmic dynamics, ablation studies are conducted. Such an approach enables me to discern the contributive significance of individual components, facilitating a more granular assessment of their collective impact on the overarching performance of the self-supervised pre-training models.

## 3.1 Self-Supervised Pre-Training Dataset

The prevailing approach entails pre-training SSL models on meticulously curated datasets such as ImageNet [54] and alternative datasets like PASS [55]. Such datasets are typically class-balanced and harbor object-centric images where the object predominantly resides at the photograph's center. However, the sheer magnitude of these datasets necessitates substantial computational resources. Considering computational constraints, a decision was made to pre-train on more compact datasets, with subsequent evaluations conducted on the self-supervised pre-trained models. Consequently, CIFAR-10 [56], CIFAR-100 [56], and STL-10 [57] were selected as the foundational datasets for these self-supervised pre-training endeavors.

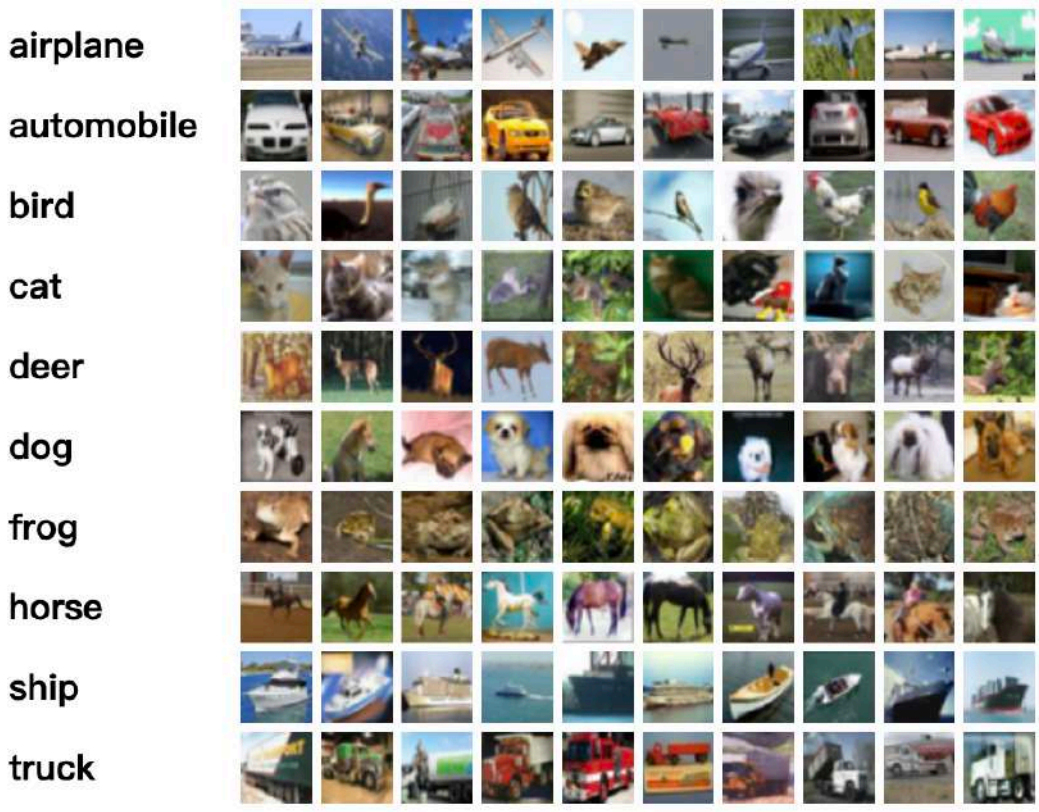



Figure 3.1. CIFAR-10 Dataset samples [56].

**CIFAR-10 Dataset**

The CIFAR-10 dataset is a renowned benchmark in the machine learning and computer vision communities. It consists of 60,000 32x32 color images spanning 10 distinct classes, with each class containing 6,000 images. The dataset is partitioned into 50,000 training images and 10,000 testing images. The 10 classes encompassing this dataset are airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. Some samples of random visualization are shown in Figure 3.1. Given its relatively moderate size and diverse content, CIFAR-10 has often served as a go-to dataset for developing and validating novel algorithms, particularly in the domain of image classification.

**CIFAR-100 Dataset**

An extension of the CIFAR-10 dataset, CIFAR-100 provides a more granular and challenging benchmark. Like its predecessor, it comprises 60,000 32x32 color images, but these are spread across 100 classes, with each class accounting for 600 images. To add an additional layer of complexity, these 100 classes are grouped into 20 super classes. Each superclass encompasses 5 classes, introducing a hierarchical structure to the dataset. Examples of super classes include aquatic mammals, fruits and vegetables, and vehicles. The dataset's split remains consistent with CIFAR-10, allocating 50,000 images for training and 10,000 for testing. Given its increased class variability, CIFAR-100 is employed when a more intricate classification task is desired.

**STL-10 Dataset**

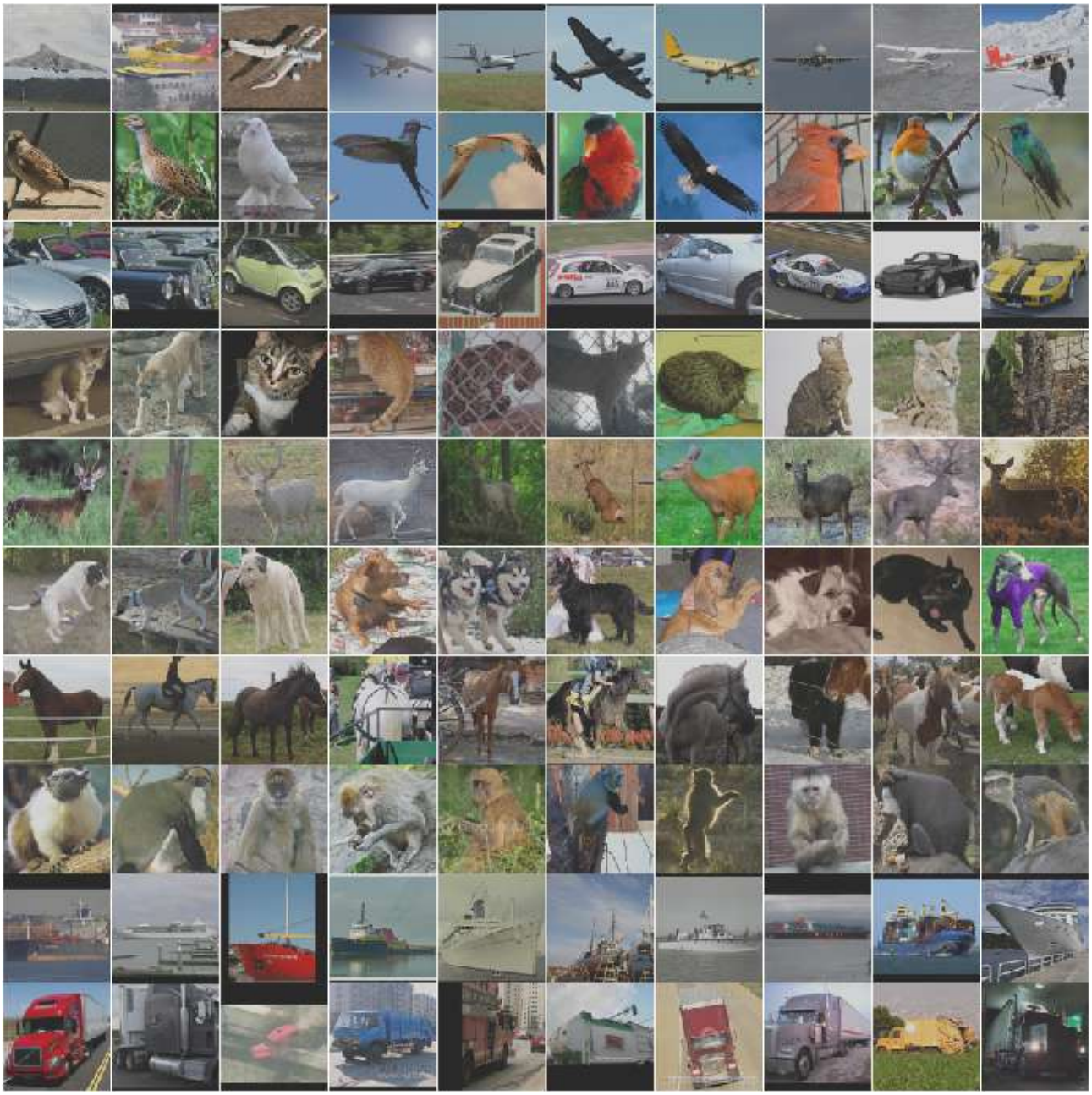


Figure 3.2. STL-10 Dataset samples [57].

The STL-10 dataset is designed specifically for developing and benchmarking unsupervised feature learning and self-taught learning algorithms. A distinct aspect of STL-10, compared to datasets like CIFAR-10 and CIFAR-100, is that it offers a more challenging set of images derived from a larger dataset, ImageNet. The dataset comprises 96x96 color images across 10 different classes, specifically: airplane, bird,

car, cat, deer, dog, horse, monkey, ship, and truck. Visual exemplars from each class can be referenced in Figure 3.2. However, the data distribution is different:

- Unlabeled Data, A significant portion of STL-10 is the 100,000 unlabeled images, which are intended for unsupervised learning or pre-training.
- Training Data, the labeled part of the dataset contains 5,000 images (500 images per class).
- Testing Data, the test set consists of 8,000 images (800 per class).

Given the high-resolution nature of its images relative to CIFAR datasets and its emphasis on unsupervised learning, STL-10 serves as a valuable intermediary benchmark, positioned between small datasets like CIFAR and large-scale datasets like ImageNet. It tests the scalability of algorithms to more realistic image sizes while still being computationally manageable.

The dataset has been a cornerstone for researchers focusing on unsupervised or self-supervised methodologies in computer vision.

## 3.2 Evaluation for SSL Models

Evaluating self-supervised pre-training primarily involves assessing the pre-trained models on image classification tasks, given that image classification remains central to computer vision. Presently, the mainstream methodologies primarily adhere to three main protocols: k-Nearest Neighbors (KNN), linear, and full fine-tuning evaluations. These three protocols pertain to offline evaluation techniques,

executed post the self-supervised training regimen, and stand in contrast to the online evaluations carried out amidst the training phase.

**KNN**

KNN [58] is a well-known unsupervised machine learning method widely utilized. In high-dimensional spaces, measuring distances between samples might be affected by the "curse of dimensionality," limiting the effectiveness of KNN. Therefore, directly evaluating the performance of its representations on these downstream tasks might be more meaningful than using KNN. In the study of self-supervised representation learning, typical evaluation methodologies include using linear classifiers, full fine-tuning, and performance analysis on downstream tasks to ascertain the quality of the acquired representations. These methods provide a direct assessment of the representations' applicability.

**Linear Probe Evaluation**

In the field of self-supervised learning, linear evaluation, or linear probe evaluation [59], initially proposed training a linear classifier on pre-trained feature representations. In this evaluation approach, features are extracted using a pretrained self-supervised model, and then a linear classifier is trained on these features to accomplish a specific supervised task. The fundamental idea behind this method is that if a model's learned representation is robust, even a simple linear classifier should achieve good performance on these representations. The protocol is a staple in numerous studies, and its popularity stems from several fundamental reasons:

- Simplicity: Linear classifiers have a fixed and straightforward structure, making performance comparisons across different representations more direct and fair.

- Limited Model Complexity: The simplicity of linear classifiers implies they cannot "compensate" for shortcomings in the original representations. Thus, if a linear classifier performs well on a certain representation, it likely signifies that the representation has captured essential data features.

- Efficiency: Its computational cost isn't high, making evaluations on large-scale datasets more feasible.

Specifically, the experiment process necessitates freezing the backbone network and appending a linear layer at the terminal stage, with training extending for approximately 100 epochs to reach completion. The advantages of employing lightweight parameters for linear evaluation, alongside the concurrent evaluation of multiple linear heads, facilitate optimizing numerous hyperparameters within the model [59].

**Full Fine-tuning**

Full fine-tuning was introduced as an evaluation metric in the MAE paper. Since linear probing's performance isn't directly correlated with fine-tuning and transfer learning, a small MLP head cannot assess non-linear features. Most subsequent works [32,42,60] adopted this type of evaluation. In terms of fine-tuning, the performance of contrastive methods falls short compared to masked image modeling, mainly because they aren't very "optimization-friendly" [61], this also piqued

researchers' interest in MIM. The drawback of this evaluation method is its high computational cost, necessitating the retraining of the entire network.

**Visual Evaluation**

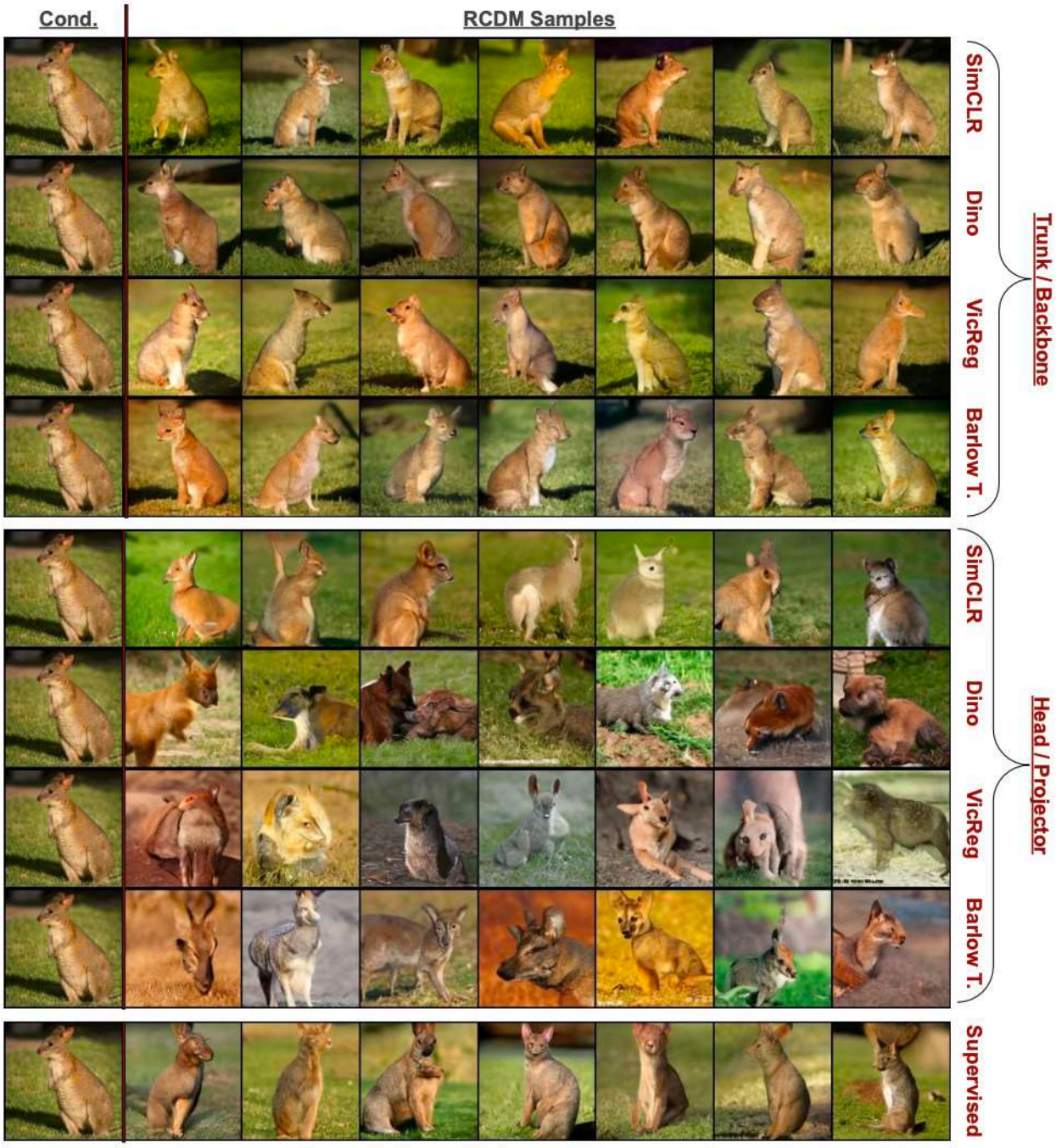



Figure 3.3. RCDM visualization, Figure from [62].

Although self-supervised learning methods have achieved significant success in downstream tasks, our understanding of these algorithms and their learned

representations remains limited. Visualizing the representations or reconstruction targets by self-supervised learning methods can enhance our understanding. MAE visualizes the reconstruction targets of the decoder, thereby further assessing what information is contained in and excluded from the representations. The Representation Conditional Diffusion Model (RCDM) is a tool [62] designed to visualize the representations learned by self-supervised learning methods. As illustrated in Figure 3.3, by comparing the representations learned at the projector level to those learned at the backbone level. Through visualization, it becomes evident that projector representations retain only global information, lacking contextual preservation, as opposed to backbone representations. This observation suggests that visual evaluation of the self-supervised learning model contributes to an in-depth understanding of self-supervised learning mechanisms.

## 3.3 Experimental Analysis of the Mixup Feature Method

To ascertain the efficacy of the novel method introduced, comparisons were drawn with antecedent generative-based and contrastive learning-based self-supervised learning algorithms. Extensive pre-training experiments were conducted across three distinct datasets—CIFAR-10, CIFAR-100, and STL-10—utilizing encoders of varying dimensions (ViT-Tiny, ViT-Small, and ViT-Base) each targeting specific reconstruction objectives. Although each model—ViT-Tiny, ViT-Small, and ViT-Base—comprises 12 transformer blocks, they are differentiated by their hidden

sizes, with ViT-Tiny possessing a hidden size of 192, ViT-Small 384, and ViT-Base 768.

This study conducts benchmark evaluations employing linear probing and full fine-tuning classification techniques. During the pre-training phase, the Mixup Feature model is developed using a self-supervised approach, independent of labels within the selected dataset. For the evaluation, the decoder utilized in the pre-training phase is omitted, maintaining solely the pre-trained encoder, to which a classification head is appended to carry out classification tasks. During linear evaluation, the parameters ingrained by the encoder throughout the pre-training phase remain static, with only the linear classification head being trained and assessed against the test subset of the dataset. In the full fine-tuning evaluation, both the pre-trained encoder and the fully connected layers are subjected to training. The study further includes ablation experiments to elucidate the effects of image augmentation, as well as the impact of masking rates and mixup hyperparameters on the reconstruction tasks of the feature map.

### 3.3.1 Implementation Details

The experiments conducted on the CIFAR-10 and CIFAR-100 datasets incorporated the VIT-Tiny and VIT-Small architectures to curb the potential for overfitting. For trials on the STL-10 dataset, the VIT-Small and VIT-Base ViT architectures were selected. The CIFAR-10 and CIFAR-100 datasets are comprised of 32 × 32-pixel images, for which a block size of 2 × 2 pixels was chosen. In

contrast, the STL-10 dataset consists of 96 × 96-pixel images, prompting the selection of a 6 × 6-pixel block size for these experiments.

Table 3.1 Pre-training settings

| **Config** | **CIFAR-10 or 100** | **STL-10** |
|---|---|---|
| architecture | ViT-T\ViT-S | ViT-S\ViT-B |
| batch size | 1024\1024 | 1024\512 |
| patch size | 2 | 6 |
| optimizer | AdamW [64] | |
| optimizer momentum | $\beta_1, \beta_2 = 0.9, 0.95$ | |
| weight decay | 0.05 | |
| learning rate schedule | Cosine decay [63] | |
| base learning rate | 1.5e-4 | |
| warmup epochs [65] | 50 | |
| Epoch | 1600\1600 | 1500\1200 |

Table 3.2 Fine-tuning setting

| **Config** | **CIFAR-10 or 100** | **STL-10** |
|---|---|---|
| architecture | ViT-T\ViT-S | ViT-S\ViT-B |
| batch size | 1024\1024 | 1024\512 |
| drop path | 0.1 | 0.1 |
| optimizer | AdamW [64] | |
| optimizer momentum | $\beta_1, \beta_2 = 0.9, 0.999$ | |
| weight decay | 0.75 | |
| learning rate schedule | Cosine decay [63] | |
| base learning rate | 2e-3 | 1e-3 |
| warmup epochs [65] | 10 | |
| epoch | 100 | |

The default configurations for pre-training can be found in Table 3.1. Default settings for fine-tuning are detailed in Table 3.2. The tables presented display various

hyperparameters in the first column. The second column showcases parameter configurations for CIFAR-10 and CIFAR-100, while the third column is dedicated to configurations for the STL-10 dataset.

**Algorithm 3: Learning Rate Scheduling with Warmup and Cosine Decay**

***Input***:*initial_learning_rate,max_learning_rate,min_learning_rate,warmup_epochs, total_epochs*

***Initialize optimizer***

***for*** *current_epoch* ***to*** *total_epochs* ***do*** *:*

*// Warmup phase*

***if*** *current_epoch* ≤ *warmup_epochs* ***then***:

$$\boldsymbol{lr} \leftarrow \boldsymbol{initial\_learning\_rate} + (\boldsymbol{max\_learning\_rate} - \boldsymbol{initial\_learning\_rate}) * (\frac{\boldsymbol{current\_epoch}}{\boldsymbol{warmup\_epochs}})$$

*// Cosine decay phase*

***else :***

$$\boldsymbol{alpha} \leftarrow \frac{\boldsymbol{current\_epoch - warmup\_epochs}}{\boldsymbol{total\_epochs - warmup\_epochs}}$$

$$\boldsymbol{lr} \leftarrow \boldsymbol{min\_learning\_rate} + \mathbf{0.5} * (\boldsymbol{max\_learning\_rate} - \boldsymbol{min\_learning\_rate}) * (\mathbf{1} + \boldsymbol{cos}(\boldsymbol{pi} * \boldsymbol{alpha}))$$

***end if***

*// Update the learning rate of the optimizer*

*set_learning_rate(optimizer, lr)*

*train_one_epoch()*

***End For***

*Save(student model)*

The optimizer employed is AdamW [64], a variant derived from the Adam optimizer, with enhancements in the approach to weight decay. Compared to traditional Adam, it offers superior regularization, aiding the model in better generalizing to new data and achieving faster convergence.

The adjustment of the learning rate adopts a strategy combining Cosine decay [63] and warmup [66]. Specifically, warmup is executed at the beginning of training,

and after the completion of warmup, cosine annealing is used to gradually reduce the learning rate. During the Warmup phase, if the current epoch is less than $warmup\_epochs$, the learning rate linearly increases from $initial\_learning\_rate$ to $max\_learning\_rate$. During the Cosine decay phase, if the current epoch is greater than or equal to $warmup\_epochs$, the learning rate is adjusted using the cosine annealing strategy. The pseudocode is shown in Algorithm 3.

### 3.3.2 Mixup Feature Scheme

The model is designed to reconstruct the Mixup feature map for each masked patch, with the decoder generating the reconstructed Mixup feature map. The mean squared error between the predicted and the target feature map constitutes the loss function. This study computes the loss across the entire predicted Mixup feature map, in contrast to methods like MAE that calculate loss solely within the masked areas. Additionally, the normalization of the reconstructed feature map was omitted in these experiments. It was noted that such normalization impacts the experimental results, as normalizing the reconstruction loss could result in significantly reduced values, which may lead to gradient vanishing issues during the optimization process. This approach diverges from the loss computation employed in MAE, where image pixel reconstruction is the focus.

To ascertain the efficacy of the proposed method, comparative analyses were conducted among experimental outcomes derived from various reconstruction targets, encompassing individual feature maps and their pairwise combinations. These evaluations of self-supervised pre-training methods, predicated on distinct

reconstruction targets, were executed across three datasets. Table 3.3 shows the results of full fine-tuning experiments, and Table 3.4 represents the results of linear probing experiments. The "+" symbol indicates the direct addition of two feature maps, while "Mixup" refers to the operation specified in equation (2.1) performed on the two feature maps inside the brackets. Our findings suggest that the feature maps generated by Mixup enhance the performance of experimental evaluations on the test set when compared to prior MAE methodologies.

Table 3.3 Full Fine-tuning (Top1-accuracy)

| reconstruction target | mask rate | CIFAR-10 | | CIFAR-100 | | STL-10 | |
|---|---|---|---|---|---|---|---|
| | | ViT-tiny | ViT-small | ViT-tiny | ViT-small | ViT-small | ViT-base |
| scratch baseline | O | 73.89 | 79.96 | 51.25 | 55.57 | 78.76 | 82.36 |
| Pixels (MAE) [33] | 0.75 | 89.87 | 91.79 | 66.72 | 67.83 | 86.20 | 87.69 |
| LBP feature map | 0.5 | 88.27 | 90.14 | 65.96 | 66.59 | 85.62 | 86.90 |
| HOG feature map [68] | 0.4 | 90.12 | 91.75 | 66.83 | **67.86** | 86.23 | 87.85 |
| Sobel edge map | 0.6 | 90.03 | 91.67 | 66.54 | 67.28 | 86.12 | 87.79 |
| LBP + HOG | 0.5 | 88.93 | 90.78 | 66.25 | 66.93 | 85.92 | 87.43 |
| LBP + Sobel | 0.5 | 88.90 | 90.93 | 66.31 | 66.99 | 86.11 | 87.31 |
| HOG + Sobel | 0.5 | 89.11 | 90.84 | 66.67 | 67.45 | 86.18 | 87.57 |
| Mixup (LBP, HOG) | 0.5 | 90.17 | 91.91 | 66.83 | 67.60 | 86.42 | 87.77 |
| Mixup (LBP,Sobel) | 0.5 | 90.08 | 91.86 | 66.85 | 67.75 | 86.29 | 87.89 |
| Mixup (HOG,Sobel) | 0.5 | **90.56** | **92.09** | **67.12** | 67.77 | **86.55** | **88.06** |

Table 3.4 linear probing (Top1-accuracy)

| reconstruction target | mask rate | CIFAR-10 | | CIFAR-100 | | STL-10 | |
|---|---|---|---|---|---|---|---|
| | | ViT-tiny | ViT-small | ViT-tiny | ViT-small | ViT-small | ViT-base |
| Pixels (MAE) [33] | 0.75 | 75.45 | 77.52 | 50.12 | 54.10 | 79.39 | 83.72 |
| LBP feature map | 0.5 | 74.30 | 75.74 | 45.96 | 52.85 | 78.17 | 82.65 |
| HOG feature map [68] | 0.45 | 75.50 | **77.89** | 50.30 | 54.15 | 79.92 | 83.98 |
| Sobel edge map | 0.6 | 75.41 | 77.68 | 49.74 | 53.96 | 79.46 | 83.81 |
| LBP + HOG | 0.5 | 74.82 | 76.47 | 47.23 | 53.34 | 78.73 | 82.78 |
| LBP + Sobel | 0.5 | 74.78 | 77.31 | 48.44 | 52.93 | 78.20 | 82.03 |
| HOG + Sobel | 0.5 | 75.11 | 77.26 | 48.02 | 53.67 | 78.13 | 82.89 |
| Mixup (LBP, HOG) | 0.5 | 75.47 | 77.81 | 49.61 | 53.94 | 79.86 | 83.89 |
| Mixup (LBP,Sobel) | 0.5 | 75.39 | 77.75 | 49.98 | 54.08 | 80.19 | 84.05 |
| Mixup (HOG,Sobel) | 0.5 | **75.96** | **77.89** | **50.39** | **54.55** | **80.43** | **84.12** |

The validation of Mixup's effectiveness through various reconstruction targets reveals that the mere aggregation of feature maps does not enhance the feature-extraction capabilities of pre-trained models. Indeed, the outcomes of simple additive reconstructions fall short compared to those of single-feature reconstructions.

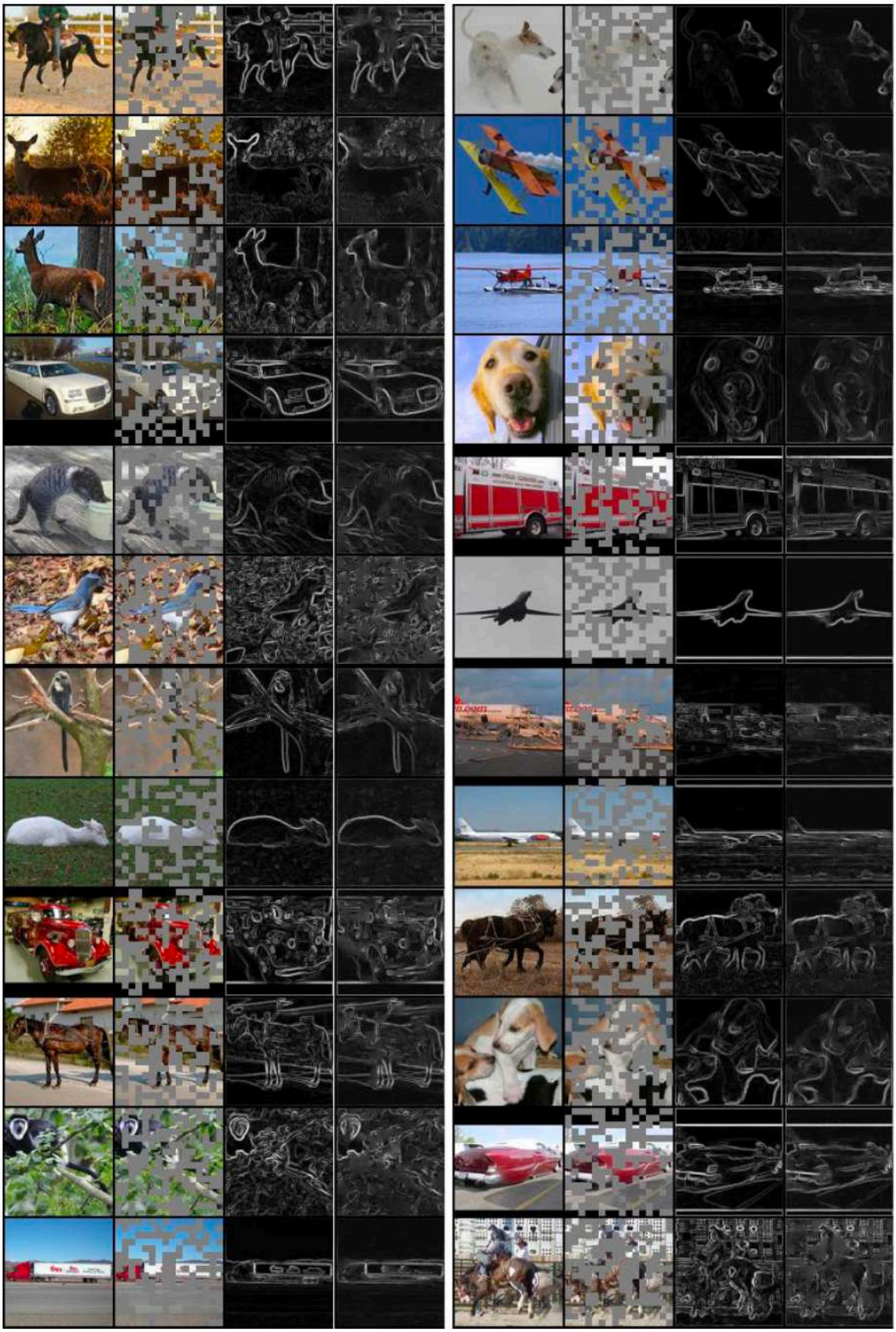


Figure 3.4. Randomly visualizing mixed feature predictions. All images are from the validation set of STL-10.

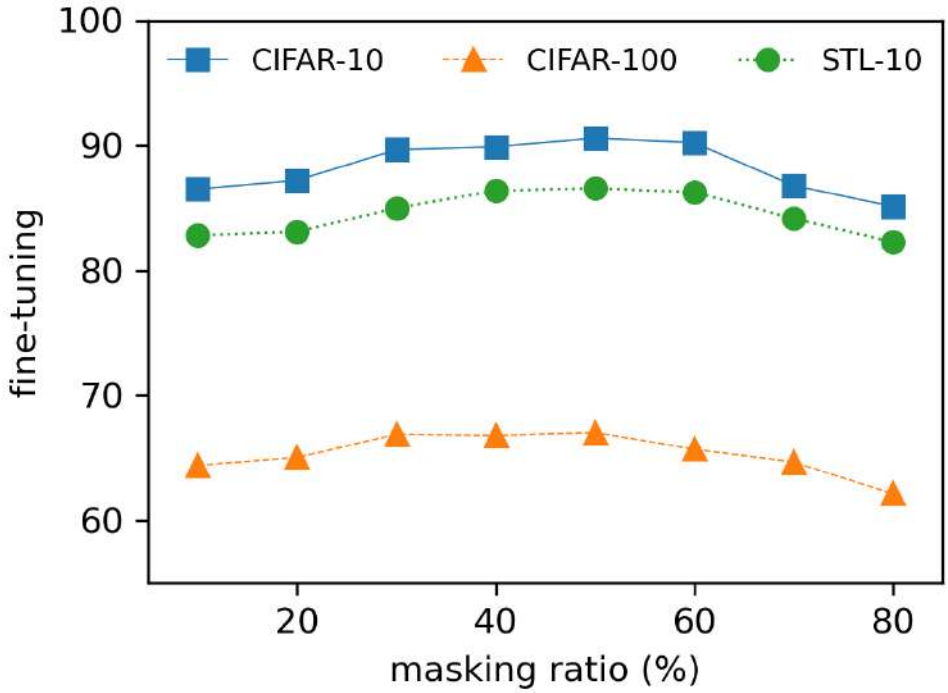
Nevertheless, Mixup of feature maps, particularly those involving HOG and Sobel methods, yielded superior results. Regarding the STL-10 dataset, feature maps reconstructed using the proposed strategy were visualized. Figure 3.4 illustrates these results, sequentially presenting the original image, the masked image, the target mixed feature map, and the reconstructed mixed feature map.

In the experiments, the models pre-trained using the Mixup strategy demonstrated significant performance improvements across multiple datasets.

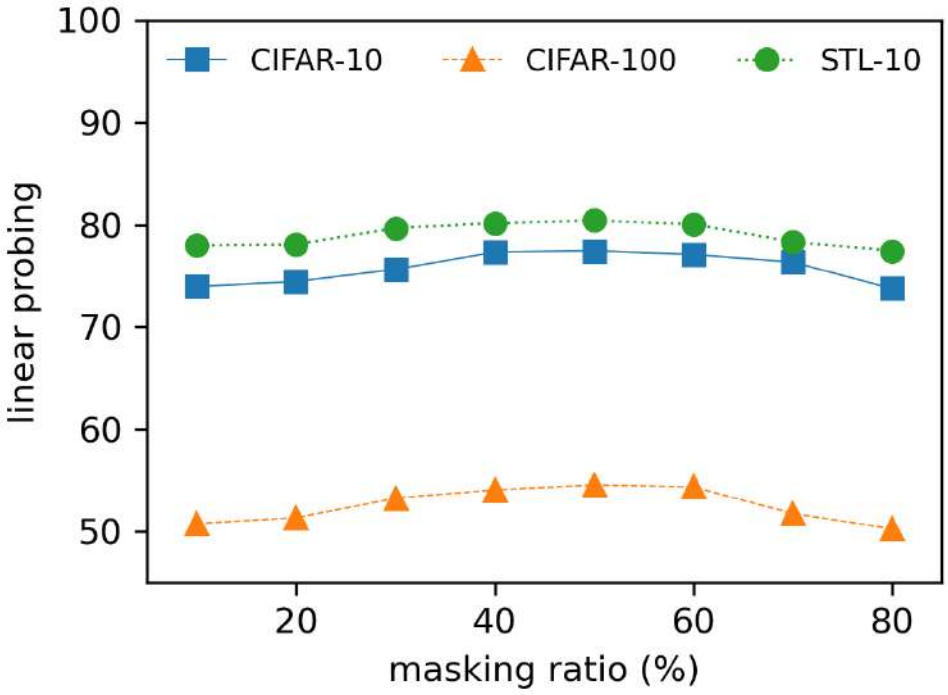
Specifically, when tested on the CIFAR-10 and CIFAR-100 datasets, our Mixup Feature pre-trained models, in their optimal configuration, realized an accuracy boost of nearly 10% compared to the baseline trained from the ground up. On the STL-10 dataset, with the ViT-small as the backbone network, we noted a performance uptick of 9% relative to the baseline. And when employing the more complex ViT-base as the backbone network, there was still an impressive 5% growth in performance. These results not only validate the effectiveness of the Mixup Feature pre-training strategy but also emphasize the potential performance differences of different backbone network structures when facing self-supervised learning tasks, offering valuable insights for subsequent research.

### 3.3.3 Masking Ratio

The study examines the influence of varying percentages of patch masking on the feature extraction capabilities of pre-trained models. In Figure 3.5, the masking rate vs accuracy curve was obtained from (a) Full fine-tuning experiments and (b) linear probing experiments




(a) Full fine-tuning experiments



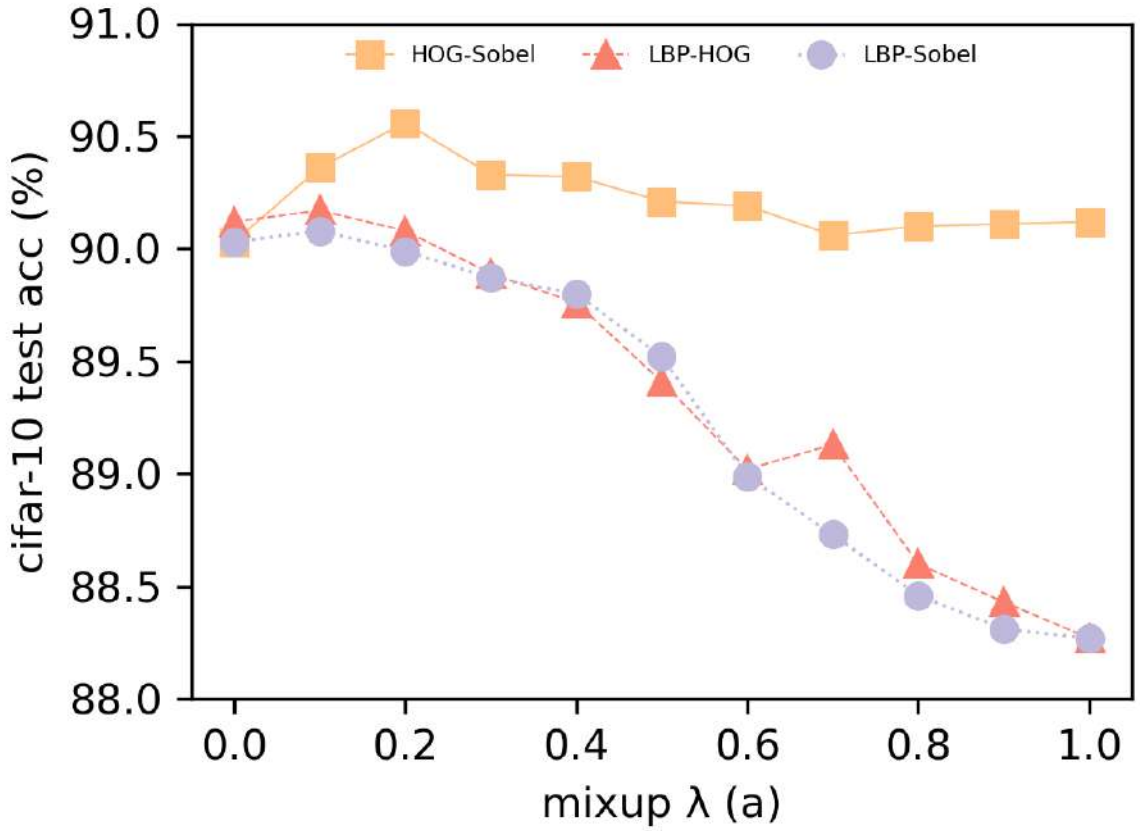

(b) linear probing experiments

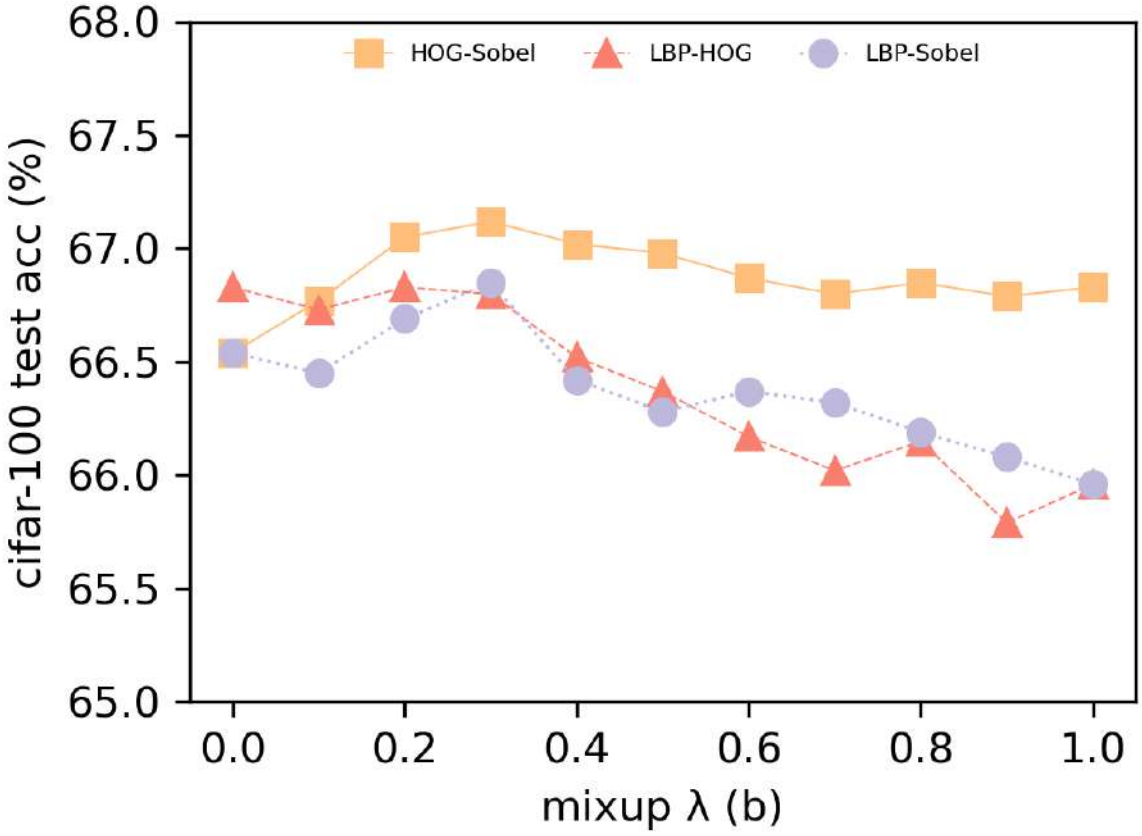
Figure 3.5 The masking rate vs accuracy curve

The y-axis represents the corresponding validation accuracy (%), and the x-axis represents the corresponding masking rate. The outcomes of two experiments were presented, indicating that models exhibit improved accuracy with a masking rate between 40% and 60%. This contrasts with the higher masking rates employed in the MAE pixel reconstruction tasks, suggesting that the effectiveness of the mixup feature method hinges upon the availability of a sufficient number of visible patches. The pair of experiments shown in Figure 2 were performed on three different datasets, with HOG and Sobel as the feature maps utilized for Mixup. In Table 3.3, this combination achieved the best experimental results. The experimental results show that an excessive masking rate makes the reconstruction of Mixup features overly complicated, while a too low masking rate oversimplifies the reconstruction. When the mask rate is set to zero, the model only learns how to extract HOG feature maps and Sobel edge information, limiting its ability to learn sufficiently abstract semantic information from images.

### 3.3.4 Mixup Factor $\lambda$

The mixing factor between two feature maps is pivotal for model performance. An investigation into its impact involved conducting experiments on the hyperparameter $\lambda$, which ranges from 0 to 1 as per equation (2.1). This investigation encompassed pairwise combinations of three distinct feature types across three datasets. Figure 3.6 illustrates the influence of the hyperparameter blending factor $\lambda$ on Full Fine-tuning performance.


HOG-Sobel
LBP-HOG
LBP-Sobel
cifar-10 test acc (%)
91.0
90.5
90.0
89.5
89.0
88.5
88.0
0.0
0.2
0.4
0.6
0.8
1.0
mixup λ (a)


HOG-Sobel
LBP-HOG
LBP-Sobel
cifar-100 test acc (%)
68.0
67.5
67.0
66.5
66.0
65.5
65.0
0.0
0.2
0.4
0.6
0.8
1.0
mixup λ (b)

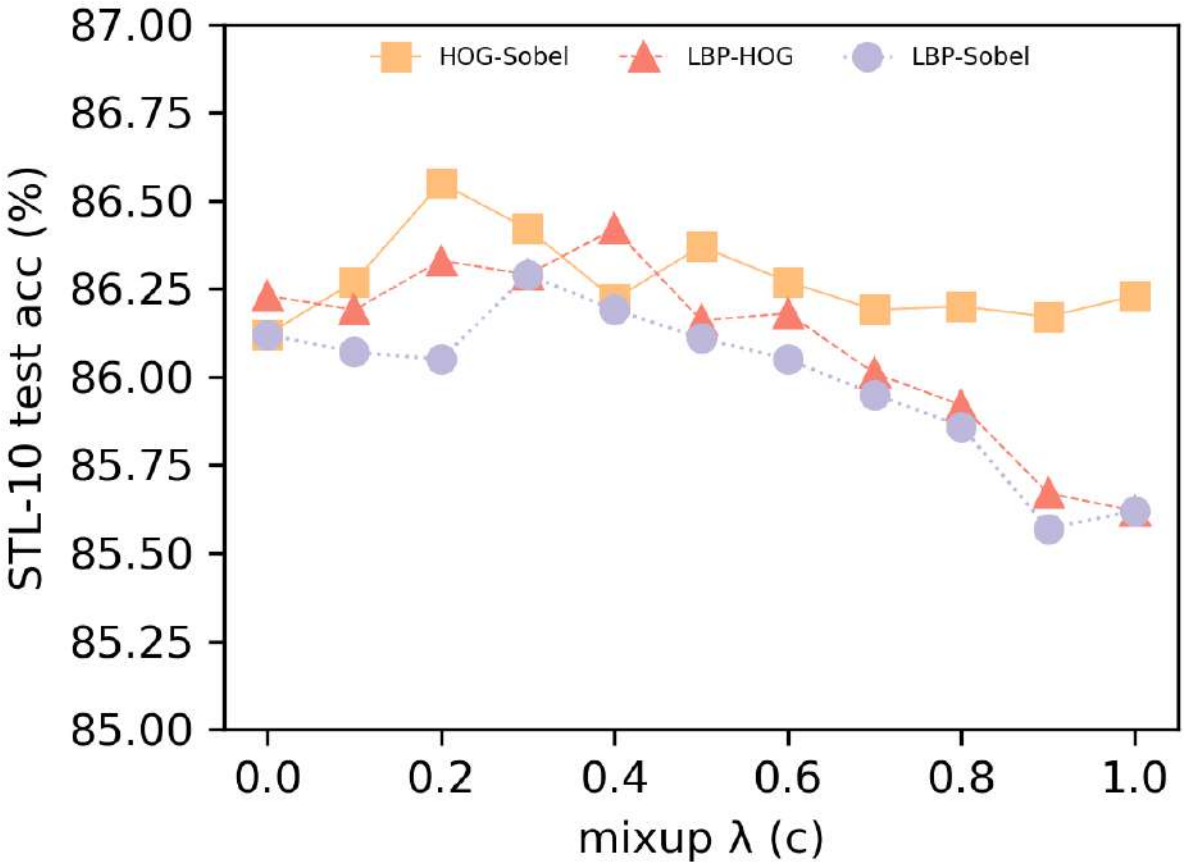



Figure 3.6 The impact of various mixup factors λ

The experimental results from the CIFAR-10, CIFAR-100, and STL-10 datasets are represented by subfigures (a), (b), and (c) respectively. On the x-axis, the mixup factor λ is shown, varying from 0 to 1 in steps of 0.1. The two Mixup features shown in Figure 3.6 are implemented based on (2.1), such as HOG-Sobel, where HOG refers to $f_1$ in (2.1) and Sobel refers to $f_2$. The y-axis represents the Top-1 accuracy of Full Fine-tuning.

The results from all three datasets consistently show that the HOG-Sobel Mixup feature combination outperforms the features from the other two Mixup pairings. Setting λ to 0 or 1 corresponds to using a single feature, either $f_2$ or $f_1$, for the purpose of reconstruction. The right configuration of λ can lead to better model

performance as compared to relying solely on a single feature. Yet, the experimental performance degrades with an emphasis on the LBP feature map due to less-than-ideal results from the LBP experiments. In the case of the HOG-Sobel combination, the best results were obtained in experiments across all three datasets when λ was set to 0.2 or 0.3. When λ leans towards the Sobel edge feature map, the reconstruction Mixup feature method yields the best results.

### 3.3.5 Comparisons with Previous Results

Table 3.5 Compared with previous methods (Top1-accuracy)

| Method | backbone | CIFAR-10 | CIFAR-100 | STL-10 |
|---|---|---|---|---|
| MoCo v2 [30] | ResNet-18 | 89.55 | 62.79 | 85.96 |
| BYOL [38] | ResNet-18 | 88.51 | 62.36 | 83.36 |
| SimCLR [27] | ResNet-18 | 86.01 | 58.21 | 82.35 |
| SimSiam [39] | ResNet-18 | 83.33 | 51.76 | 84.24 |
| MAE [33] | Vit-Small | 91.79 | 67.83 | 86.09 |
| Mask feat [68] | Vit-Small | 91.75 | **67.86** | 86.20 |
| Mixup-feature (ours) | Vit-Small | **92.09** | 67.77 | **86.55** |

To perform a more exhaustive evaluation and benchmark against previous methodologies, a fine-tuning process was executed across all listed self-supervised learning approaches, assessing their performance on three distinct datasets. The comprehensive results are delineated in Table 3.4. The selected architecture for generative self-supervised learning was the ViT-small, whereas ResNet-18 [69]. was utilized for contrastive learning approaches. The Mixup-feature technique was applied to reconstruct the HOG-Sobel feature map. When juxtaposed with the fine-tuned outcomes of the generative self-supervised ViT-Small model, the results were on par. On the CIFAR-10 and STL-10 datasets, the proposed method surpassed

others, achieving accuracies of 92.09% and 86.55%, respectively. For the CIFAR-100 dataset, performance was slightly lower than that of Mask Feat [68] by a margin of 0.9%. It is noteworthy that generative models based on Masked Image Modeling have exhibited substantial advancements over traditional contrastive learning methods. Notably, generative models generally necessitate less pre-training time than their contrastive learning counterparts. The pre-training duration for the proposed method was comparable to that required for the Masked Autoencoder approach.

### 3.3.6 Data Augmentation

The investigation explored the influence of various data augmentation techniques on the model's experimental outcomes. The experimental configuration for the method was established with a mask ratio of 50%, a mix factor (λ) set at 0.2, and the combination of HOG and Sobel feature maps as the target for reconstruction. Table 3.6 elucidates the discrepancies arising from distinct data augmentations employed during the Mixup feature pre-training phase. A comparative analysis was conducted on data augmentation methods comprising random cropping, color jittering, and RandAugment [70]. Experimental results indicate that only using random cropping for data augmentation is the most effective, color jittering has minimal impact on the results, and RandAugment [70] slightly reduced performance. The impact of data augmentation on the proposed method appeared to be relatively modest, indicating that commendable levels of accuracy are sustainable even without its application. In contrast to contrastive learning approaches that are heavily

dependent on data augmentation, achieving favorable outcomes in the absence of such augmentations poses a considerable challenge.

Table 3.6 The influence of the data augmentation (Top1-accuracy)

| Augmentation Method | CIFAR-10 | | CIFAR-100 | | STL-10 | |
|---|---|---|---|---|---|---|
| | ft | lin | ft | lin | ft | lin |
| none | 91.8 | 76.9 | 67.4 | 53.9 | 86.3 | 79.9 |
| rand crop | **92.1** | **77.9** | **67.8** | **54.6** | **86.6** | **80.4** |
| crop,color jit | 92.0 | 77.3 | 67.6 | 54.8 | 86.4 | 80.1 |
| crop, rand aug | 91.6 | 76.6 | 67.1 | 54.3 | 86.1 | 80.1 |

## 3.4 Experimental Analysis of the Denoising Self-Distillation MAE

This section evaluates the feature extraction capability of the Denoising Self-Distillation MAE self-supervised pre-trained encoder across CIFAR-10, CIFAR-100, and STL-10 datasets. The assessment focuses on classification performance via full fine-tuning and linear probing of the pre-trained encoder. Subsequent to this, comparisons with alternative methodologies are presented. An ablation study on the key components of the proposed method concludes this examination.

### 3.4.1 Implementation Details of the Denoising Self-Distillation MAE

In the experimental setup, several different ViT architectures were examined as the backbone network, including: ViT-tiny - Consisting of 12 Transformer blocks, each having a hidden layer dimension of 192. ViT-small - Also encompassing 12 Transformer blocks, but with each block boasting a hidden layer size of 384. ViT-

base - A more extensive architecture, comprising 12 Transformer blocks, each with a hidden layer capacity of 768.

The architectural design of the models in question incorporates a pronounced focus on the self-attention mechanism. The regression component integrates two Transformer blocks harnessing self-attention. Conversely, the decoder module comprises four self-attention blocks, culminating in a linear projection layer dedicated to prediction output. Corresponding to prior experimentation with Mixup features, CIFAR-10 and CIFAR-100 datasets were utilized, both consisting of images with 32x32 pixel dimensions. To accommodate the ViT-Tiny and ViT-Small architectures, images were segmented into a grid of 16x16, with each segment measuring 2x2 pixels. For the STL-10 dataset, which presents images of 96x96 pixels, a similar grid configuration of 16x16 was adopted, where each section spans 6x6 pixels to maintain consistency with the design standards of both ViT-Small and ViT-Base architectures.

The pre-training parameters and configurations for the conducted experiments are comprehensively tabulated in Table 3.7. For a visual appraisal, Figure 3.7 exhibits denoising outcomes on the STL-10 dataset, with the sequence showcasing noisy images, denoised results, and the original clean images. Due to limitations in computational resources, the current experiments were conducted on datasets of relatively reduced scale. This constraint, however, does not preclude the applicability of the method to more expansive datasets. To accommodate larger datasets, adjustments to the grid size and partitioning strategy would be necessitated, corresponding to the actual dimensions of the images in question.

Figure 3.7 The visual denoising results of the STL-10 dataset

Table 3.7 Pre-training settings of the Denoising Self-Distillation MAE

| Config | CIFAR-10 or 100 | STL-10 |
|---|---|---|
| architecture | ViT-T\ViT-S | ViT-S\ViT-B |
| batch size | 2048\2048 | 2048\1024 |
| patch size | 2 | 6 |
| optimizer | AdamW | |
| optimizer momentum | $\beta_1, \beta_2 = 0.9, 0.95$ | |
| weight decay | 0.05 | |
| learning rate schedule | Cosine decay | |
| base learning rate | 1.5e-4 | |
| warmup epochs | 20 | |
| Epoch | 500 | 500 |
| EMA $\eta$ | cosine scheduler (0.96,0.99) | |

### 3.4.2 Evaluation

In the experimental evaluation phase of this study, pioneering works from prior research were chosen as benchmarks to ensure strict comparability and fairness in the comparative analysis. To ensure consistency in the backbone network used during experimentation, the self-supervised learning method, MoCo v3 [71], based on the VIT backbone network, was selected from the contrastive learning paradigm. Concurrently, seminal works like BEIT [32], MAE [33], and Mask Feat [68] were selected from the domain of Masked image modeling. It's noteworthy that both CAE [72] and SdAE [73] employ a branched structure, demonstrating a predilection for optimizing models within the feature domain. To be specific, SdAE's student branch utilizes an encoder-decoder structure to reconstruct masked information from the input image, whereas its teacher branch aims to produce latent representations of masked tokens. The proposed method distinctively optimizes pixel-level

reconstruction in tandem with feature-level regression. A thorough comparison with established methodologies facilitates a comprehensive assessment of the performance and merits of this approach.

Table 3.8 presents the evaluation results of the pre-trained models based on the ViT backbone on three datasets. Subtable (a) illustrates the evaluation outcomes of the full fine-tuning experiments, while subtable (b) presents the results from the linear probing experiments.

Table 3.8 Results of the Pre-training Evaluation Experiment

(a) full fine-tuning

| **Method** | **pre-trained Epochs** | **CIFAR-10** | | **CIFAR-100** | | **STL-10** | |
|---|---|---|---|---|---|---|---|
| | | **ViT-tiny** | **ViT-small** | **ViT-tiny** | **ViT-small** | **ViT-small** | **ViT-base** |
| scratch baseline | - | 73.88 | 79.86 | 51.55 | 56.17 | 77.98 | 82.41 |
| BEiT | 500 | 88.93 | 90.65 | 66.32 | 66.93 | 84.32 | 86.22 |
| MoCo v3 | 500 | 88.91 | 90.88 | 66.17 | 67.39 | 84.61 | 87.07 |
| MAE | 500 | 88.77 | 90.26 | 65.93 | 66.51 | 85.63 | 86.38 |
| MAE | 1200 | 89.87 | 91.79 | 66.72 | 67.83 | 86.20 | 87.69 |
| Mask feat | 1200 | **90.12** | 91.75 | 66.83 | **67.86** | 86.23 | 87.85 |
| CAE | 300 | 89.93 | 91.56 | 66.84 | 67.83 | 86.08 | 87.76 |
| SdAE | 300 | 89.98 | 91.83 | 66.96 | 67.79 | 85.90 | 87.71 |
| Ours | 500 | 89.76 | **91.94** | **67.23** | 67.77 | **86.31** | **87.86** |

(b) linear probing

| **Method** | **pre-trained Epochs** | **CIFAR-10** | | **CIFAR-100** | | **STL-10** | |
|---|---|---|---|---|---|---|---|
| | | **ViT-tiny** | **ViT-small** | **ViT-tiny** | **ViT-small** | **ViT-small** | **ViT-base** |
| BEiT | 500 | 47.68 | 56.73 | 27.89 | 33.78 | 42.21 | 48.57 |
| MoCo v3 | 500 | **76.20** | **77.91** | **50.46** | **54.41** | **78.87** | 82.75 |
| MAE | 500 | 73.77 | 76.78 | 48.22 | 51.89 | 76.10 | 80.16 |
| MAE | 1200 | 75.87 | 77.53 | 50.19 | 54.08 | 77.96 | 82.36 |
| Mask feat | 1200 | 75.58 | 77.43 | 50.25 | **54.41** | 78.32 | 82.61 |
| CAE | 300 | 74.90 | 77.15 | 50.21 | 54.26 | 78.48 | 81.62 |
| SdAE | 300 | 75.17 | 76.86 | 50.01 | 53.71 | 78.30 | 82.11 |
| Ours | 500 | 75.53 | 76.98 | 50.17 | 53.84 | 78.56 | **82.79** |

Experimental results shows that the proposed method yielded enhanced performance during full fine-tuning using the VIT-Small backbone on the CIFAR-10

and STL-10 datasets, registering an approximate boost of 0.1% in top-1 accuracy. Moreover, when harnessing the VIT-tiny backbone for full fine-tuning on CIFAR-100, a rise of about 0.3% in top-1 accuracy was observed. Similarly, a VIT-base backbone recorded a nearly 0.1% increase in top-1 accuracy during full fine-tuning on the STL-10 dataset. This experimental evidence robustly attests to the efficacy of the introduced methodology, matching the performance benchmarks set by MAE and Mask Feat. Remarkably, our technique achieved the performance levels of MAE pretraining within a span of 500 epochs, contrasting with MAE's need for 1200 epochs, indicating a pronounced decrease in pretraining time. When compare with SdAE, a performance improvement of 0.1% in top-1 precision was realized, further substantiating the effectiveness of our self-distillation design and image denoising. In full fine-tuning evaluation experiments across three datasets with diverse ViT architectures, our method consistently outperformed the classical contrastive learning approach, MoCo v3. However, in evaluations employing linear probing, the contrastive learning-based MoCo v3 surpassed MIM-based methodologies. This aligns with prior empirical evidence suggesting the inherent superiority of contrastive learning over MIM-based approaches in the context of linear probing [33,72].

### 3.4.3 Ablation Study

The framework consists of a teacher encoder, student encoder, student decoder, and regressor. Ablation studies investigated the impact of employing solely the $L_y$ loss, which omits the teacher network and regressor, and the $L_z$ loss independently, which excludes the denoising process performed by the student decoder. Performance

was compared in linear probing assessments across three datasets, with results detailed in Table 3. Utilization of only the $L_y$ loss resulted in a minor performance drop of approximately 2% to 4% in the linear probing tests. Conversely, a reliance solely on the $L_z$ led to a marked decrease in performance. These outcomes substantiate the joint effectiveness of $L_y$ and $L_z$ losses in bolstering the feature extraction capability of the pre-trained model across both pixel-level and high-dimensional feature domains.

Table 3.9 presents ablation studies on the model framework. All models employed the VIT-Small as their primary backbone and were pretrained for 300 epochs across CIFAR-10, CIFAR-100, and STL-10 datasets.

Table 3.9 Ablation studies on the model framework

| Decoder | regression decoder | CIFAR-10 | CIFAR-100 | STL-10 |
|---|---|---|---|---|
| √ | × | 74.20 | 49.74 | 75.35 |
| × | √ | 68.93 | 44.51 | 61.83 |
| √ | √ | 76.98 | 53.84 | 78.56 |

The study of the noise masking rate's effect on experimental results in MAE shows that the masking rate in masked image modeling can reach 75%. The proposed methodology amalgamates Masked Autoencoders (MAE) with knowledge distillation, necessitating an elevated noise masking ratio for the denoising of impaired images, as depicted in Figure 3.8. Experimental results from three different datasets indicate that the best performance can be achieved when the noise masking rate is between 90% and 95%. Employing an exceedingly low noise masking rate can render the proxy task for the pre-trained model overly simplistic, resulting in diminished feature extraction prowess.

The impact of the $\lambda$ parameter in Equation 2.15 on the feature extraction capabilities of the pre-trained model was examined by experimenting with a range of $\lambda$ values. Figure 3.9 illustrates the fine-tuning outcomes of the pre-trained model. Experimental results from three datasets indicate that the best fine-tuning performance of the pre-trained model is achieved when the λ value is set to 0.6.

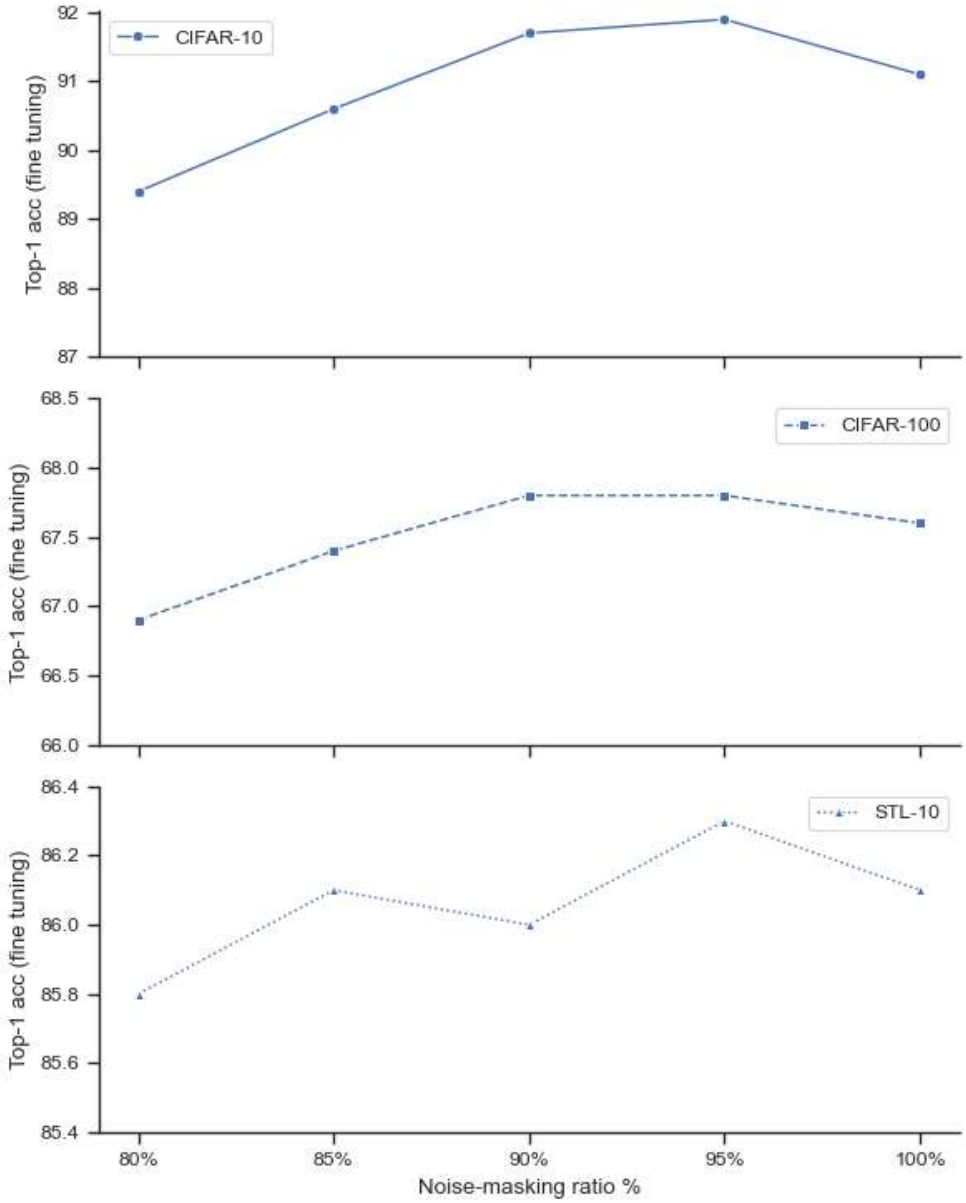



Figure 3.8 investigates the impact of the noise masking rate on experimental results.

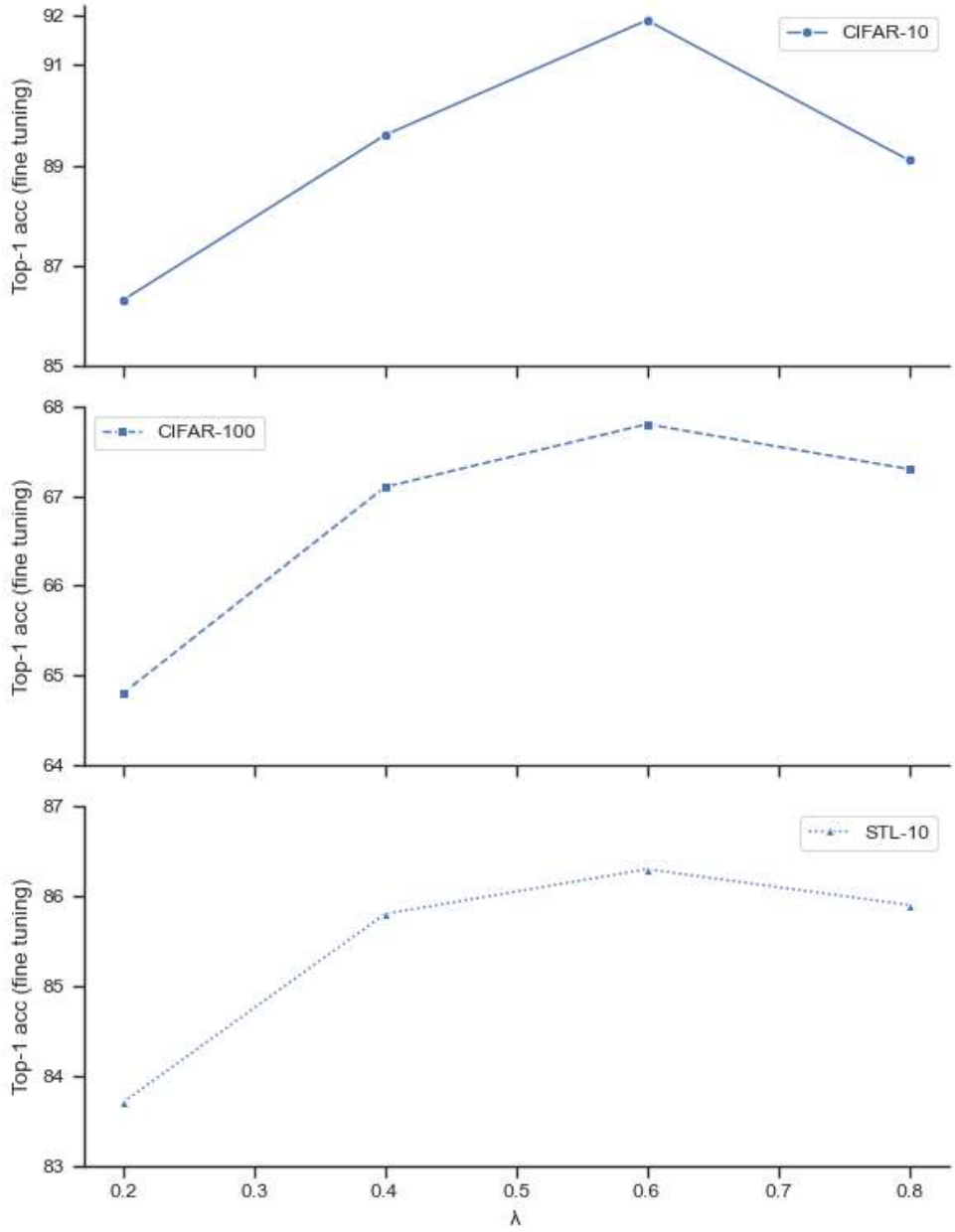



Figure 3.8 investigates the impact of the λ value on experimental results.

## 3.5 Conclusion of Chapter 3

In Chapter 3, comprehensive experiments were conducted to verify the two proposed self-supervised learning algorithms - Mixup Feature and Denoising Distillation Masked Autoencoder.

For Mixup Feature, the model was pretrained on the CIFAR-10, CIFAR-100, and STL-10 datasets. Downstream evaluation involves linear probing and full fine-tuning. Results showed that mixing traditional features like Sobel, HOG, LBP improved performance compared to single features. The unnormalized combination of features delivered the most optimal performance when set as targets. Ideally, a masking ratio ranging from 40-60% is preferred. The mixup factor λ of 0.2-0.3 yielded the best results for the HOG-Sobel mixup. This method matches or surpasses MAE and is superior to contrastive learning.

For the Denoising Distillation Masked Autoencoder, the model adopted ViT as the backbone network and was pretrained on the same three datasets. Removing randomly added Gaussian noise blocks served as the pretext task. Full fine-tuning and linear probing benchmark tests exhibited comparable performance to MAE and MaskFeat within 500 epochs, faster than the 1600 epochs of MAE. This strategy outperforms the seminal contrastive learning approach of MoCo v3. The ablation study confirmed the combined effect of pixel reconstruction loss and feature regression loss, demonstrating the effectiveness of the proposed framework. Optimal results can be achieved with a 90-95% noise masking rate and a λ value of 0.6 from equation 2.15.

In conclusion, comprehensive experiments demonstrated the effectiveness of the two proposed self-supervised learning algorithms on three different datasets compared to existing methods. The results verified the innovation of advancing masked image modeling techniques through mixed feature objectives and denoising

distillation MAE. Future work can evaluate extending these methods to larger datasets.

Subsequent chapters will delve into the specific applications and experimental evaluations of two prominent self-supervised learning algorithms within the realm of medical image analysis. The potential of self-supervised learning is becoming increasingly evident across various computer vision tasks, especially in scenarios where data annotation is scarce or costly. The challenges of annotating medical images, due to their specialized nature and sensitivity, underscore the significance of self-supervised learning in this field. Empirical studies on these algorithms will be conducted to validate their effectiveness and viability for real-world medical imaging tasks. The MAE method will serve as a benchmark, facilitating a quantitative comparison with self-supervised learning algorithms across various metrics. Comprehensive insights regarding experimental design, dataset description, results analysis, and further discussions will be provided in the ensuing chapter.

# CHAPTER 4. APPLICATION OF SELF-SUPERVISED PRETRAINED MODELS IN CT SCAN CLASSIFICATION

Obtaining data labels in the field of medical imaging is a challenging task, Professional radiologists are required for accurate annotations [80]. The dataset of medical images exhibits a significant imbalance, where positive samples are notably fewer than negative samples [81]. Through self-supervised pre-training, models can attain general feature extraction abilities even without labels, additionally, these pre-trained models exhibit reduced sensitivity to dataset imbalances [82].

This chapter primarily details the application of self-supervised pre-trained models for CT scan classification. CT (Computerized Tomography) scanning is pivotal in medical imaging, diagnosing a range of diseases and symptoms. With the development of self-supervised learning techniques, self-supervised pre-training can learn useful representations from unlabeled CT scan data. This chapter delves into harnessing self-supervised pre-trained models to derive significant features from CT scans, enabling precise and efficient CT scan classification. The process of pre-training using CT scan data and the subsequent fine-tuning and classification experimental results are discussed. Additionally, this research examines the performance and advantages of the proposed self-supervised pre-trained models in comparison to traditional supervised methods for CT scan classification. Through extensive experiments, this chapter emphasizes the potential of self-supervised pre-trained models as a promising method to enhance CT scan analysis and improve medical outcomes.

## 4.1 CT Scan Datasets

COVID-CTset [74] is a vast dataset of COVID-19 CT scans, comprising over 60,000 CT images. In our experiments, this dataset served as the pre-training foundation during the self-supervised learning stage.

The COVID-CT-Dataset [75] is a concise dataset, featuring 397 COVID19-negative samples and 349 COVID19-positive samples. This dataset is utilized for the fine-tuning and classification tests of the pre-trained model.

The SARS-CoV-2 [76] CT scan dataset, hosted on the Kaggle platform, comprises 2482 samples in total. The positive and negative samples in the dataset are roughly balanced. tuning and classification experiments of the pre-trained model. This dataset is similarly employed for the fine-tuning and classification trials of the pre-trained model. It serves as a comparative experiment against the COVID-CT-Dataset.

## 4.2 Self-Supervised Pre-Training on CT Scan Dataset

The COVID-CTset dataset serves as the training set for this phase of self-supervised pre-training, without the use of labels. All images in the dataset are uniformly cropped using RandomResizedCrop [77] to dimensions of 224x224, incorporating the RandomHorizontalFlip [77] data augmentation technique.

Pre-training was conducted on three distinct self-supervised learning methods: MAE [33], Mixup Feature [67], and Denoising Self-Distillation MAE [85]. Each of the three methods employs the same backbone network, specifically the ViT-Small.

The patch size designated for segmenting the images is set at 14, with other parameter configurations adhering to the experimental setup outlined in Chapter 3. MAE focuses on the reconstruction of masked pixels and requires 1200 epochs for training iterations. Mixup Feature aims to reconstruct the target feature maps of masked images and undergoes 800 epochs for training iterations. Denoising Self-Distillation MAE is designed to denoise random noise and is trained over 500 epochs. Since MAE involves pixel-level reconstruction, it necessitates a greater number of epochs for pre-training.

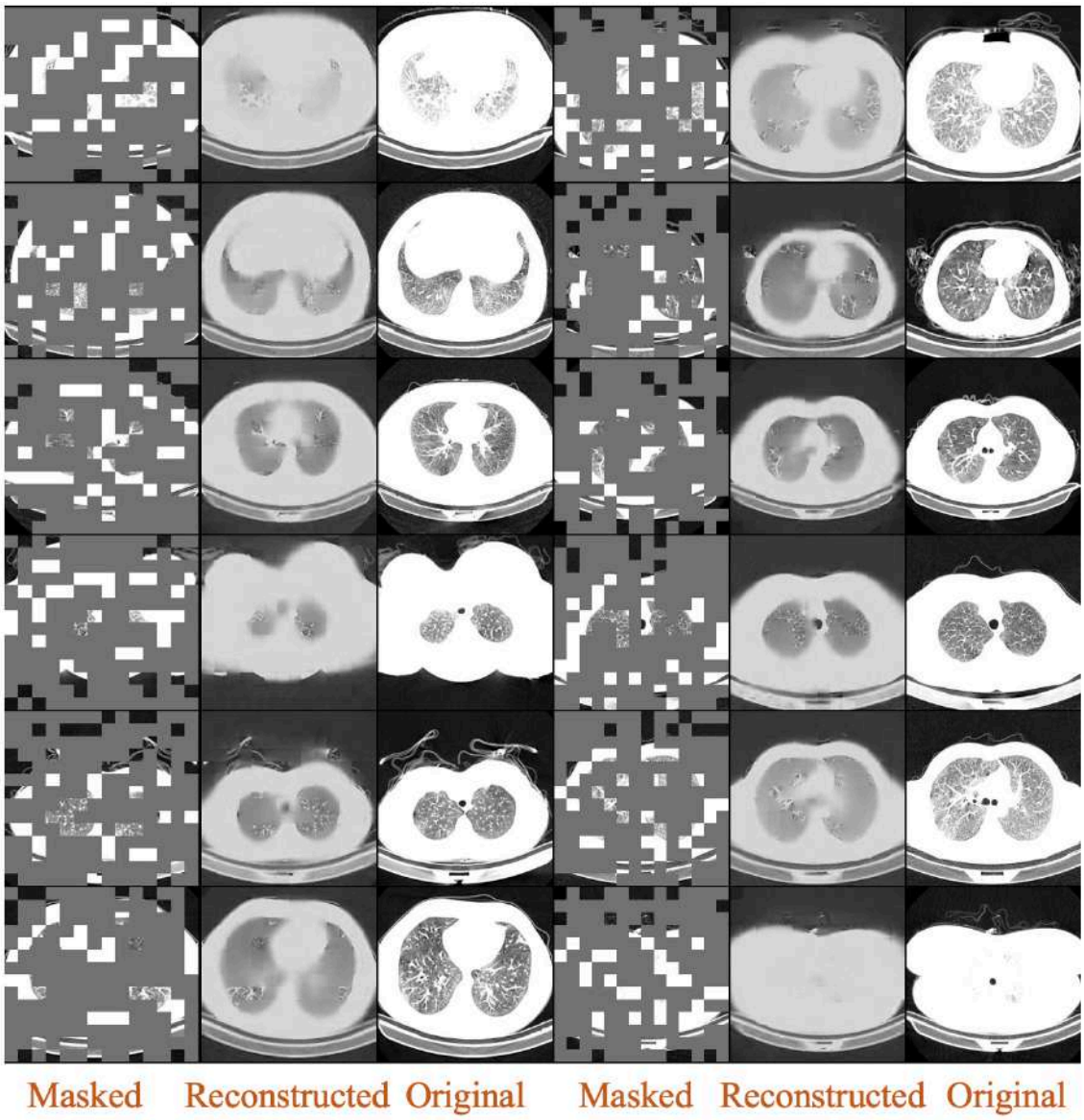



Figure 4.1 Reconstruction of the COVID-CTset validation set images using MAE.

The quality of the reconstructed images or feature maps is evaluated using the images from the validation set. As depicted in Figure 4.1, the MAE pre-trained model demonstrates the quality of reconstructed images from the COVID-CTset validation set after undergoing 1200 epochs of training. The masking rate of the images are 75%.

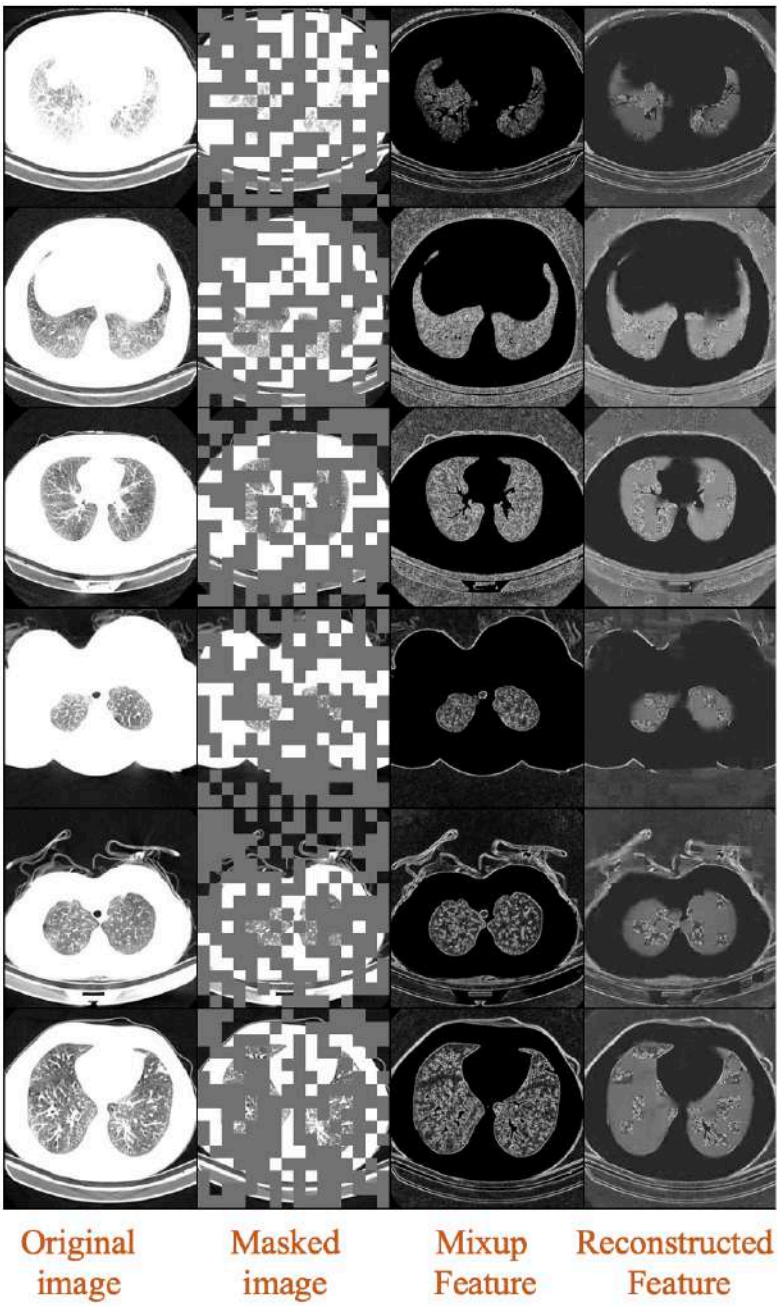



Figure 4.2 Reconstructed Features of the COVID-CTset validation set images using Mixup Feature.

Figure 4.2 illustrates the feature map reconstruction outcomes achieved by the Mixup Feature self-supervised pre-trained model on the COVID-CTset validation set. The masking rate of the image is 60%, The target feature map is derived using the HOG-Sobel Mixup Feature strategy applied to the original image, with a $\lambda$ setting of 0.3, for further details of $\lambda$, refer to equation 2.1.

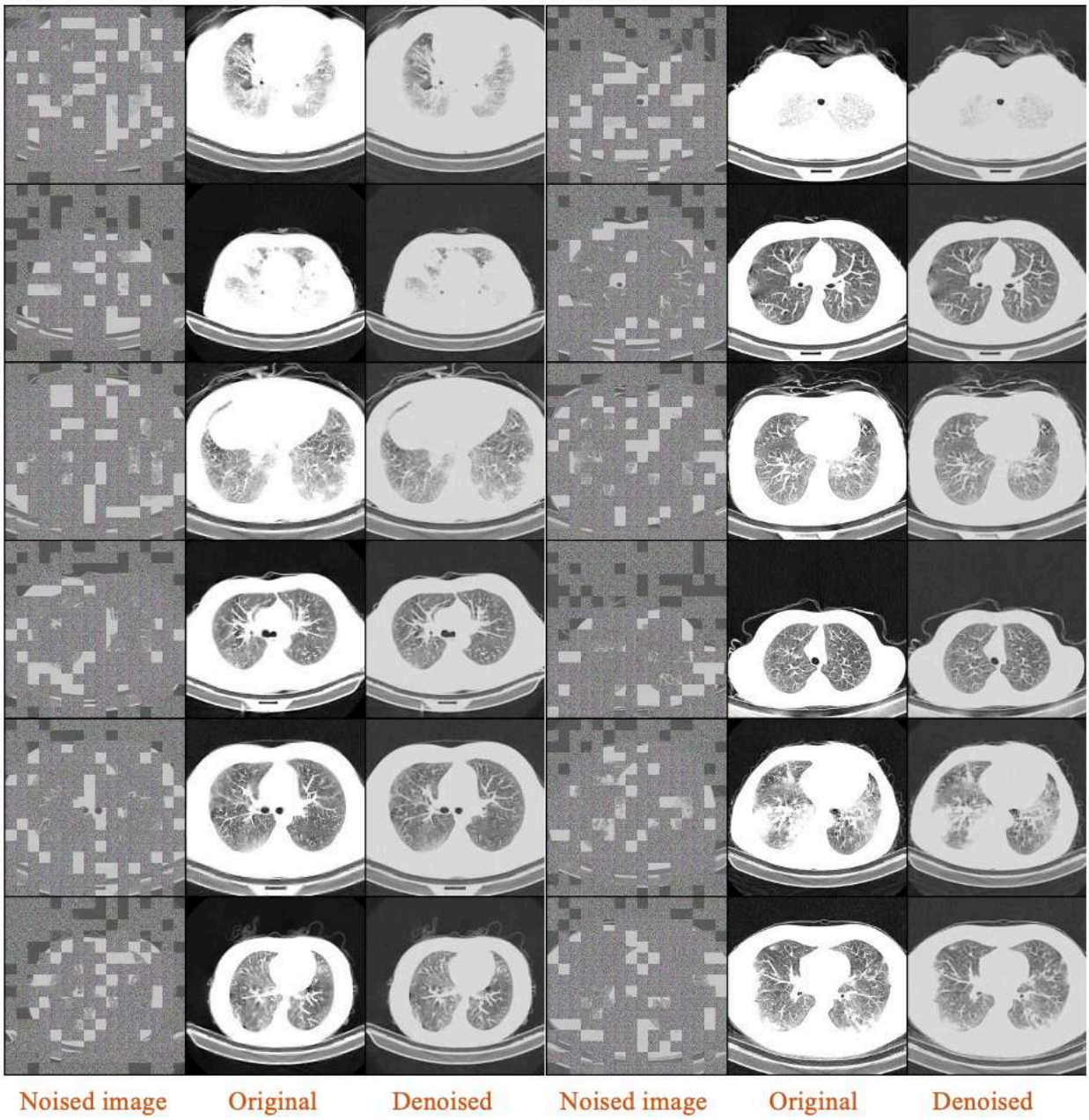



Figure 4.3 Denoised images of the COVID-CTset validation set images using Denoising Self-Distillation MAE.

Figure 4.3 illustrates the reconstruction outcomes of the Denoising Self-Distillation MAE self-supervised pre-trained model on the COVID-CTset validation set images, post 500 epochs of training. The pre-trained model has a noise masking rate of 80%.

The models underwent pre-training on the COVID-CTset, and visualization outcomes suggest that each can predict the overall anatomical structure within CT Scan images, yet both MAE and Mixup Feature fall short in reconstructing the intricate textures of lung tissue slices. This outcome is anticipated, given that MAE perceives only 25% of the input image and endeavors to reconstruct the remaining 75% during training, a feat even seasoned radiologists find daunting. Denoising Self-Distillation MAE is adept at not only reconstructing the overall anatomical structure but also capturing the detailed textures of lung tissue slices. This capability stems from its structural design, which emphasizes both global information and local information.

However, no concrete evidence suggests a positive correlation between reconstruction capability and transfer learning performance. Conversely, original autoencoders with a 0% mask ratio [78] undoubtedly outperform mask autoencoders in image reconstruction, but the representations they produce are less effective in downstream tasks than those from masked autoencoders [79]. Ultimately, it's worth noting that the primary objective of self-supervised learning isn't the image reconstruction task per se, but rather a proxy task within the realm of self-supervised learning. Objective evaluation must be based on its generalization capability and transferability in downstream tasks.

### 4.3 Fine-Tuning on CT Scan Dataset

To further assess the three self-supervised pre-trained models described in section 4.2, two sets of downstream tasks were established. The self-supervised pre-trained models were subject to Full Finetuning experiments on both the COVID-CT-Dataset and the SARS-CoV-2 dataset. These datasets were segmented into training, test, and validation sets at a split ratio of 4:1:1. Data samples were resized to 224*224 pixels, aligning with the input size of the pre-trained models. The dataset maintains a balance between positive and negative samples.

Table 4.1 The Full Fine-tuning results on test set (%)

| Model | COVID-CT | | | SARS-CoV-2 | | |
|---|---|---|---|---|---|---|
| | Acc | F1 score | AUC | Acc | F1 score | AUC |
| ViT-scratch | 87.46 | 87.70 | 91.57 | 90.33 | 90.40 | 94.48 |
| MAE(STL) | 91.38 | 91.78 | 96.74 | 92.37 | 92.07 | 96.99 |
| Mixup(STL) | 92.43 | 92.57 | 96.83 | 92.98 | 92.91 | 97.24 |
| DMAE(STL) | 90.32 | 90.81 | 95.39 | 91.88 | 91.18 | 96.76 |
| DMAE(Med) | 91.25 | 91.54 | 96.70 | 93.18 | 93.06 | 98.37 |
| MAE(Med) | 93.16 | 93.41 | 97.61 | 98.26 | 98.25 | 99.65 |
| Mixup(Med) | **93.01** | **93.26** | **97.33** | **98.08** | **98.13** | **99.49** |

During the Full Finetuning experiment, the model underwent training for 200 epochs. A comparative experiment was also conducted, where training occurred directly on these two datasets without using pre-trained models (ViT-scratch). To investigate the model's transfer learning capabilities, the pre-trained models from experiments in sections 3.3 and 3.4 were utilized on the STL-10 dataset for Full Fine-tuning. The datasets were resized to conform to the STL-10 sample dimensions of 96x96 pixels. The results of the Full Fine-tuning on these test sets are presented in Table 4.1.

The table 4.1. presents the outcomes of each approach across the two test datasets. "Acc" denotes accuracy, "F1 score" represents the harmonic mean of model precision and recall [83], "AUC" stands for Area Under the Curve [84]. In the Model's column, "Med" signifies those pre-trained on the COVID-CTset. "STL" designates models pre-trained on the STL-10 dataset. "scratch" refers to models trained directly on the medical dataset.

Experimental results indicate that the three self-supervised pre-training methods outperform direct training on both datasets, showcasing a noticeable enhancement. On the COVID-CT dataset, using the MAE pre-training approach led to an improvement of 5.7% in accuracy, 5.71% in F1 score, and 6.04% in AUC when compared to the scratch training method. For COVID-CT dataset, the Mixup Feature pre-training approach yielded a 5.55% increase in accuracy, a 5.56% enhancement in F1 score, and a 5.76% rise in AUC, in comparison to the scratch training technique. Using the DMAE pre-training approach on COVID-CT dataset resulted in an enhancement of 3.79% in accuracy, 3.84% in F1 score, and 5.13% in AUC when set against the scratch training protocol. For the SARS-COV-2 dataset, the MAE pre-training method, when compared to the scratch training, showed an accuracy improvement of 7.93%, an F1 score enhancement of 7.85%, and an AUC rise of 5.17%. On the SARS-COV-2 dataset, when juxtaposing the Mixup Feature pre-training method with the scratch training, we observed an accuracy boost of 7.75%, an F1 score increment of 7.73%, and an AUC elevation of 5.01%. For the SARS-COV-2 dataset, when juxtaposing the DMAE pre-training method with the scratch

training, we observed an accuracy boost of 2.85%, an F1 score increment of 2.66%, and an AUC elevation of 3.89%.

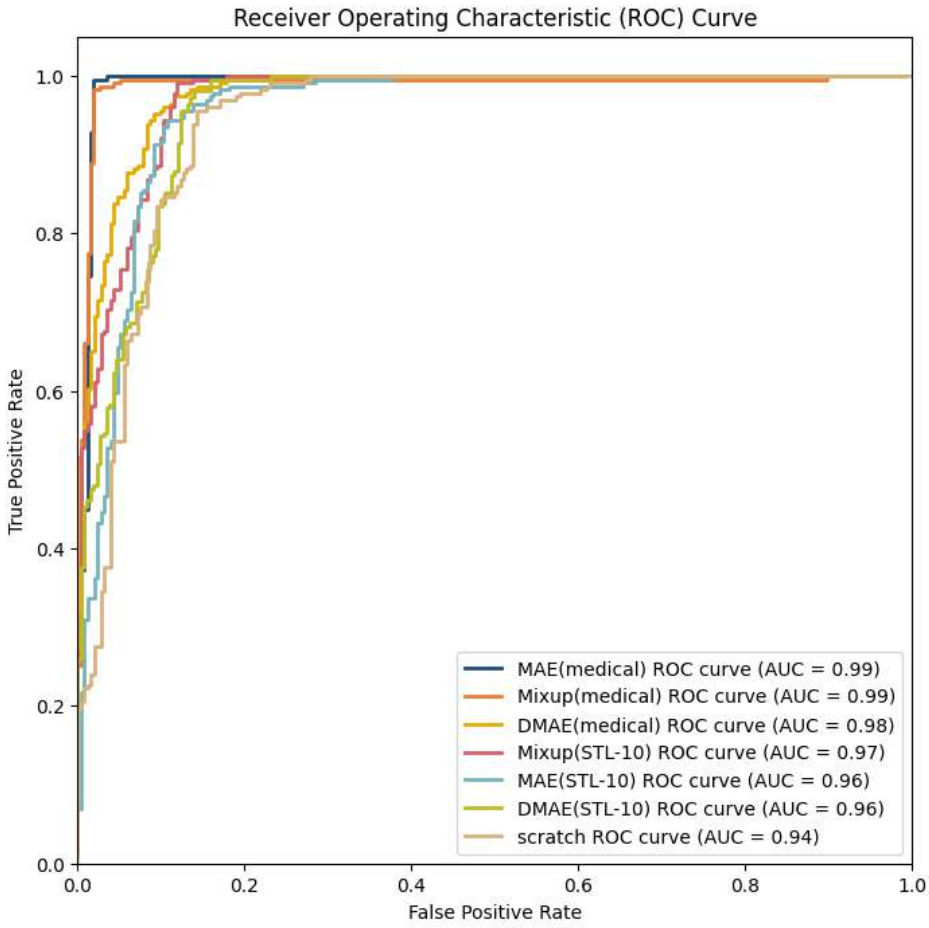



Figure 4.4 The AUC plot of the different training strategies on SARS-COV-2

This further underscores that leveraging self-supervised pre-training is a highly effective method to boost model accuracy. The Mixup Feature (Med) pre-training approach yielded results closely aligned with MAE (Med), and interestingly, the Mixup Feature (STL) even marginally exceeded MAE (STL). This further emphasizes the promising potential of the Mixup Feature method I introduced within the medical image processing domain. DMAE's overall performance lags behind Mixup Feature and MAE. Although DMAE's reconstructed target images excel in

section 4.2 over Mixup Feature and MAE, it doesn't necessarily imply superior feature extraction capabilities than those methods.

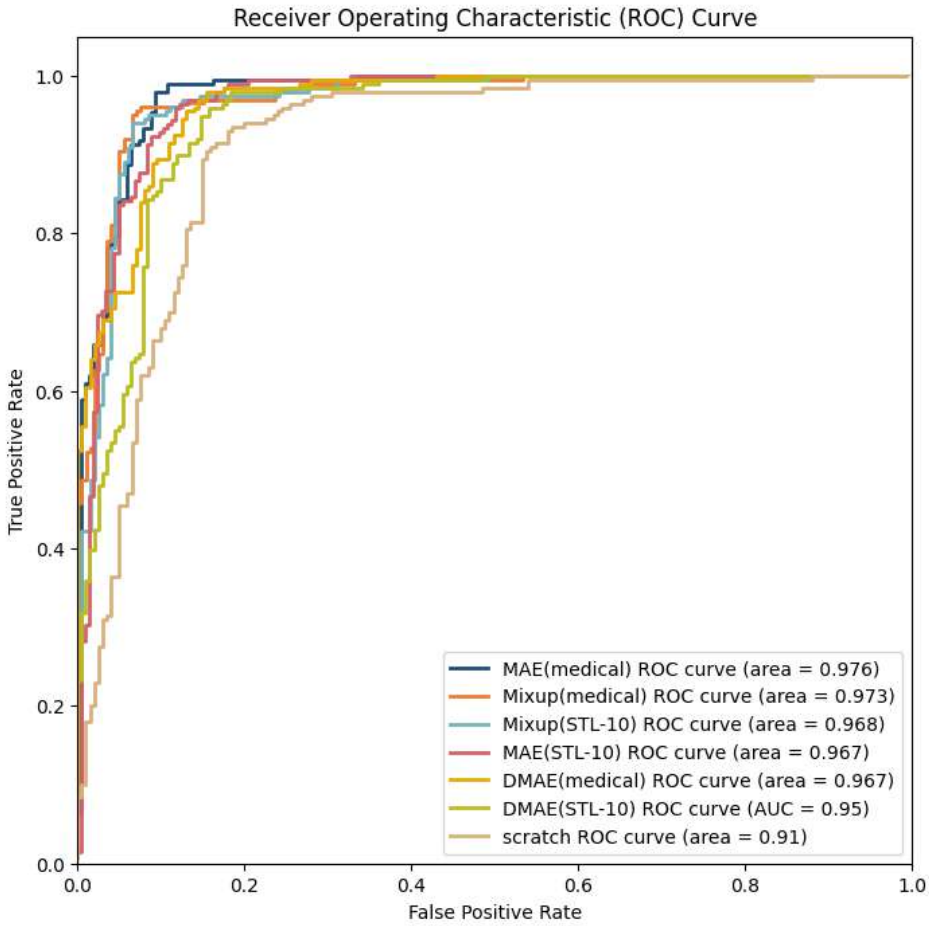



Figure 4.5. The AUC plot of the different training strategies on COVID-CT

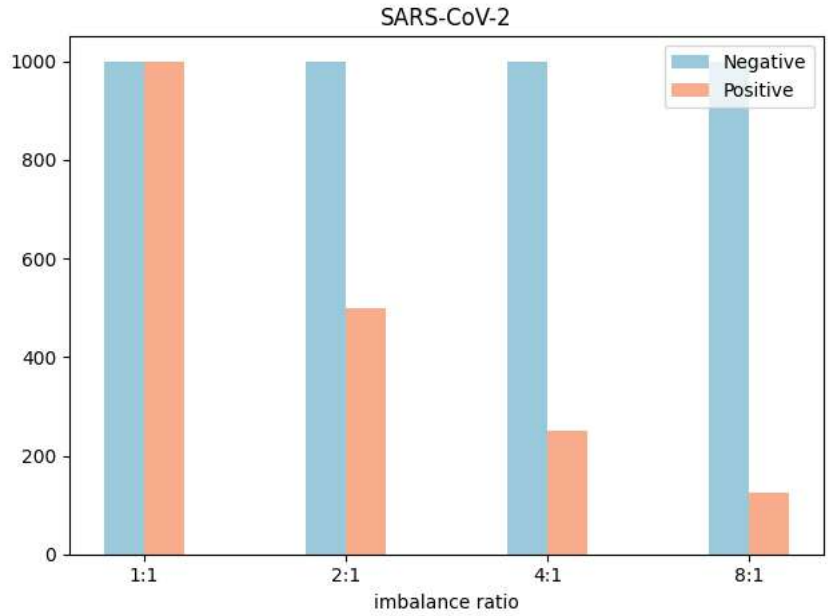
Figure 4.4 illustrates the AUC curves of seven training methods when applied to the SARS-COV-2 test dataset, Figure 4.5 portrays the AUC curves of seven training methods for the COVID-CT test dataset, both experimental outcomes consistently highlight that MAE (Med) and Mixup (Med) excel in AUC performance relative to the other strategies presented. Fine-tuning pre-trained models on medical image datasets yields superior results compared to the STL-10 dataset. The considerable differences between STL-10 data samples and medical image data samples account for this. The Mixup feature, when pre-trained on STL-10, exhibits superior

performance over MAE in cross-domain imaging tasks, demonstrating the transfer learning capability of the Mixup feature.

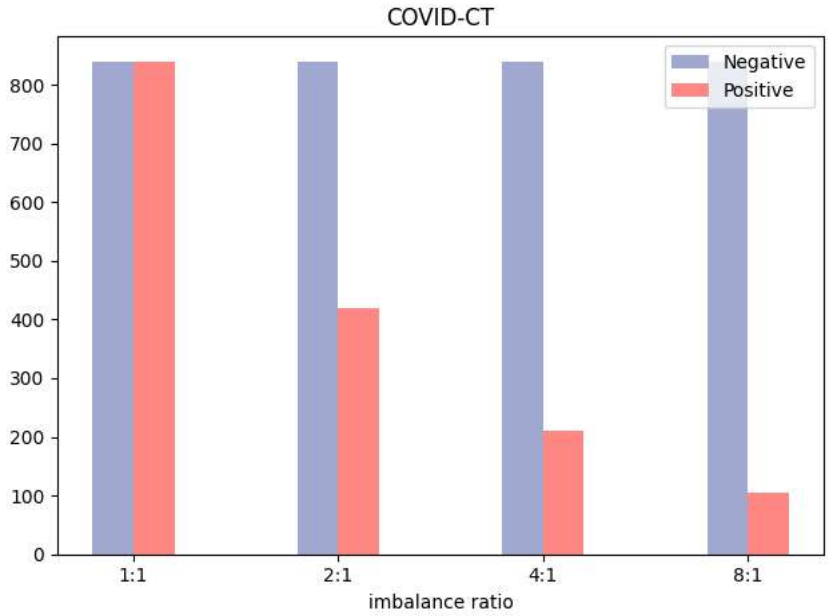
## 4.4 Investigating the Robustness of Self-Supervised Pre-Trained Models on Imbalanced Datasets

In medical imaging, datasets frequently exhibit imbalances, often stemming from factors like the rarity of specific diseases or challenges in data acquisition. Such imbalances can potentially influence model training and its subsequent performance. In this section, an experimental setup was designed specifically for imbalanced datasets, with the objective to thoroughly evaluate the robustness and efficacy of self-supervised pre-trained models in such contexts. This experimental design facilitates a more comprehensive understanding of the potential value and limitations of models in practical medical applications.

To more closely resemble real-world medical imaging scenarios, where the number of positive samples (such as confirmed cases) is often much lower than negative samples, the SARS-COV-2 dataset and COVID-CT dataset were used as experimental bases. By integrating the two datasets, training sets were constructed with varying imbalance ratios to mimic real-world data distributions. Four distinct positive-to-negative sample ratios were configured: 1:1, 1:2, 1:4, and 1:8, as depicted in Figure 4.6. This configuration was critical to enable a comprehensive evaluation of the robustness of the self-supervised pre-trained models under diverse imbalanced conditions.




(a)




(b)

Figure 4.6. The training set is configured with imbalanced ratios. (a) details the imbalance ratio settings for the SARS-COV-2 dataset, while (b) provides the imbalance ratio settings for the COVID-CT dataset.

The investigation into the performance of self-supervised pre-trained models on imbalanced datasets involved the creation of several imbalanced data scenarios,

illustrated in Figure 4.6. Full finetuning was executed for three models: Masked Autoencoder (MAE), Mixup Feature, and Denoising Masked Autoencoder (DMAE).

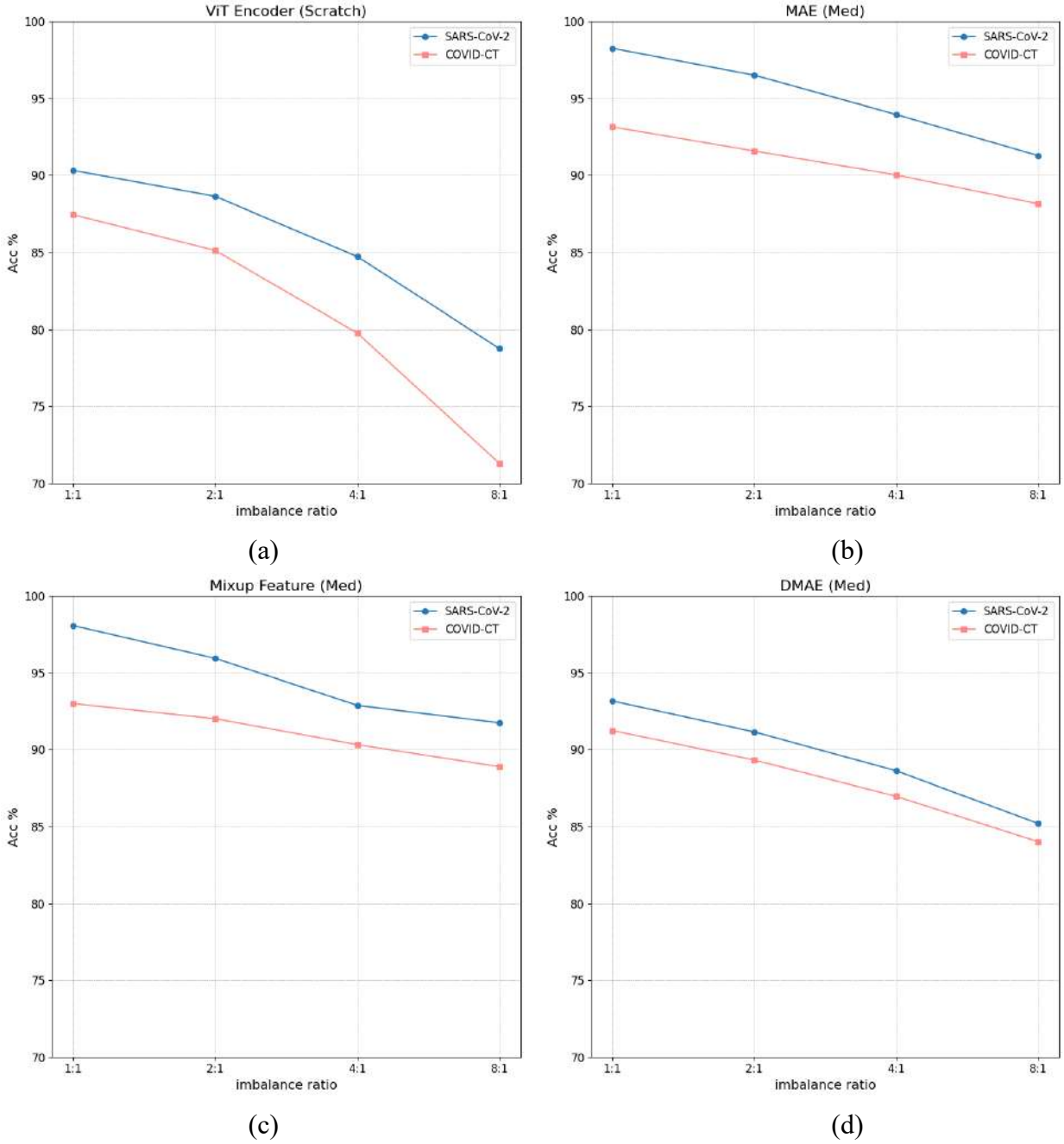



Figure 4.7. Experimental results under different imbalance ratios, the blue line signifies the experimental results from the SARS-COV-2 dataset. the orange line depicts the experimental results derived from the COVID-CT dataset. (a)ViT Encoder Scratch，(b) MAE pre-trained on COVID-CTset, (c) Mixup Feature pre-trained on COVID-CTset, (d) DMAE pre-trained on COVID-CTset.

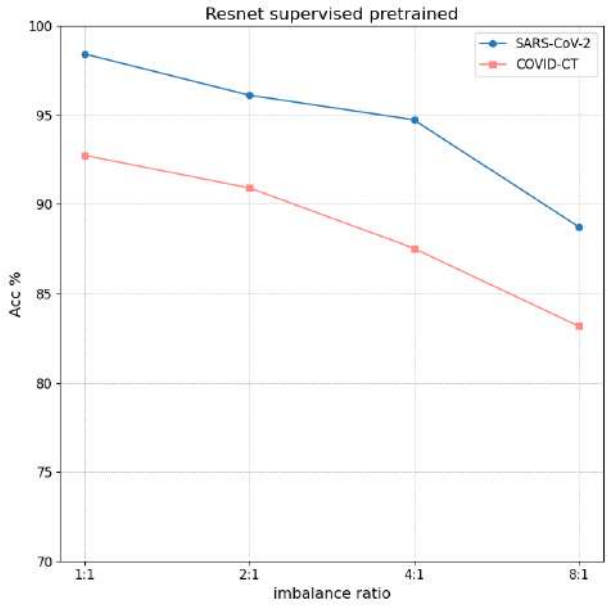
Experiments were conducted on these pre-trained models on imbalanced downstream task datasets to verify their robustness. For comparison, the ViT-Encoder was also directly trained on the said imbalanced dataset.

The experimental findings indicate that models pre-trained via self-supervision notably surpassed the direct use of ViT-Encoder in terms of robustness. Specifically, as shown in Figure 4.7, with the increase in imbalance ratio, the performance of the directly trained ViT-Encoder dropped sharply on two downstream task datasets, and the decline was significant. In contrast, the three models pre-trained through self-supervision showed a more stable decline in performance facing different imbalance ratios, and the drop was relatively smaller. This observation implies that self-supervised pretraining indeed provides models with a more robust foundation for feature learning. In the ensuing fine-tuning phase, such a foundational learning aids models in adeptly navigating the challenges presented by imbalanced data, leading to more precise medical image analyses.

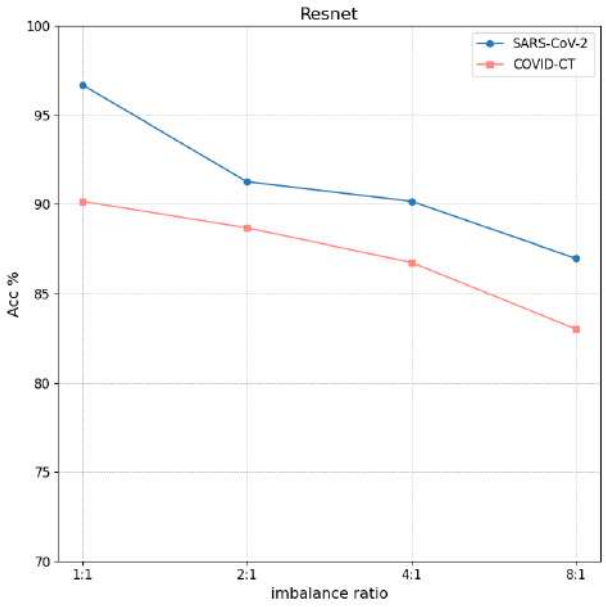
To deeply explore the performance differences between self-supervised pre-training and supervised pre-training on imbalanced datasets. In comparison with CNN models, our objective is to discern the effects of the two distinct pre-training techniques on the experimental outcomes and to elucidate the observations made. In this experiment, we selected three advanced CNN models for evaluation: EfficientNet [86], ResNet [41], and VGG [87]. These three models are widely used in various computer vision tasks and have excellent performance on many standard datasets [88][89][90].

For comparative analysis, we adopted two different pre-training strategies for each CNN model:

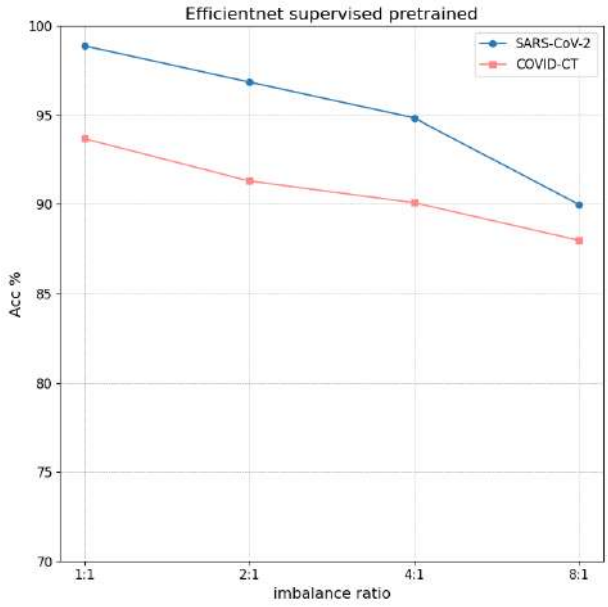
Models pre-trained on ImageNet [54]: These models underwent full supervised pre-training on the large ImageNet dataset and were fine-tuned based on this for the imbalanced dataset.



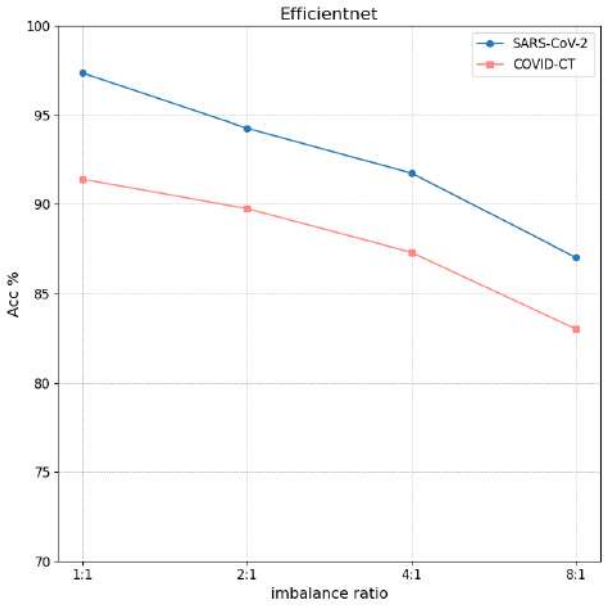

(a)



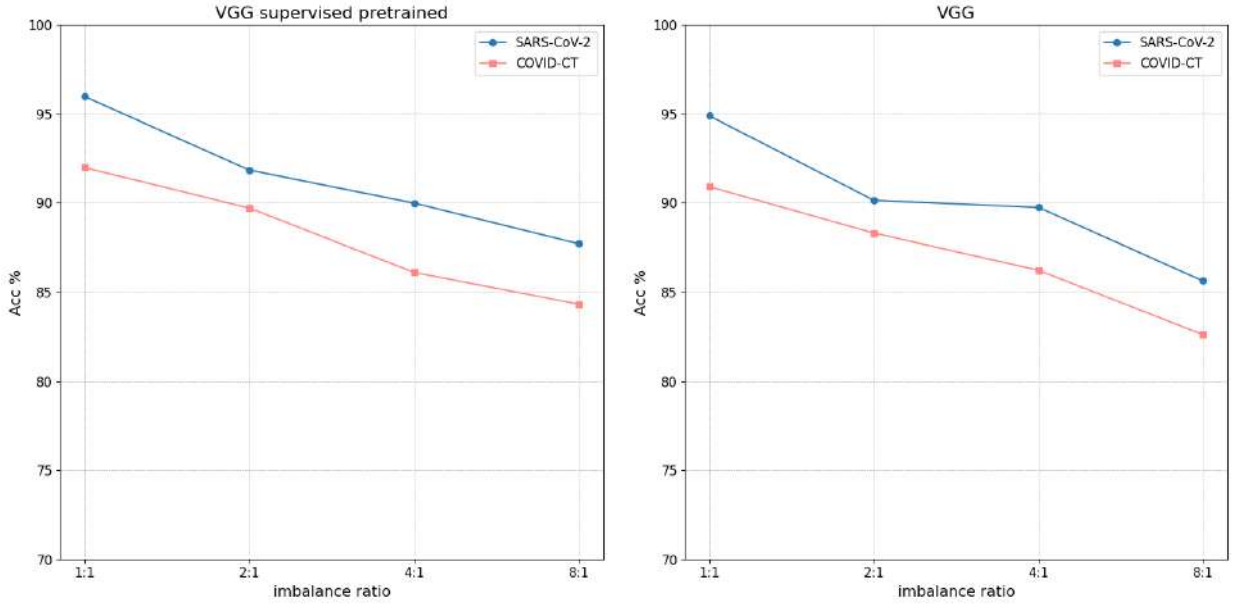

(b)




(c)




(d)




(e) (f)

Figure 4.8. Experimental results of CNN models on imbalanced datasets. The blue line signifies the experimental results from the SARS-COV-2 dataset. The orange line depicts the experimental results derived from the COVID-CT dataset. (a) Pre-trained ResNet 121, (b) ResNet 121, (c) Pre-trained EfficientNet b0, (d) EfficientNet b0, (e) Pre-trained VGG 16, (f) VGG 16.

Models trained from scratch: These models did not undergo any pre-training but were directly trained under full supervision on the imbalanced dataset.

This experimental design is intended to elucidate the potential strengths and weaknesses inherent to both self-supervised and fully supervised pre-training approaches when managing imbalanced datasets.

Compared to traditional CNN architectures, the Vision Transformer (ViT), when trained from scratch, experiences a more pronounced accuracy drop with highly imbalanced data. The main reason for this performance decline is that ViT's design philosophy emphasizes capturing global context information in images, showing

clear advantages in scenarios where global dependencies are strong and context parsing is crucial [91]. However, capturing this global context comes at a cost. ViT typically requires a large amount of training data to realize its maximum potential and hopes to achieve equivalent or higher performance levels compared to classical CNN frameworks [92].

Comparison of Figures 4.8 and 4.7 reveals that, with a data balance ratio of 1:1, the accuracy of the CNN model pre-trained on ImageNet marginally surpasses that of the self-supervised method examined in this study. The reason for this difference lies in the vast disparity in training data volumes: due to computational constraints, the self-supervised method in this paper was only pretrained on 60,000 medical images, while in comparison, ImageNet's training set contains 1,281,167 images [54], a massive difference in data scale. Encouragingly, when the data imbalance ratio expands to 8:1, we find that the accuracy obtained by the self-supervised pretraining method in this study surpasses that of the CNN models pretrained on ImageNet. This finding further confirms that our self-supervised pre-training model can still demonstrate satisfactory robustness and strong generalization capabilities when dealing with highly imbalanced data scenarios.

## 4.5 Conclusion of Chapter 4

This chapter examined the application of self-supervised pre-trained models for CT scan classification. Obtaining accurate labels for medical imaging data is challenging due to the expertise required. Medical imaging datasets also often exhibit

significant class imbalance. The chapter explored how self-supervised pre-training can learn representations from unlabeled CT scans to address these issues.

Three self-supervised learning methods - MAE, Mixup Feature, and Denoising Self-Distillation MAE - were applied to pre-train on the unlabeled COVID-CTset dataset. Reconstruction images on held-out data indicated the model captured global anatomical structure but struggled with fine tissue textures. However, reconstruction performance does not necessarily correlate with transfer learning ability.

The pre-trained models were fine-tuned on two smaller labeled datasets for classification. All self-supervised methods outperformed direct training, demonstrating enhanced performance through self-supervised pre-training. Mixup Feature achieved results comparable to MAE, highlighting its potential in medical domains.

To evaluate robustness under real-world class imbalances, models were tested with varying positive and negative sample ratios. Self-supervised pre-training stabilized performance as imbalance increased, whereas direct training showed steeper declines. This suggests self-supervised learning provides a more robust feature representation foundation.

Comparison to CNNs pre-trained on ImageNet revealed self-supervised learning maintained higher accuracy even at extreme 8:1 imbalance. While supervised pre-training performed slightly better with balanced data, self-supervised learning exhibited stronger generalization to highly imbalanced scenarios.

In conclusion, this work demonstrated the promise of self-supervised pre-training for enhancing CT scan analysis and robust medical image classification.

Self-supervised models achieved superior and more stable performance compared to direct training, especially under realistic class imbalances. The Mixup Feature method displayed transferability competitive with state-of-the-art MAE. Overall, self-supervised learning represents a promising approach for medical imaging tasks involving limited labeled data.

# CONCLUSIONS

This dissertation delves into innovative methods of self-supervised learning algorithms within visual representation learning. Furthermore, it investigates its applications in the domain of medical image analysis. During the research process, a series of academically valuable and practically applicable results were achieved.

1. This research presents a comprehensive review of self-supervised learning algorithms, systematically organizing them into four core categories: contrastive learning, masked image modeling, self-distillation, and canonical correlation analysis. For each methodology, this study meticulously elucidates its features and strengths, paving a theoretical foundation for ensuing research and guiding the introduction of novel algorithms.

2. This dissertation presents two pioneering self-supervised learning algorithms centered on masked image modeling, with an emphasis on enhancing the efficacy of visual feature capture. The Mixup Feature strategy introduces an innovative predictive task, targeting the reconstruction of a fusion of traditional image features like Sobel, HOG, and LBP. This sets forth a formidable challenge for the masked autoencoder. This approach not only elevates the model's predictive complexity but also enriches the information pool for visual representation. Conversely, the Denoising Self-Distillation Masked Autoencoder method adeptly integrates self-distillation with the masked autoencoder. The loss design takes into account both pixel-level

image restoration and feature-level regression, striking a balance between detailed and high-level semantic information recovery.

3. Extensive experimental evaluations were conducted on the proposed self-supervised learning algorithms, testing their performance across multiple standard datasets. Overall, compared to the current state-of-the-art methods, these two new algorithms demonstrated significant advantages on three distinct datasets. Especially the Mixup Feature and the Denoising Self-Distillation Masked Autoencoder, their performances underscored their innovativeness in the self-supervised learning domain. For further research on these two methods, their effects on larger datasets can be explored.

4. In the field of medical imaging, this paper particularly focuses on the classification problem of CT scans with a lack of extensive labeled data. Experimental results indicate that, compared to traditional methods, utilizing self-supervised pre-training can significantly improve model classification performance, especially in imbalanced data scenarios. Notably, Mixup Feature achieved exceptional results comparable to MAE, further emphasizing its tremendous potential in medical image classification.

In summation, this study, anchored by two innovative algorithms and substantiated by empirical validations on both visual and medical imaging datasets, has enriched the realm of self-supervised learning. Indeed, the proposed methods are competitive with the state-of-the-art and showcase novel innovations. Evaluations under realistic class imbalances have proven the robustness of self-supervised pre-

training for healthcare applications involving limited labeled data. Overall, this paper highlights the prospects of self-supervised learning as an effective approach for medical image classification, offering new methods and models that provide valuable insights and guidance for future computer vision research. Future work can explore extending these methods to larger datasets and a wider range of medical imaging modalities.

# REFERENCES


1. Balestriero, R., Ibrahim, M., Sobal, V., Morcos, A., Shekhar, S., Goldstein, T., Bordes, F., Bardes, A., Mialon, G., Tian, Y. and Schwarzschild, A., 2023. A cookbook of self-supervised learning. arXiv preprint arXiv:2304.12210.
2. Liu, Y., Han, T., Ma, S., Zhang, J., Yang, Y., Tian, J., He, H., Li, A., He, M., Liu, Z. and Wu, Z., 2023. Summary of chatgpt/gpt-4 research and perspective towards the future of large language models. arXiv preprint arXiv:2304.01852.
3. Chowdhery, Aakanksha, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham et al. "Palm: Scaling language modeling with pathways." arXiv preprint arXiv:2204.02311 (2022).
4. Goyal, Priya, Mathilde Caron, Benjamin Lefaudeux, Min Xu, Pengchao Wang, Vivek Pai, Mannat Singh et al. "Self-supervised pretraining of visual features in the wild." arXiv preprint arXiv:2103.01988 (2021).
5. Kirillov, A., Mintun, E., Ravi, N., Mao, H., Rolland, C., Gustafson, L., Xiao, T., Whitehead, S., Berg, A.C., Lo, W.Y. and Dollár, P., 2023. Segment anything. arXiv preprint arXiv:2304.02643.
6. Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., ... & Houlsby, N. (2020). An image is worth 16x16 words: Transformers for image recognition at scale. arXiv preprint arXiv:2010.11929.
7. Zhang, R., Isola, P. and Efros, A.A., 2016. Colorful image colorization. In Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The

Netherlands, October 11-14, 2016, Proceedings, Part III 14 (pp. 649-666). Springer International Publishing.

8. Larsson, G., Maire, M. and Shakhnarovich, G., 2016. Learning representations for automatic colorization. In Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV 14 (pp. 577-593). Springer International Publishing.
9. Pathak, D., Krahenbuhl, P., Donahue, J., Darrell, T. and Efros, A.A., 2016. Context encoders: Feature learning by inpainting. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 2536-2544).
10. Gidaris, S., Singh, P. and Komodakis, N., 2018. Unsupervised representation learning by predicting image rotations. arXiv preprint arXiv:1803.07728.
11. Doersch, C., Gupta, A. and Efros, A.A., 2015. Unsupervised visual representation learning by context prediction. In Proceedings of the IEEE international conference on computer vision (pp. 1422-1430).
12. Caron, Mathilde, Piotr Bojanowski, Armand Joulin, and Matthijs Douze. "Deep clustering for unsupervised learning of visual features." In Proceedings of the European conference on computer vision (ECCV), pp. 132-149. 2018.
13. Vincent, Pascal, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. "Extracting and composing robust features with denoising autoencoders." In Proceedings of the 25th international conference on Machine learning, pp. 1096-1103. 2008.

14. Wang, W., Arora, R., Livescu, K. and Bilmes, J., 2015, June. On deep multi-view representation learning. In International conference on machine learning (pp. 1083-1092). PMLR.
15. Zhang, Richard, Phillip Isola, and Alexei A. Efros. "Split-brain autoencoders: Unsupervised learning by cross-channel prediction." In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 1058-1067. 2017.
16. Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. "Generative adversarial nets." Advances in neural information processing systems 27 (2014).
17. Salimans, Tim, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. "Improved techniques for training gans." Advances in neural information processing systems 29 (2016).
18. Bachman, Philip, R. Devon Hjelm, and William Buchwalter. "Learning representations by maximizing mutual information across views." Advances in neural information processing systems 32 (2019).
19. Bromley, J., Guyon, I., LeCun, Y., Säckinger, E. and Shah, R., 1993. Signature verification using a" siamese" time delay neural network. Advances in neural information processing systems, 6.
20. Hadsell, R., Chopra, S. and LeCun, Y., 2006, June. Dimensionality reduction by learning an invariant mapping. In 2006 IEEE computer society conference on computer vision and pattern recognition (CVPR'06) (Vol. 2, pp. 1735-1742). IEEE.

21. Weinberger, Kilian Q., and Lawrence K. Saul. "Distance metric learning for large margin nearest neighbor classification." Journal of machine learning research 10, no. 2 (2009).
22. Sohn, Kihyuk. "Improved deep metric learning with multi-class n-pair loss objective." Advances in neural information processing systems 29 (2016).
23. Oord, A.V.D., Li, Y. and Vinyals, O., 2018. Representation learning with contrastive predictive coding. arXiv preprint arXiv:1807.03748.
24. Henaff, O., 2020, November. Data-efficient image recognition with contrastive predictive coding. In International conference on machine learning (pp. 4182-4192). PMLR.
25. Oh Song, H., Xiang, Y., Jegelka, S. and Savarese, S., 2016. Deep metric learning via lifted structured feature embedding. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 4004-4012).
26. Ke Ji Meng Shou. " Self-Supervised Learning super detailed interpretation (2): SimCLR series" ZhiHu, 2021.06.17, https://zhuanlan.zhihu.com/p/378953015.
27. Chen, Ting, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. "A simple framework for contrastive learning of visual representations." In International conference on machine learning, pp. 1597-1607. PMLR, 2020.
28. Nair, Vinod, and Geoffrey E. Hinton. "Rectified linear units improve restricted boltzmann machines." In Proceedings of the 27th international conference on machine learning (ICML-10), pp. 807-814. 2010.

29. Wu, Z., Xiong, Y., Yu, S. X., & Lin, D. (2018). Unsupervised feature learning via non-parametric instance discrimination. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 3733-3742).

30. He, K., Fan, H., Wu, Y., Xie, S. and Girshick, R., 2020. Momentum contrast for unsupervised visual representation learning. In Proceedings of the IEEE/CVF conference on computer vision and pattern recognition (pp. 9729-9738).

31. Devlin, J., Chang, M.W., Lee, K. and Toutanova, K., 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.

32. Bao, Hangbo, Li Dong, Songhao Piao, and Furu Wei. "Beit: Bert pre-training of image transformers." arXiv preprint arXiv:2106.08254 (2021).

33. He, Kaiming, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. "Masked autoencoders are scalable vision learners." In Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, pp. 16000-16009. 2022.

34. Xie, Zhenda, Zheng Zhang, Yue Cao, Yutong Lin, Jianmin Bao, Zhuliang Yao, Qi Dai, and Han Hu. "Simmim: A simple framework for masked image modeling." In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 9653-9663. 2022.

35. Oquab, Maxime, et al. "Dinov2: Learning robust visual features without supervision." arXiv preprint arXiv:2304.07193 (2023).

36. Woo, Sanghyun, et al. "Convnext v2: Co-designing and scaling convnets with masked autoencoders." Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2023.
37. Chang, Huiwen, et al. "Muse: Text-to-image generation via masked generative transformers." arXiv preprint arXiv:2301.00704 (2023).
38. Grill, Jean-Bastien, et al. "Bootstrap your own latent-a new approach to self-supervised learning." Advances in neural information processing systems 33 (2020): 21271-21284.
39. Chen, Xinlei, and Kaiming He. "Exploring simple siamese representation learning." Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 2021.
40. Caron, Mathilde, et al. "Emerging properties in self-supervised vision transformers." Proceedings of the IEEE/CVF international conference on computer vision. 2021.
41. He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.
42. Zhou, Jinghao, Chen Wei, Huiyu Wang, Wei Shen, Cihang Xie, Alan Yuille, and Tao Kong. "ibot: Image bert pre-training with online tokenizer." arXiv preprint arXiv:2111.07832 (2021).
43. Oquab, Maxime, Timothée Darcet, Théo Moutakanni, Huy Vo, Marc Szafraniec, Vasil Khalidov, Pierre Fernandez et al. "Dinov2: Learning robust visual features without supervision." arXiv preprint arXiv:2304.07193 (2023).

44. Bardes, Adrien, Jean Ponce, and Yann LeCun. "Vicreg: Variance-invariance-covariance regularization for self-supervised learning." arXiv preprint arXiv:2105.04906 (2021).
45. Zbontar, Jure, Li Jing, Ishan Misra, Yann LeCun, and Stéphane Deny. "Barlow twins: Self-supervised learning via redundancy reduction." In International Conference on Machine Learning, pp. 12310-12320. PMLR, 2021.
46. Caron, M., Misra, I., Mairal, J., Goyal, P., Bojanowski, P. and Joulin, A., 2020. Unsupervised learning of visual features by contrasting cluster assignments. Advances in neural information processing systems, 33, pp.9912-9924.
47. Ermolov, Aleksandr, Aliaksandr Siarohin, Enver Sangineto, and Nicu Sebe. "Whitening for self-supervised representation learning." In International Conference on Machine Learning, pp. 3015-3024. PMLR, 2021.
48. N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05), San Diego, CA, USA, 2005, pp. 886-893 vol. 1, doi: 10.1109/CVPR.2005.177.
49. T. Ahonen, A. Hadid and M. Pietikainen, "Face Description with Local Binary Patterns: Application to Face Recognition," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 28, no. 12, pp. 2037-2041, Dec. 2006, doi: 10.1109/TPAMI.2006.244.
50. N. Kanopoulos, N. Vasanthavada and R. L. Baker, "Design of an image edge detection filter using the Sobel operator," in IEEE Journal of Solid-State Circuits, vol. 23, no. 2, pp. 358-367, April 1988, doi: 10.1109/4.996.

51. Ahonen, T., Hadid, A. and Pietikinen, M. (2006) Face Description with Local Binary Patterns: Application to Face Recogniton. IEEE Transactions on Pattern Analysis and Machine Intelligence, 28, 2037-2041.
52. Ojala, T., Pietikinen, M. and Menp, T. (2002) Multiresolution Gray Scale and Rotation Invariant Texture Classification with Local Binary Patterns. IEEE Transactions on Pattern Analysis and Machine Intelligence, 24, 971-998.
53. R. C. Gonzalez; R. E. Woods, Digital Image Processing, Prentice Hall, Upper Saddle River, NJ., 2002. ISBN 013168728X.
54. J. Deng, W. Dong, R. Socher, L. -J. Li, Kai Li and Li Fei-Fei, "ImageNet: A large-scale hierarchical image database," 2009 IEEE Conference on Computer Vision and Pattern Recognition, Miami, FL, USA, 2009, pp. 248-255, doi: 10.1109/CVPR.2009.5206848.
55. Asano, Y. M., Rupprecht, C., Zisserman, A., & Vedaldi, A. (2021). Pass: An imagenet replacement for self-supervised pretraining without humans. arXiv preprint arXiv:2109.13228.
56. R. C. Gonzalez and R. E. Woods, Digital Image Processing. Upper Saddle River, NJ, USA: Prentice-Hall, 2002.
57. A. Krizhevsky, ‘‘Learning multiple layers of features from tiny images,’’ Tech. Rep., 2009.
58. T. Cover and P. Hart, "Nearest neighbor pattern classification," in IEEE Transactions on Information Theory, vol. 13, no. 1, pp. 21-27, January 1967, doi: 10.1109/TIT.1967.1053964.

59. Bordes, F., Balestriero, R. and Vincent, P., 2023. Towards Democratizing Joint-Embedding Self-Supervised Learning. arXiv preprint arXiv:2303.01986.

60. Dong, Xiaoyi, Jianmin Bao, Ting Zhang, Dongdong Chen, Weiming Zhang, Lu Yuan, Dong Chen, Fang Wen, Nenghai Yu, and Baining Guo. "Peco: Perceptual codebook for bert pre-training of vision transformers." In Proceedings of the AAAI Conference on Artificial Intelligence, vol. 37, no. 1, pp. 552-560. 2023.

61. Wei, Y., Hu, H., Xie, Z., Zhang, Z., Cao, Y., Bao, J., Chen, D. and Guo, B., 2022. Contrastive learning rivals masked image modeling in fine-tuning via feature distillation. arXiv preprint arXiv:2205.14141.

62. Bordes, Florian, Randall Balestriero, and Pascal Vincent. "High fidelity visualization of what your self-supervised representation knows about." arXiv preprint arXiv:2112.09164 (2021).

63. I. Loshchilov and F. Hutter, ‘‘Decoupled weight decay regularization,’’ 2017, arXiv:1711.05101.

64. I. Loshchilov and F. Hutter, ‘‘SGDR: Stochastic gradient descent with warm restarts,’’ 2016, arXiv:1608.03983.

65. P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, ‘‘Accurate, large minibatch SGD: Training ImageNet in 1 hour,’’ 2017, arXiv:1706.02677.

66. Ma, Jerry, and Denis Yarats. "On the adequacy of untuned warmup for adaptive optimization." In Proceedings of the AAAI Conference on Artificial Intelligence, vol. 35, no. 10, pp. 8828-8836. 2021.

67. J. Xu and S. Stirenko, "Mixup Feature: A Pretext Task Self-Supervised Learning Method for Enhanced Visual Feature Learning," in IEEE Access, vol. 11, pp. 82400-82409, 2023, doi: 10.1109/ACCESS.2023.3301561.

68. Wei, Chen, Haoqi Fan, Saining Xie, Chao-Yuan Wu, Alan Yuille, and Christoph Feichtenhofer. "Masked feature prediction for self-supervised visual pre-training." In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 14668-14678. 2022.

69. K. He, X. Zhang, S. Ren, and J. Sun, ‘‘Deep residual learning for image recognition,’’ in Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR), Jun. 2016, pp. 770–778.

70. E. D. Cubuk, B. Zoph, J. Shlens, and Q. V. Le, ‘‘RandAugment: Practical automated data augmentation with a reduced search space,’’ in Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. Workshops (CVPRW), Jun. 2020, pp. 3008–3017.

71. Fan, Haoqi, Bo Xiong, Karttikeya Mangalam, Yanghao Li, Zhicheng Yan, Jitendra Malik, and Christoph Feichtenhofer. "Multiscale vision transformers." In Proceedings of the IEEE/CVF international conference on computer vision, pp. 6824-6835.

72. Chen, X., Ding, M., Wang, X., Xin, Y., Mo, S., Wang, Y., ... & Wang, J. (2022). Context autoencoder for self-supervised representation learning. arXiv preprint arXiv:2202.03026.

73. Chen, Yabo, Yuchen Liu, Dongsheng Jiang, Xiaopeng Zhang, Wenrui Dai, Hongkai Xiong, and Qi Tian. "Sdae: Self-distillated masked autoencoder." In

European Conference on Computer Vision, pp. 108-124. Cham: Springer Nature Switzerland, 2022.

74. Nguyen, D., Kay, F., Tan, J., Yan, Y., Ng, Y. S., Iyengar, P., ... & Jiang, S. (2021). Deep learning–based COVID-19 pneumonia classification using chest CT images: model generalizability. Frontiers in Artificial Intelligence, 4.

75. Zhao J, Zhang Y, He X, Xie P. Covid-ct-dataset: a ct scan dataset about covid-19. arXiv preprint arXiv:2003.13865. 2020 Jun;490.

76. Angelov, Plamen, and Eduardo Almeida Soares. "SARS-CoV-2 CT-scan dataset: A large dataset of real patients CT scans for SARS-CoV-2 identification." MedRxiv (2020).

77. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... & Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Advances in Neural Information Processing Systems 32 (pp. 8024-8035).

78. Hinton, G. E., & Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. science, 313(5786), 504-507.

79. Zhou, Zongwei, et al. "Models genesis: Generic autodidactic models for 3d medical image analysis." Medical Image Computing and Computer Assisted Intervention–MICCAI 2019: 22nd International Conference, Shenzhen, China, October 13–17, 2019, Proceedings, Part IV 22. Springer International Publishing, 2019.

80. Liu, Fengbei, Yu Tian, Filipe R. Cordeiro, Vasileios Belagiannis, Ian Reid, and Gustavo Carneiro. "Self-supervised mean teacher for semi-supervised chest x-ray classification." In International Workshop on Machine Learning in Medical Imaging, pp. 426-436. Cham: Springer International Publishing, 2021.

81. Xu, Jiashu. "A review of self-supervised learning methods in the field of medical image analysis." International Journal of Image, Graphics and Signal Processing (IJIGSP) 13, no. 4 (2021): 33-46.

82. Liu, Hong, Jeff Z. HaoChen, Adrien Gaidon, and Tengyu Ma. "Self-supervised learning is more robust to dataset imbalance." arXiv preprint arXiv:2110.05025 (2021).

83. Cyril, Goutte., Eric, Gaussier. (2005). A probabilistic interpretation of precision, recall and F -score, with implication for evaluation.

84. C. X. Ling, J. Huang, and H. Zhang, "AUC: A Statistically Consistent and More Discriminating Measure than Accuracy," in Proceedings of the 18th International Joint Conference on Artificial Intelligence, Acapulco, Mexico, 2003, pp. 519-524.

85. Jiashu Xu, Sergii Stirenko, "Denoising Self-Distillation Masked Autoencoder for Self-Supervised Learning", International Journal of Image, Graphics and Signal Processing (IJIGSP), Vol.15, No.5, pp. 29-38, 2023.

86. Tan, Mingxing, and Quoc Le. "Efficientnet: Rethinking model scaling for convolutional neural networks." In International conference on machine learning, pp. 6105-6114. PMLR, 2019.

87. Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

88. Marques, Gonçalo, Deevyankar Agarwal, and Isabel De la Torre Díez. "Automated medical diagnosis of COVID-19 through EfficientNet convolutional neural network." Applied soft computing 96 (2020): 106691.

89. Yang, Wei, Huijuan Zhang, Jian Yang, Jiasong Wu, Xiangrui Yin, Yang Chen, Huazhong Shu et al. "Improving low-dose CT image using residual convolutional network." Ieee Access 5 (2017): 24698-24705.

90. Tan, Wenjun, Pan Liu, Xiaoshuo Li, Yao Liu, Qinghua Zhou, Chao Chen, Zhaoxuan Gong, Xiaoxia Yin, and Yanchun Zhang. "Classification of COVID-19 pneumonia from chest CT images based on reconstructed super-resolution images and VGG neural network." Health Information Science and Systems 9 (2021): 1-12.

91. Zhou, Hong-Yu, Chixiang Lu, Sibei Yang, and Yizhou Yu. "Convnets vs. transformers: Whose visual representations are more transferable?." In Proceedings of the IEEE/CVF International Conference on Computer Vision, pp. 2230-2238. 2021.

92. Bai, Yutong, Jieru Mei, Alan L. Yuille, and Cihang Xie. "Are transformers more robust than cnns?." Advances in neural information processing systems 34 (2021): 26831-26843.

93. Wu, Chenfei, Shengming Yin, Weizhen Qi, Xiaodong Wang, Zecheng Tang, and Nan Duan. "Visual chatgpt: Talking, drawing and editing with visual foundation models." arXiv preprint arXiv:2303.04671 (2023).

94. Liu, Yen-Cheng, Chih-Yao Ma, Zijian He, Chia-Wen Kuo, Kan Chen, Peizhao Zhang, Bichen Wu, Zsolt Kira, and Peter Vajda. "Unbiased teacher for semi-supervised object detection." arXiv preprint arXiv:2102.09480 (2021).

# APPENDIX A

## Part of the program code

### *Chapter 3 Experimental Part Code*

*Mixup_Model.py*

```
import torch
import timm
import numpy as np
from skimage.feature import hog
from einops import repeat, rearrange
from einops.layers.torch import Rearrange

from timm.models.layers import trunc_normal_
from timm.models.vision_transformer import Block
import torch.nn.functional as F

def random_indexes(size : int):
    forward_indexes = np.arange(size)
    np.random.shuffle(forward_indexes)
    backward_indexes = np.argsort(forward_indexes)
    return forward_indexes, backward_indexes

def take_indexes(sequences, indexes):
    return torch.gather(sequences, 0, repeat(indexes, 't b -> t b c', c=sequences.shape[-1]))

class PatchShuffle(torch.nn.Module):
    def __init__(self, ratio) -> None:
        super().__init__()
        self.ratio = ratio

    def forward(self, patches : torch.Tensor):
        T, B, C = patches.shape
        remain_T = int(T * (1 - self.ratio))

        indexes = [random_indexes(T) for _ in range(B)]
        forward_indexes = torch.as_tensor(np.stack([i[0] for i in indexes], axis=-1),
dtype=torch.long).to(patches.device)
        backward_indexes = torch.as_tensor(np.stack([i[1] for i in indexes], axis=-1),
dtype=torch.long).to(patches.device)

        patches = take_indexes(patches, forward_indexes)
        patches = patches[:remain_T]

        return patches, forward_indexes, backward_indexes
```

```
class MAE_Encoder(torch.nn.Module):
    def __init__(self,
                 image_size=32,
                 patch_size=2,
                 emb_dim=192,
                 num_layer=12,
                 num_head=3,
                 mask_ratio=0.55,
                 ) -> None:
        super().__init__()

        self.cls_token = torch.nn.Parameter(torch.zeros(1, 1, emb_dim))
        self.pos_embedding = torch.nn.Parameter(torch.zeros((image_size // patch_size) ** 2, 1,
emb_dim))
        self.shuffle = PatchShuffle(mask_ratio)

        self.patchify = torch.nn.Conv2d(3, emb_dim, patch_size, patch_size)

        self.transformer = torch.nn.Sequential(*[Block(emb_dim, num_head) for _ in range(num_layer)])

        self.layer_norm = torch.nn.LayerNorm(emb_dim)

        self.init_weight()

    def init_weight(self):
        trunc_normal_(self.cls_token, std=.02)
        trunc_normal_(self.pos_embedding, std=.02)

    def forward(self, img):
        patches = self.patchify(img)
        patches = rearrange(patches, 'b c h w -> (h w) b c')
        patches = patches + self.pos_embedding

        patches, forward_indexes, backward_indexes = self.shuffle(patches)

        patches = torch.cat([self.cls_token.expand(-1, patches.shape[1], -1), patches], dim=0)
        patches = rearrange(patches, 't b c -> b t c')
        features = self.layer_norm(self.transformer(patches))
        features = rearrange(features, 'b t c -> t b c')

        return features, backward_indexes

class MAE_Decoder(torch.nn.Module):
    def __init__(self,
                 image_size=32,
```

```
                patch_size=2,
                emb_dim=192,
                num_layer=4,
                num_head=3,
                ) -> None:
        super().__init__()

        self.mask_token = torch.nn.Parameter(torch.zeros(1, 1, emb_dim))
        self.pos_embedding = torch.nn.Parameter(torch.zeros((image_size // patch_size) ** 2 + 1, 1,
emb_dim))

        self.transformer = torch.nn.Sequential(*[Block(emb_dim, num_head) for _ in range(num_layer)])

        self.head = torch.nn.Linear(emb_dim, 1 * patch_size ** 2)
        self.patch2img = Rearrange('(h w) b (c p1 p2) -> b c (h p1) (w p2)', p1=patch_size,
p2=patch_size, h=image_size//patch_size)

        self.init_weight()

    def init_weight(self):
        trunc_normal_(self.mask_token, std=.02)
        trunc_normal_(self.pos_embedding, std=.02)

    def forward(self, features, backward_indexes):
        T = features.shape[0]
        backward_indexes = torch.cat([torch.zeros(1, backward_indexes.shape[1]).to(backward_indexes),
backward_indexes + 1], dim=0)
        features = torch.cat([features, self.mask_token.expand(backward_indexes.shape[0] -
features.shape[0], features.shape[1], -1)], dim=0)
        features = take_indexes(features, backward_indexes)
        features = features + self.pos_embedding

        features = rearrange(features, 't b c -> b t c')
        features = self.transformer(features)
        features = rearrange(features, 'b t c -> t b c')
        features = features[1:] # remove global feature

        patches = self.head(features)
        mask = torch.zeros_like(patches)
        mask[T:] = 1
        mask = take_indexes(mask, backward_indexes[1:] - 1)
        img = self.patch2img(patches)
        mask = self.patch2img(mask)

        return img, mask
```

```
class MAE_ViT(torch.nn.Module):
    def __init__(self,
                 image_size=32,
                 patch_size=2,
                 emb_dim=384,
                 encoder_layer=12,
                 encoder_head=8,
                 decoder_layer=4,
                 decoder_head=8,
                 mask_ratio=0.55,
                 ) -> None:
        super().__init__()

        self.encoder = MAE_Encoder(image_size, patch_size, emb_dim, encoder_layer, encoder_head,
mask_ratio)
        self.decoder = MAE_Decoder(image_size, patch_size, emb_dim, decoder_layer, decoder_head)

    def forward(self, img):
        features, backward_indexes = self.encoder(img)
        predicted_img, mask = self.decoder(features,  backward_indexes)
        return predicted_img, mask

class ViT_Classifier(torch.nn.Module):
    def __init__(self, encoder : MAE_Encoder, num_classes=10) -> None:
        super().__init__()
        self.cls_token = encoder.cls_token
        self.pos_embedding = encoder.pos_embedding
        self.patchify = encoder.patchify
        self.transformer = encoder.transformer
        self.layer_norm = encoder.layer_norm
        self.head = torch.nn.Linear(self.pos_embedding.shape[-1], num_classes)

    def forward(self, img):
        patches = self.patchify(img)
        patches = rearrange(patches, 'b c h w -> (h w) b c')
        patches = patches + self.pos_embedding
        patches = torch.cat([self.cls_token.expand(-1, patches.shape[1], -1), patches], dim=0)
        patches = rearrange(patches, 't b c -> b t c')
        features = self.layer_norm(self.transformer(patches))
        features = rearrange(features, 'b t c -> t b c')
        logits = self.head(features[0])
        return logits


def calculate_hog(batch_images):
    hog_images = []
```

```
    for image in batch_images:
        hog_feature,hog_image= hog(image.to("cpu"), orientations=9, pixels_per_cell=(8, 8),
cells_per_block=(4, 4), visualize=True)
        hog_images.append(hog_image)
    return torch.tensor(np.array(hog_images))

def mixup_sobel_hog(imgs,mixup=True,lamda = 0):
  # 定义Sobel卷积核
  sobel_kernel_x = torch.Tensor([[1, 0, -1], [2, 0, -2], [1, 0, -1]]).unsqueeze(0).unsqueeze(0)
  sobel_kernel_y = torch.Tensor([[1, 2, 1], [0, 0, 0], [-1, -2, -1]]).unsqueeze(0).unsqueeze(0)
  # 将图像数据转换为灰度图像
  imgs_gray = 0.299 * imgs[:, 0, :, :] + 0.587 * imgs[:, 1, :, :] + 0.114 * imgs[:, 2, :, :]
  imgs_gray_2 = imgs.permute(0,2,3,1)
  imgs_gray_2 =  0.299 * imgs_gray_2[:, :, :, 0] + 0.587 * imgs_gray_2[:, :, :, 1] + 0.114 *
imgs_gray_2[:, :, :, 2]
  hog_images = calculate_hog(imgs_gray_2)
  hog_images = torch.unsqueeze(hog_images,1)
  # 将灰度图像进行卷积操作，获取x和y方向上的边缘强度
  imgs_gray = imgs_gray.unsqueeze(1)
  sobel_kernel_x = sobel_kernel_x.to(imgs_gray.device)
  sobel_kernel_y = sobel_kernel_y.to(imgs_gray.device)
  edge_x = F.conv2d(imgs_gray, sobel_kernel_x, padding=1)
  edge_y = F.conv2d(imgs_gray, sobel_kernel_y, padding=1)
  # 计算边缘强度
  edge_strength = torch.sqrt(torch.pow(edge_x, 2) + torch.pow(edge_y, 2))

  #对边缘强度进行归一化
  # edge_strength /= edge_strength.max()
  if mixup:
    mixup_feature = lamda*edge_strength + (1-lamda)*hog_images.to(imgs_gray.device)
  else:
    mixup_feature = edge_strength + hog_images.to(imgs_gray.device)

  return mixup_feature




if __name__ == '__main__':
    shuffle = PatchShuffle(0.75)
    a = torch.rand(16, 2, 10)
    b, forward_indexes, backward_indexes = shuffle(a)
    img = torch.rand(4, 3, 32, 32)
    sobel_img = mixup_sobel_hog(img)
    print("mixup shape",sobel_img.shape)
    encoder = MAE_Encoder()
```

```
    decoder = MAE_Decoder()
    features, backward_indexes = encoder(img)
    print(forward_indexes.shape)
    predicted_img, mask = decoder(features, backward_indexes)
    print(predicted_img.shape)
    #loss = torch.mean((predicted_img - sobel_img) ** 2 * mask) / 0.75
    loss = torch.mean((predicted_img - sobel_img) ** 2)
    print(loss)
```

*Mixup_pretrained.py*

```
import os
import argparse
import math
import torch
import torchvision
from torch.utils.tensorboard import SummaryWriter
from torchvision.transforms import ToTensor, Compose, Normalize
from tqdm import tqdm
from torchvision import utils as vutils
import dataset_process
from model import *
from utils import setup_seed

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--seed', type=int, default=30)
    parser.add_argument('--patch_size', type=int, default=2)
    parser.add_argument('--batch_size', type=int, default=1024)
    parser.add_argument('--max_device_batch_size', type=int, default=128)
    parser.add_argument('--base_learning_rate', type=float, default=1.5e-4)
    parser.add_argument('--weight_decay', type=float, default=0.05)
    parser.add_argument('--mask_ratio', type=float, default=0.65)
    parser.add_argument('--total_epoch', type=int, default=1200)
    parser.add_argument('--warmup_epoch', type=int, default=50)
    parser.add_argument('--model_path', type=str, default='')
    parser.add_argument('--save_path', type=str, default='')
    parser.add_argument('--data_name', type=str, default='cifar10')
    parser.add_argument('--image_size', type=int, default=32)
    parser.add_argument('--lamda', type=float, default=0.5)
    parser.add_argument('--vit_type', type=str, default="vit-tiny")
    args = parser.parse_args()
    setup_seed(args.seed)

    batch_size = args.batch_size
    load_batch_size = min(args.max_device_batch_size, batch_size)
```

```
    assert batch_size % load_batch_size == 0
    steps_per_update = batch_size // load_batch_size
    if args.data_name == 'cifar10':
        train_dataset = torchvision.datasets.CIFAR10('data', train=True, download=True,
                                                     transform=Compose([ToTensor(), Normalize(0.5,
0.5)]))
        val_dataset = torchvision.datasets.CIFAR10('data', train=False, download=True,
                                                   transform=Compose([ToTensor(), Normalize(0.5,
0.5)]))
    elif args.data_name == 'cifar100':
        train_dataset = torchvision.datasets.CIFAR100('data', train=True, download=True,
                                                     transform=Compose([ToTensor(), Normalize(0.5,
0.5)]))
        val_dataset = torchvision.datasets.CIFAR100('data', train=False, download=True,
                                                   transform=Compose([ToTensor(), Normalize(0.5,
0.5)]))
    elif args.data_name == 'stl10':
        train_dataset = torchvision.datasets.STL10('data', transform=Compose([ToTensor(),
Normalize(0.5, 0.5)]), split='train', download=True)
        val_dataset = torchvision.datasets.STL10('data', split='test', download=True,
transform=Compose([ToTensor(), Normalize(0.5, 0.5)]))
    dataloader = torch.utils.data.DataLoader(train_dataset, load_batch_size, shuffle=True,
num_workers=4)

    #dataloader,num_class,val_dataset  = load_data(args.data_dir, args.data_name,True,
args.image_size, load_batch_size, 2)
    writer = SummaryWriter(args.save_path + os.path.join('logs',args.data_name,
'mixup_feature_pretrain'))
    device = 'cuda' if torch.cuda.is_available() else 'cpu'
    if args.vit_type == "vit_tiny":
        model = MAE_ViT(mask_ratio=args.mask_ratio,
                        image_size = args.image_size,
                        patch_size=args.patch_size).to(device)
    elif args.vit_type == "vit_base":
        model = MAE_ViT(mask_ratio=args.mask_ratio,
                        image_size=args.image_size,
                        patch_size=args.patch_size,
                        emb_dim = 768,
                        encoder_layer = 12,
                        encoder_head = 12,
                        decoder_layer = 6,
                        decoder_head = 12,).to(device)

    optim = torch.optim.AdamW(model.parameters(), lr=args.base_learning_rate * args.batch_size / 256,
betas=(0.9, 0.95),
                              weight_decay=args.weight_decay)
```

```
lr_func = lambda epoch: min((epoch + 1) / (args.warmup_epoch + 1e-8),
                            0.5 * (math.cos(epoch / args.total_epoch * math.pi) + 1))
lr_scheduler = torch.optim.lr_scheduler.LambdaLR(optim, lr_lambda=lr_func, verbose=True)

step_count = 0
optim.zero_grad()


if os.path.exists(args.save_path + 'checkpoint.pth'):

    checkpoint = torch.load(args.save_path + 'checkpoint.pth')
    model.load_state_dict(checkpoint['model_state_dict'])
    optim.load_state_dict(checkpoint['optimizer_state_dict'])
    start_epoch = checkpoint['epoch']
    loss = checkpoint['loss']
    print("load previous model epoch {}".format(checkpoint['epoch']))
else:
    start_epoch = 0

for e in range(start_epoch, args.total_epoch):
    model.train()
    losses = []
    for img, label in tqdm(iter(dataloader)):
        step_count += 1
        img = img.to(device)
        mixup_img = mixup_sobel_hog(img,lamda =args.lamda)
        predicted_img, mask = model(img)
        #loss = torch.mean((predicted_img - mixup_img) ** 2 * mask) / args.mask_ratio
        loss = torch.mean((predicted_img - mixup_img) ** 2)
        loss.backward()
        if step_count % steps_per_update == 0:
            optim.step()
            optim.zero_grad()
        losses.append(loss.item())
    lr_scheduler.step()
    avg_loss = sum(losses) / len(losses)
    writer.add_scalar('mae_loss', avg_loss, global_step=e)
    print(f'In epoch {e}, average traning loss is {avg_loss}.')

    ''' visualize the first 16 predicted images on val dataset'''
    model.eval()
    with torch.no_grad():
        val_img = torch.stack([val_dataset[i][0] for i in range(48)])
        val_img = val_img.to(device)
        sobel_val_img = mixup_sobel_hog(val_img,lamda =args.lamda)
        predicted_val_img, mask = model(val_img)
```

```
            grid_4 = vutils.make_grid(predicted_val_img, nrow=8, padding=2, normalize=True)
            predicted_val_img = predicted_val_img * mask + sobel_val_img * (1 - mask)
            # img = torch.cat([sobel_val_img * (1 - mask), predicted_val_img, sobel_val_img], dim=0)
            # img = rearrange(img, '(v h1 w1) c h w -> c (h1 h) (w1 v w)', w1=2, v=3)
            # img_1 = torch.cat([val_img * (1 - mask),val_img,val_img * (1 - mask)], dim=0)
            # img_1 = rearrange(img_1, '(v h1 w1) c h w -> c (h1 h) (w1 v w)', w1=2, v=3)
            # writer.add_image('mae_image', (img + 1) / 2, global_step=e)
            # vutils.save_image(img.to(torch.device('cpu')), args.save_path + "gen_image.png")
            # vutils.save_image(img_1.to(torch.device('cpu')), args.save_path + "masked_image.png")

            grid = vutils.make_grid(val_img, nrow=8, padding=2, normalize=True)
            grid_1 = vutils.make_grid(sobel_val_img, nrow=8, padding=2, normalize=True)
            grid_2 = vutils.make_grid(predicted_val_img, nrow=8, padding=2, normalize=True)
            grid_3 = vutils.make_grid(val_img * (1 - mask), nrow=8, padding=2, normalize=True)

            writer.add_image('mask_image', grid_3, global_step=e)
            writer.add_image('pred_image', grid_2, global_step=e)
            writer.add_image('targ_image', grid_1, global_step=e)
            writer.add_image('orig_image', grid, global_step=e)
            vutils.save_image(grid.to(torch.device('cpu')), args.save_path + "real_image.png")
            vutils.save_image(grid_1.to(torch.device('cpu')), args.save_path + "target_image.png")
            vutils.save_image(grid_2.to(torch.device('cpu')), args.save_path + "pred_image.png")
            vutils.save_image(grid_3.to(torch.device('cpu')), args.save_path + "masked_image.png")
            vutils.save_image(grid_4.to(torch.device('cpu')), args.save_path + "pred_image_1.png")
        ''' save model '''

        torch.save(model, args.save_path + args.model_path)
        # save checkpoint

        checkpoint = {
            'epoch': e + 1,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optim.state_dict(),
            'loss': loss
        }
        torch.save(checkpoint, args.save_path + 'checkpoint.pth')
```

*Mixup_fineturing.py*

```
import os
import argparse
import math
import torch
import torchvision
from torch.utils.tensorboard import SummaryWriter
from torchvision.transforms import ToTensor, Compose, Normalize
from tqdm import tqdm
```

```
from model import *
from utils import setup_seed

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--seed', type=int, default=42)
    parser.add_argument('--batch_size', type=int, default=128)
    parser.add_argument('--max_device_batch_size', type=int, default=256)
    parser.add_argument('--base_learning_rate', type=float, default=1e-3)
    parser.add_argument('--weight_decay', type=float, default=0.05)
    parser.add_argument('--total_epoch', type=int, default=100)
    parser.add_argument('--warmup_epoch', type=int, default=5)
    parser.add_argument('--pretrained_model_path', type=str, default=None)
    parser.add_argument('--output_model_path', type=str, default='vit-t-classifier-from_pertain.pt')
    parser.add_argument('--save_path', type=str, default='./')
    parser.add_argument('--pretrain_type', type=str, default="fine_turn")
    parser.add_argument('--data_name', type=str, default='cifar10')
    args = parser.parse_args()

    setup_seed(args.seed)

    batch_size = args.batch_size
    load_batch_size = min(args.max_device_batch_size, batch_size)

    assert batch_size % load_batch_size == 0
    steps_per_update = batch_size // load_batch_size

    train_dataset = torchvision.datasets.CIFAR10('data', train=True, download=True,
                                                 transform=Compose([ToTensor(), Normalize(0.5,
0.5)]))
    val_dataset = torchvision.datasets.CIFAR10('data', train=False, download=True,
                                               transform=Compose([ToTensor(), Normalize(0.5, 0.5)]))
    train_dataloader = torch.utils.data.DataLoader(train_dataset, load_batch_size, shuffle=True,
num_workers=4)
    val_dataloader = torch.utils.data.DataLoader(val_dataset, load_batch_size, shuffle=False,
num_workers=4)
    device = 'cuda' if torch.cuda.is_available() else 'cpu'

    if args.pretrain == "fine_turn":
        model = torch.load(args.save_path + args.pretrained_model_path, map_location='cpu')
        writer = SummaryWriter(args.save_path + os.path.join('logs_s_b', args.data_name, 'mixup-
pretrain-cls'))
    else:
        model = MAE_ViT()
        writer = SummaryWriter(args.save_path + os.path.join('logs',args.data_name, 'scratch-cls'))
```

```
model = ViT_Classifier(model.encoder, num_classes=10).to(device)

loss_fn = torch.nn.CrossEntropyLoss()
acc_fn = lambda logit, label: torch.mean((logit.argmax(dim=-1) == label).float())

optim = torch.optim.AdamW(model.parameters(), lr=args.base_learning_rate * args.batch_size / 256,
                          betas=(0.9, 0.999), weight_decay=args.weight_decay)
lr_func = lambda epoch: min((epoch + 1) / (args.warmup_epoch + 1e-8),
                            0.5 * (math.cos(epoch / args.total_epoch * math.pi) + 1))
lr_scheduler = torch.optim.lr_scheduler.LambdaLR(optim, lr_lambda=lr_func, verbose=True)

best_val_acc = 0
step_count = 0
optim.zero_grad()
# 检查是否存在之前的检查点文件
if os.path.exists(args.save_path + 'PF_checkpoint.pth'):
    # 加载之前的检查点

    checkpoint = torch.load(args.save_path + 'PF_checkpoint.pth')
    model.load_state_dict(checkpoint['model_state_dict'])
    optim.load_state_dict(checkpoint['optimizer_state_dict'])
    start_epoch = checkpoint['epoch']
    loss = checkpoint['loss']
    print("load previous model epoch {}".format(checkpoint['epoch']))
else:
    start_epoch = 0

for e in range(start_epoch, args.total_epoch):
    model.train()
    losses = []
    acces = []
    for img, label in tqdm(iter(train_dataloader)):
        step_count += 1
        img = img.to(device)
        label = label.to(device)
        logits = model(img)
        loss = loss_fn(logits, label)
        acc = acc_fn(logits, label)
        loss.backward()
        if step_count % steps_per_update == 0:
            optim.step()
            optim.zero_grad()
        losses.append(loss.item())
        acces.append(acc.item())
    lr_scheduler.step()
```

```
        avg_train_loss = sum(losses) / len(losses)
        avg_train_acc = sum(acces) / len(acces)
        print(f'In epoch {e}, average training loss is {avg_train_loss}, average training acc is
{avg_train_acc}.')

        model.eval()
        with torch.no_grad():
            losses = []
            acces = []
            for img, label in tqdm(iter(val_dataloader)):
                img = img.to(device)
                label = label.to(device)
                logits = model(img)
                loss = loss_fn(logits, label)
                acc = acc_fn(logits, label)
                losses.append(loss.item())
                acces.append(acc.item())
            avg_val_loss = sum(losses) / len(losses)
            avg_val_acc = sum(acces) / len(acces)
            print(f'In epoch {e}, average validation loss is {avg_val_loss}, average validation acc
is {avg_val_acc}.')

        if avg_val_acc > best_val_acc:
            best_val_acc = avg_val_acc
            print(f'saving best model with acc {best_val_acc} at {e} epoch!')
            torch.save(model, args.save_path + args.output_model_path)

        writer.add_scalars('cls/loss', {'train': avg_train_loss, 'val': avg_val_loss}, global_step=e)
        writer.add_scalars('cls/acc', {'train': avg_train_acc, 'val': avg_val_acc}, global_step=e)

        # save checkpoint
        # 保存检查点
        checkpoint = {
            'epoch': e + 1,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optim.state_dict(),
            'loss': loss
        }
        torch.save(checkpoint, args.save_path + 'PF_checkpoint.pth')
```

*DMAE_model.py*

```
import torch
import timm
import numpy as np

from einops import repeat, rearrange
```

```
from einops.layers.torch import Rearrange

from timm.models.layers import trunc_normal_
from timm.models.vision_transformer import Block

def random_indexes(size : int):
    forward_indexes = np.arange(size)
    np.random.shuffle(forward_indexes)
    backward_indexes = np.argsort(forward_indexes)
    return forward_indexes, backward_indexes

def take_indexes(sequences, indexes):
    return torch.gather(sequences, 0, repeat(indexes, 't b -> t b c', c=sequences.shape[-1]))


def add_gaussian_noise(patche, std = 0.9):
    noise = torch.randn(patche.size()) * std
    noisy_patche = patche + noise.to(patche.device)
    return noisy_patche

class PatchNoise(torch.nn.Module):
    def __init__(self, Noiseratio) -> None:
        super().__init__()
        self.Noiseratio = Noiseratio

    def forward(self, patches : torch.Tensor):
        T, B, C = patches.shape
        remain_T = int(T * (1 - self.Noiseratio))

        indexes = [random_indexes(T) for _ in range(B)]
        forward_indexes = torch.as_tensor(np.stack([i[0] for i in indexes], axis=-1),
dtype=torch.long).to(patches.device)
        backward_indexes = torch.as_tensor(np.stack([i[1] for i in indexes], axis=-1),
dtype=torch.long).to(patches.device)

        patches = take_indexes(patches, forward_indexes)
        for i in range(remain_T,T):
          patches[i] = add_gaussian_noise(patches[i])

        return patches, forward_indexes, backward_indexes,remain_T

class MAE_Encoder(torch.nn.Module):
    def __init__(self,
                 image_size=32,
                 patch_size=2,
                 emb_dim=192,
```

```
                 num_layer=12,
                 num_head=3,
                 mask_ratio=0.75,
                 ) -> None:
        super().__init__()

        self.cls_token = torch.nn.Parameter(torch.zeros(1, 1, emb_dim))
        self.pos_embedding = torch.nn.Parameter(torch.zeros((image_size // patch_size) ** 2, 1,
emb_dim))
        self.shuffle = PatchNoise(mask_ratio)

        self.patchify = torch.nn.Conv2d(3, emb_dim, patch_size, patch_size)

        self.transformer = torch.nn.Sequential(*[Block(emb_dim, num_head) for _ in range(num_layer)])

        self.layer_norm = torch.nn.LayerNorm(emb_dim)

        self.init_weight()

    def init_weight(self):
        trunc_normal_(self.cls_token, std=.02)
        trunc_normal_(self.pos_embedding, std=.02)

    def forward(self, img):
        patches = self.patchify(img)
        patches = rearrange(patches, 'b c h w -> (h w) b c')
        patches = patches + self.pos_embedding

        patches, forward_indexes, backward_indexes,remain_T = self.shuffle(patches)

        patches = torch.cat([self.cls_token.expand(-1, patches.shape[1], -1), patches], dim=0)
        patches = rearrange(patches, 't b c -> b t c')
        features = self.layer_norm(self.transformer(patches))
        features = rearrange(features, 'b t c -> t b c')

        return features, backward_indexes,remain_T



class MAE_Decoder(torch.nn.Module):
    def __init__(self,
                 image_size=32,
                 patch_size=2,
                 emb_dim=192,
                 num_layer=4,
                 num_head=3,
```

```
                ) -> None:
        super().__init__()

        self.mask_token = torch.nn.Parameter(torch.zeros(1, 1, emb_dim))
        self.pos_embedding = torch.nn.Parameter(torch.zeros((image_size // patch_size) ** 2 + 1, 1,
emb_dim))

        self.transformer = torch.nn.Sequential(*[Block(emb_dim, num_head) for _ in range(num_layer)])

        self.head = torch.nn.Linear(emb_dim, 3 * patch_size ** 2)
        self.patch2img = Rearrange('(h w) b (c p1 p2) -> b c (h p1) (w p2)', p1=patch_size,
p2=patch_size, h=image_size//patch_size)

        self.init_weight()

    def init_weight(self):
        trunc_normal_(self.mask_token, std=.02)
        trunc_normal_(self.pos_embedding, std=.02)

    def forward(self, features, backward_indexes,remain_T):
        # T = features.shape[0]
        backward_indexes = torch.cat([torch.zeros(1, backward_indexes.shape[1]).to(backward_indexes),
backward_indexes + 1], dim=0)
        #features = torch.cat([features, self.mask_token.expand(backward_indexes.shape[0] -
features.shape[0], features.shape[1], -1)], dim=0)
        features = take_indexes(features, backward_indexes)
        features = features + self.pos_embedding

        features = rearrange(features, 't b c -> b t c')
        features = self.transformer(features)
        features = rearrange(features, 'b t c -> t b c')
        features = features[1:] # remove global feature

        patches = self.head(features)
        mask = torch.zeros_like(patches)
        mask[remain_T:] = 1
        mask = take_indexes(mask, backward_indexes[1:] - 1)
        img = self.patch2img(patches)
        mask = self.patch2img(mask)

        return img, mask

class MAE_Teacher(torch.nn.Module):
    def __init__(self,
                 image_size=32,
                 patch_size=2,
```

```
                emb_dim=192,
                num_layer=12,
                num_head=3,
                ) -> None:
        super().__init__()

        self.cls_token = torch.nn.Parameter(torch.zeros(1, 1, emb_dim))
        self.pos_embedding = torch.nn.Parameter(torch.zeros((image_size // patch_size) ** 2, 1,
emb_dim))

        self.patchify = torch.nn.Conv2d(3, emb_dim, patch_size, patch_size)

        self.transformer = torch.nn.Sequential(*[Block(emb_dim, num_head) for _ in range(num_layer)])

        self.layer_norm = torch.nn.LayerNorm(emb_dim)

        self.init_weight()

    def init_weight(self):
        trunc_normal_(self.cls_token, std=.02)
        trunc_normal_(self.pos_embedding, std=.02)

    def forward(self, img):
        patches = self.patchify(img)
        patches = rearrange(patches, 'b c h w -> (h w) b c')
        patches = patches + self.pos_embedding
        patches = torch.cat([self.cls_token.expand(-1, patches.shape[1], -1), patches], dim=0)
        patches = rearrange(patches, 't b c -> b t c')
        features = self.layer_norm(self.transformer(patches))
        features = rearrange(features, 'b t c -> t b c')

        return features

class MAE_ViT(torch.nn.Module):
    def __init__(self,
                image_size=32,
                patch_size=2,
                emb_dim=192,
                encoder_layer=12,
                encoder_head=3,
                decoder_layer=4,
                decoder_head=3,
                mask_ratio=0.75,
                ) -> None:
        super().__init__()
```

```
        self.encoder = MAE_Encoder(image_size, patch_size, emb_dim, encoder_layer, encoder_head,
mask_ratio)
        self.decoder = MAE_Decoder(image_size, patch_size, emb_dim, decoder_layer, decoder_head)

    def forward(self, img):
        features, backward_indexes,remain_T = self.encoder(img)
        predicted_img, mask = self.decoder(features,  backward_indexes,remain_T)
        return predicted_img, mask, features

class ViT_Classifier(torch.nn.Module):
    def __init__(self, encoder : MAE_Encoder, num_classes=10) -> None:
        super().__init__()
        self.cls_token = encoder.cls_token
        self.pos_embedding = encoder.pos_embedding
        self.patchify = encoder.patchify
        self.transformer = encoder.transformer
        self.layer_norm = encoder.layer_norm
        self.head = torch.nn.Linear(self.pos_embedding.shape[-1], num_classes)

    def forward(self, img):
        patches = self.patchify(img)
        patches = rearrange(patches, 'b c h w -> (h w) b c')
        patches = patches + self.pos_embedding
        patches = torch.cat([self.cls_token.expand(-1, patches.shape[1], -1), patches], dim=0)
        patches = rearrange(patches, 't b c -> b t c')
        features = self.layer_norm(self.transformer(patches))
        features = rearrange(features, 'b t c -> t b c')
        logits = self.head(features[0])
        return logits


if __name__ == '__main__':
    shuffle = PatchNoise(0.75)
    a = torch.rand(16, 2, 10)
    b, forward_indexes, backward_indexes,_ = shuffle(a)
    print(backward_indexes.shape)

    img = torch.rand(2, 3, 32, 32)
    encoder = MAE_Encoder()
    decoder = MAE_Decoder()
    teacher_network = MAE_Teacher()

    features, backward_indexes,remain_T = encoder(img)
    features2= teacher_network(img)
    print(features2.shape)
    print(features.shape)
```

```
    predicted_img, mask = decoder(features, backward_indexes,remain_T)
    # print(predicted_img.shape)
    # print(mask.shape)

    loss = torch.mean((predicted_img - img) ** 2) + torch.mean((features2 - features) ** 2)
    print(loss)
```

*DMAE_Pretrain.py*

```
import os
import argparse
import math
import torch
import torchvision
from torch.utils.tensorboard import SummaryWriter
from torchvision.transforms import ToTensor, Compose, Normalize
from tqdm import tqdm

from model import *
from utils import setup_seed,cosine_scheduler
from torchvision import utils as vutils
from PIL import Image

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--seed', type=int, default=42)
    parser.add_argument('--patch_size', type=int, default=2)
    parser.add_argument('--batch_size', type=int, default=2048)
    parser.add_argument('--max_device_batch_size', type=int, default=128)
    parser.add_argument('--base_learning_rate', type=float, default=1.5e-4)
    parser.add_argument('--weight_decay', type=float, default=0.05)
    parser.add_argument('--mask_ratio', type=float, default=0.75)
    parser.add_argument('--total_epoch', type=int, default=500)
    parser.add_argument('--warmup_epoch', type=int, default=20)
    parser.add_argument('--model_path', type=str, default='noise-mae.pt')
    parser.add_argument('--data_name', type=str, default='cifar10')
    parser.add_argument('--vit_type', type=str, default="vit-tiny")
    parser.add_argument('--save_path', type=str, default='')
    parser.add_argument('--image_size', type=int, default=32)
    args = parser.parse_args()

    setup_seed(args.seed)

    batch_size = args.batch_size
    load_batch_size = min(args.max_device_batch_size, batch_size)
```

```
    assert batch_size % load_batch_size == 0
    steps_per_update = batch_size // load_batch_size


    if args.data_name == 'cifar10':
        train_dataset = torchvision.datasets.CIFAR10('data', train=True, download=True,
                                                     transform=Compose([ToTensor(), Normalize(0.5,
0.5)]))
        val_dataset = torchvision.datasets.CIFAR10('data', train=False, download=True,
                                                   transform=Compose([ToTensor(), Normalize(0.5,
0.5)]))
    elif args.data_name == 'cifar100':
        train_dataset = torchvision.datasets.CIFAR100('data', train=True, download=True,
                                                     transform=Compose([ToTensor(), Normalize(0.5,
0.5)]))
        val_dataset = torchvision.datasets.CIFAR100('data', train=False, download=True,
                                                   transform=Compose([ToTensor(), Normalize(0.5,
0.5)]))
    elif args.data_name == 'stl10':
        train_dataset = torchvision.datasets.STL10('data', transform=Compose([ToTensor(),
Normalize(0.5, 0.5)]), split='train', download=True)
        val_dataset = torchvision.datasets.STL10('data', split='test', download=True,
transform=Compose([ToTensor(), Normalize(0.5, 0.5)]))

    dataloader = torch.utils.data.DataLoader(train_dataset, load_batch_size, shuffle=True,
num_workers=4)

    #dataloader,num_class,val_dataset  = load_data(args.data_dir, args.data_name,True,
args.image_size, load_batch_size, 2)
    writer = SummaryWriter(args.save_path + os.path.join('logs',args.data_name, 'n_mae_pretrain'))
    device = 'cuda' if torch.cuda.is_available() else 'cpu'


    # if args.vit_type == "vit_tiny":
    student_model = MAE_ViT(mask_ratio=args.mask_ratio,
                    image_size = args.image_size,
                    patch_size=args.patch_size).to(device)
    teacher_model = MAE_Teacher(image_size = args.image_size,
                                patch_size=args.patch_size).to(device)
    # elif args.vit_type == "vit_base":
    #     student_model = MAE_ViT(mask_ratio=args.mask_ratio,
    #                     image_size=args.image_size,
    #                     patch_size=args.patch_size,
    #                     emb_dim = 768,
    #                     encoder_layer = 12,
    #                     encoder_head = 12,
```

```
    #                     decoder_layer = 6,
    #                     decoder_head = 12,).to(device)
    #     teacher_model = MAE_Teacher(image_size = args.image_size,
    #                                 patch_size=args.patch_size,
    #                                 emb_dim = 768,
    #                                 num_layer = 12,
    #                                 num_head = 12,).to(device)

    optim = torch.optim.AdamW(student_model.parameters(), lr=args.base_learning_rate *
args.batch_size / 256, betas=(0.9, 0.95), weight_decay=args.weight_decay)
    lr_func = lambda epoch: min((epoch + 1) / (args.warmup_epoch + 1e-8), 0.5 * (math.cos(epoch /
args.total_epoch * math.pi) + 1))
    lr_scheduler = torch.optim.lr_scheduler.LambdaLR(optim, lr_lambda=lr_func, verbose=True)

    step_count = 0
    optim.zero_grad()

    momentum_schedule = cosine_scheduler(0.96, 0.99,args.total_epoch, 1)

    # 检查是否存在之前的检查点文件
    if os.path.exists(args.save_path + 'student_checkpoint.pth'):
        # 加载之前的student_model
        checkpoint_s = torch.load(args.save_path + 'student_checkpoint.pth')
        student_model.load_state_dict(checkpoint_s['model_state_dict'])
        optim.load_state_dict(checkpoint_s['optimizer_state_dict'])
        start_epoch = checkpoint_s['epoch']
        loss = checkpoint_s['loss']
        # 加载之前的teacher_model
        checkpoint_t = torch.load(args.save_path + 'teacher_checkpoint.pth')
        teacher_model.load_state_dict(checkpoint_t['model_state_dict'])
        print("load previous model epoch {}".format(checkpoint_s['epoch']))
    else:
        start_epoch = 0

    # ema update teacher

    for e in range(args.total_epoch):
        student_model.train()
        losses = []
        m = momentum_schedule[e]
        for img, label in tqdm(iter(dataloader)):
            step_count += 1
            img = img.to(device)
            #teacher network
            features_t = teacher_model(img)
            #mae_student
```

```
            predicted_img, mask, features_s = student_model(img)
            #new_loss
            loss = torch.mean((predicted_img - img) ** 2 ) + torch.mean((features_t - features_s) **
2)
            loss.backward()



            # with torch.no_grad():
            #   for param_s, param_t in zip(student_model.encoder.parameters(),
teacher_model.parameters()):
            #       param_t.data.mul_(m).add_((1 - m) * param_s.detach().data)

            if step_count % steps_per_update == 0:
                optim.step()
                optim.zero_grad()
            losses.append(loss.item())

        with torch.no_grad():
          for param_s, param_t in zip(student_model.encoder.parameters(),
teacher_model.parameters()):
              param_t.data.mul_(m).add_((1 - m) * param_s.detach().data)

        lr_scheduler.step()
        avg_loss = sum(losses) / len(losses)
        writer.add_scalar('mae_loss', avg_loss, global_step=e)
        print(f'In epoch {e}, average traning loss is {avg_loss}.')



        ''' visualize the first 16 predicted images on val dataset'''
        student_model.eval()
        with torch.no_grad():
            val_img = torch.stack([val_dataset[i][0] for i in range(48)])
            val_img = val_img.to(device)
            predicted_val_img, mask,_ = student_model(val_img)
            predicted_val_img = predicted_val_img * mask + val_img * (1 - mask)

            grid = vutils.make_grid(val_img, nrow=8, padding=2, normalize=True)
            grid_2 =  vutils.make_grid(predicted_val_img, nrow=8, padding=2, normalize=True)
            grid_3 =  vutils.make_grid(val_img * (1 - mask), nrow=8, padding=2, normalize=True)

            img = torch.cat([val_img * (1 - mask), predicted_val_img, val_img], dim=0)
            img = rearrange(img, '(v h1 w1) c h w -> c (h1 h) (w1 v w)', w1=2, v=3)
            writer.add_image('mae_image', (img + 1) / 2, global_step=e)
```

```
            vutils.save_image(img.to(torch.device('cpu')), args.save_path + "image.png")
            vutils.save_image(grid.to(torch.device('cpu')), args.save_path + "1_image.png")
            vutils.save_image(grid_2.to(torch.device('cpu')), args.save_path + "2_image.png")
            vutils.save_image(grid_3.to(torch.device('cpu')), args.save_path + "3_image.png")

        ''' save model '''
        torch.save(student_model, args.model_path)
        checkpoint_student = {
            'epoch': e + 1,
            'model_state_dict': student_model.state_dict(),
            'optimizer_state_dict': optim.state_dict(),
            'loss': loss
        }
        torch.save(checkpoint_student, args.save_path + 'student_checkpoint.pth')

        checkpoint_teacher = {
            'model_state_dict': teacher_model.state_dict()
        }
        torch.save(checkpoint_teacher, args.save_path + 'teacher_checkpoint.pth')
```

## *Chapter 4 Experimental Part Code*

*Mixup_pretrain_med.py*

```
import os
import argparse
import math
import torch
import torchvision
from torch.utils.tensorboard import SummaryWriter
from torchvision.transforms import ToTensor, Compose, Normalize
from tqdm import tqdm
from torchvision import utils as vutils
import torchvision.transforms as transforms
import torchvision.datasets as datasets
from torch.utils.data import Dataset, DataLoader
import tifffile as tiff

from model import *
from utils import setup_seed,CovidCTDataset

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--seed', type=int, default=42)
    parser.add_argument('--batch_size', type=int, default=2048)
    parser.add_argument('--max_device_batch_size', type=int, default=256)
    parser.add_argument('--base_learning_rate', type=float, default=1.5e-4)
```

```
parser.add_argument('--weight_decay', type=float, default=0.05)
parser.add_argument('--mask_ratio', type=float, default=0.6)
parser.add_argument('--total_epoch', type=int, default=800)
parser.add_argument('--warmup_epoch', type=int, default=50)
parser.add_argument('--model_path', type=str, default='vit_mae_sobel_Hog_feature.pt')
parser.add_argument('--save_path', type=str, default='./')
args = parser.parse_args()

setup_seed(args.seed)

batch_size = args.batch_size
load_batch_size = min(args.max_device_batch_size, batch_size)

assert batch_size % load_batch_size == 0
steps_per_update = batch_size // load_batch_size

dataPath = "/content/CT_data"
CT_data_name = os.listdir(dataPath)
targetpath = "/content/CT_data_mixup_features_targets"
##训练集
trainCT_COVID = CT_data_name[0:60000]
valCT_COVID = CT_data_name[60000:]
normalize = transforms.Normalize(mean=[0.45271412, 0.45271412, 0.45271412],
                                 std=[0.33165374, 0.33165374, 0.33165374])

tar_transform = transforms.Compose([
transforms.Resize((224, 224)),  # Resize the image
transforms.ToTensor()           # Convert the image to a tensor
])

train_transformer = transforms.Compose([
    transforms.Resize((224,224)),
    # transforms.RandomResizedCrop((224),scale=(0.5,1.0)),
    # transforms.RandomHorizontalFlip(),
    # transforms.ColorJitter(brightness=0.2, contrast=0.2),
    transforms.ToTensor(),
    normalize
])

val_transformer = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.ToTensor(),
    normalize
])

train_dataset = CovidCTDataset(data_dir=dataPath,
```

```
                                    target_dir=targetpath,
                                    txt_path= trainCT_COVID,
                                    transform= train_transformer,
                                    tar_transform =tar_transform)
    val_dataset = CovidCTDataset(data_dir=dataPath,
                                 target_dir=targetpath,
                                 txt_path=valCT_COVID,
                                 transform= val_transformer,
                                 tar_transform =tar_transform)
    dataloader = torch.utils.data.DataLoader(train_dataset, load_batch_size, shuffle=True,
num_workers=4)
    writer = SummaryWriter(args.save_path + os.path.join('logs_s_b', 'ct', 'mae_feature_-pretrain'))
    device = 'cuda' if torch.cuda.is_available() else 'cpu'

    model = MAE_ViT(mask_ratio=args.mask_ratio).to(device)
    optim = torch.optim.AdamW(model.parameters(), lr=args.base_learning_rate * args.batch_size / 256,
betas=(0.9, 0.95), weight_decay=args.weight_decay)
    lr_func = lambda epoch: min((epoch + 1) / (args.warmup_epoch + 1e-8), 0.5 * (math.cos(epoch /
args.total_epoch * math.pi) + 1))
    lr_scheduler = torch.optim.lr_scheduler.LambdaLR(optim, lr_lambda=lr_func, verbose=True)

    step_count = 0
    optim.zero_grad()


    # 检查是否存在之前的检查点文件
    if os.path.exists(args.save_path+'checkpoint.pth'):
        # 加载之前的检查点

        checkpoint = torch.load(args.save_path+'checkpoint.pth')
        model.load_state_dict(checkpoint['model_state_dict'])
        optim.load_state_dict(checkpoint['optimizer_state_dict'])
        start_epoch = checkpoint['epoch']
        loss = checkpoint['loss']
        print("load previous model epoch {}".format(checkpoint['epoch']))
    else:
        start_epoch = 0

    for e in range(start_epoch,args.total_epoch):
        model.train()
        losses = []
        for img,mixup_img in tqdm(iter(dataloader)):
            step_count += 1
            img = img.to(device)
            mixup_img = mixup_img.to(device)
            # mixup_img = tar
```

```
        predicted_img, mask = model(img)
        loss = torch.mean((predicted_img - mixup_img) ** 2 * mask) / args.mask_ratio
        loss.backward()
        if step_count % steps_per_update == 0:
            optim.step()
            optim.zero_grad()
        losses.append(loss.item())
    lr_scheduler.step()
    avg_loss = sum(losses) / len(losses)
    writer.add_scalar('mae_loss', avg_loss, global_step=e)
    print(f'In epoch {e}, average traning loss is {avg_loss}.')

    ''' visualize the first 16 predicted images on val dataset'''
    model.eval()
    with torch.no_grad():
        val_img = torch.stack([val_dataset[i][0] for i in range(48)])
        val_img = val_img.to(device)
        sobel_val_img = torch.stack([val_dataset[i][1] for i in range(48)])
        sobel_val_img = sobel_val_img.to(device)
        predicted_val_img, mask = model(val_img)
        grid_4 = vutils.make_grid(predicted_val_img, nrow=8, padding=2, normalize=True)
        predicted_val_img = predicted_val_img * mask + sobel_val_img * (1 - mask)

        grid = vutils.make_grid(val_img, nrow=8, padding=2, normalize=True)
        grid_1 = vutils.make_grid(sobel_val_img, nrow=8, padding=2, normalize=True)
        grid_2 = vutils.make_grid(predicted_val_img, nrow=8, padding=2, normalize=True)
        grid_3 = vutils.make_grid(val_img * (1 - mask), nrow=8, padding=2, normalize=True)

        writer.add_image('mask_image', grid_3, global_step=e)
        writer.add_image('pred_image', grid_2, global_step=e)
        writer.add_image('targ_image', grid_1, global_step=e)
        writer.add_image('orig_image', grid, global_step=e)
        vutils.save_image(grid.to(torch.device('cpu')), args.save_path + "real_image.png")
        vutils.save_image(grid_1.to(torch.device('cpu')), args.save_path + "target_image.png")
        vutils.save_image(grid_2.to(torch.device('cpu')), args.save_path + "pred_image.png")
        vutils.save_image(grid_3.to(torch.device('cpu')), args.save_path + "masked_image.png")
        vutils.save_image(grid_4.to(torch.device('cpu')), args.save_path + "pred_image_1.png")
    ''' save model '''
    torch.save(model, args.save_path+args.model_path)
    # save checkpoint
    checkpoint = {
        'epoch': e + 1,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optim.state_dict(),
        'loss': loss
    }
```

```
        torch.save(checkpoint, args.save_path+'checkpoint.pth')
```

*classification_mixup_med.py*

```
import os
import argparse
import math
import torch
import torchvision
from torch.utils.tensorboard import SummaryWriter
from torchvision import transforms
from tqdm import tqdm

from model import *
from utils import setup_seed,CovidCTDataset_for_down_task
from torch.utils.data import Dataset, DataLoader
from sklearn.metrics import roc_auc_score, f1_score

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--seed', type=int, default=42)
    parser.add_argument('--batch_size', type=int, default=512)
    parser.add_argument('--max_device_batch_size', type=int, default=256)
    parser.add_argument('--base_learning_rate', type=float, default=1e-3)
    parser.add_argument('--weight_decay', type=float, default=0.05)
    parser.add_argument('--total_epoch', type=int, default=100)
    parser.add_argument('--warmup_epoch', type=int, default=5)
    parser.add_argument('--pretrained_model_path', type=str, default=None)
    parser.add_argument('--output_model_path', type=str, default='classifier-from_pertain.pt')
    parser.add_argument('--save_path', type=str, default='./')
    args = parser.parse_args()

    setup_seed(args.seed)

    batch_size = args.batch_size
    load_batch_size = min(args.max_device_batch_size, batch_size)

    assert batch_size % load_batch_size == 0
    steps_per_update = batch_size // load_batch_size

    ### 划分数据集
    filetrainPath_COVID = "/content/train/CT_COVID"
    filetrainpath_NO_COVID = "/content/train/CT_NonCOVID"
    filetestPath_COVID = "/content/test/CT_COVID"
    filetestpath_NO_COVID = "/content/test/CT_NonCOVID"

    filevalPath_COVID = "/content/val/CT_COVID"
    filevalpath_NO_COVID = "/content/val/CT_NonCOVID"
```

```
CT_train_COVID_name = os.listdir(filetrainPath_COVID)
CT_train_NonCOVID_name = os.listdir(filetrainpath_NO_COVID)
CT_test_COVID_name = os.listdir(filetestPath_COVID)
CT_test_NonCOVID_name = os.listdir(filetestpath_NO_COVID)
CT_val_COVID_name = os.listdir(filevalPath_COVID)
CT_val_NonCOVID_name = os.listdir(filevalpath_NO_COVID)


normalize = transforms.Normalize(mean=[0.45271412, 0.45271412, 0.45271412],
                                 std=[0.33165374, 0.33165374, 0.33165374])

train_transformer = transforms.Compose([
    transforms.Resize(256),
    transforms.RandomResizedCrop((224),scale=(0.5,1.0)),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(brightness=0.2, contrast=0.2),
    transforms.ToTensor(),
    normalize
])

val_transformer = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.ToTensor(),
    normalize
])



trainset = CovidCTDataset_for_down_task(root_dir='/content/train/',
                           txt_COVID= CT_train_COVID_name,
                           txt_NonCOVID=CT_train_NonCOVID_name,
                           transform= train_transformer)
valset = CovidCTDataset_for_down_task(root_dir='/content/val/',
                           txt_COVID=CT_val_COVID_name,
                           txt_NonCOVID=CT_val_NonCOVID_name,
                           transform= val_transformer)
testset = CovidCTDataset_for_down_task(root_dir='/content/test/',
                           txt_COVID=CT_test_COVID_name ,
                           txt_NonCOVID=CT_test_NonCOVID_name ,
                           transform= val_transformer)
print("划分后训练集数据个数：",trainset.__len__())
print("划分后验证集集数据个数：",valset.__len__())
print("划分后测试集数据个数：",testset.__len__())
```

```
#载入数据
train_loader = DataLoader(trainset, batch_size=load_batch_size , drop_last=False, shuffle=True)
val_loader = DataLoader(valset, batch_size=load_batch_size , drop_last=False, shuffle=True)
test_loader = DataLoader(testset, batch_size=load_batch_size, drop_last=False, shuffle=False)
device = 'cuda' if torch.cuda.is_available() else 'cpu'
model = torch.load(args.save_path + args.pretrained_model_path, map_location='cpu')
writer = SummaryWriter(args.save_path + os.path.join('logs','pretrain_mixup_mdoel'))
model = ViT_Classifier(model.encoder, num_classes=2).to(device)
loss_fn = torch.nn.CrossEntropyLoss()
acc_fn = lambda logit, label: torch.mean((logit.argmax(dim=-1) == label).float())
optim = torch.optim.AdamW(model.parameters(), lr=args.base_learning_rate * args.batch_size / 256,
                          betas=(0.9, 0.999), weight_decay=args.weight_decay)
lr_func = lambda epoch: min((epoch + 1) / (args.warmup_epoch + 1e-8),
                            0.5 * (math.cos(epoch / args.total_epoch * math.pi) + 1))
lr_scheduler = torch.optim.lr_scheduler.LambdaLR(optim, lr_lambda=lr_func, verbose=True)

best_val_acc = 0
step_count = 0
optim.zero_grad()
# 检查是否存在之前的检查点文件
if os.path.exists(args.save_path + 'PF_checkpoint.pth'):
    # 加载之前的检查点

    checkpoint = torch.load(args.save_path + 'PF_checkpoint.pth')
    model.load_state_dict(checkpoint['model_state_dict'])
    optim.load_state_dict(checkpoint['optimizer_state_dict'])
    start_epoch = checkpoint['epoch']
    loss = checkpoint['loss']
    print("load previous fineturn model epoch {}".format(checkpoint['epoch']))
else:
    start_epoch = 0

for e in range(start_epoch, args.total_epoch):
    model.train()
    losses = []
    acces = []
    for batch_samples in tqdm(iter(train_loader)):
        step_count += 1
        img = batch_samples['img'].to(device)
        label = batch_samples['label'].to(device)
        logits = model(img)
        loss = loss_fn(logits, label)
        acc = acc_fn(logits, label)
        loss.backward()
        if step_count % steps_per_update == 0:
            optim.step()
```

```
                optim.zero_grad()
            losses.append(loss.item())
            acces.append(acc.item())
        lr_scheduler.step()
        avg_train_loss = sum(losses) / len(losses)
        avg_train_acc = sum(acces) / len(acces)
        print(f'In epoch {e}, average training loss is {avg_train_loss}, average training acc is
{avg_train_acc}.')
        model.eval()
        with torch.no_grad():
            losses = []
            acces = []
            f1s = []
            AUCs = []
            for batch_samples in tqdm(iter(val_loader)):
                img = batch_samples['img'].to(device)
                label = batch_samples['label'].to(device)
                logits = model(img)
                loss = loss_fn(logits, label)
                acc = acc_fn(logits, label)
                logits_np = logits.cpu().detach().numpy()
                label_np = label.cpu().detach().numpy()
                auc = roc_auc_score(label_np,logits_np[:, 1])
                f1 = f1_score(label_np,np.argmax(logits_np, axis=1))
                losses.append(loss.item())
                acces.append(acc.item())
                f1s.append(f1)
                AUCs.append(auc)
            avg_val_loss = sum(losses) / len(losses)
            avg_val_acc = sum(acces) / len(acces)
            avg_f1 =  sum(f1s) / len(f1s)
            avg_auc = sum(AUCs) / len(AUCs)
            print(f'In epoch {e}, average validation loss is {avg_val_loss}, average validation acc
is {avg_val_acc},average f1 {avg_f1},avg auc {avg_auc}')

        if avg_val_acc > best_val_acc:
            best_val_acc = avg_val_acc
            print(f'saving best model with acc {best_val_acc} at {e} epoch!')
            torch.save(model, args.save_path + args.output_model_path)

        writer.add_scalars('cls/loss', {'train': avg_train_loss, 'val': avg_val_loss}, global_step=e)
        writer.add_scalars('cls/acc', {'train': avg_train_acc, 'val': avg_val_acc}, global_step=e)
        writer.add_scalars('cls/f1', {'val': avg_f1}, global_step=e)
        writer.add_scalars('cls/auc', {'val': avg_auc}, global_step=e)
        # save checkpoint
        # 保存检查点
```

```
        checkpoint = {
            'epoch': e + 1,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optim.state_dict(),
            'loss': loss
        }
        torch.save(checkpoint, args.save_path + 'PF_checkpoint.pth')
```

*imbalance_study_mixup.py*

```
import os
import argparse
import math
import torch
import torchvision
from torch.utils.tensorboard import SummaryWriter
from torchvision import transforms
from tqdm import tqdm

from model import *
from utils import setup_seed,CovidCTDataset_for_down_task
from torch.utils.data import Dataset, DataLoader
from sklearn.metrics import roc_auc_score, f1_score

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--seed', type=int, default=42)
    parser.add_argument('--batch_size', type=int, default=256)
    parser.add_argument('--max_device_batch_size', type=int, default=128)
    parser.add_argument('--base_learning_rate', type=float, default=1e-3)
    parser.add_argument('--weight_decay', type=float, default=0.05)
    parser.add_argument('--total_epoch', type=int, default=400)
    parser.add_argument('--warmup_epoch', type=int, default=5)
    parser.add_argument('--pretrained_model_path', type=str, default=None)
    parser.add_argument('--output_model_path', type=str, default='classifier-from_pertain.pt')
    parser.add_argument('--save_path', type=str, default='./')
    parser.add_argument('--imblance', type=str, default='1-1')
    args = parser.parse_args()

    setup_seed(args.seed)

    batch_size = args.batch_size
    load_batch_size = min(args.max_device_batch_size, batch_size)

    assert batch_size % load_batch_size == 0
    steps_per_update = batch_size // load_batch_size
```

```
### dataset
filePath_COVID = "/content/COVID"
filepath_NO_COVID = "/content/non-COVID"
CT_COVID_name = os.listdir(filePath_COVID)
CT_NonCOVID_name = os.listdir(filepath_NO_COVID)
print("COVID",len(CT_COVID_name))
print("non-COVID",len(CT_NonCOVID_name))

train_count = 1000
if args.imblance == "1-1":
  train_count = 1000
elif args.imblance == "2-1":
  train_count = 500
elif args.imblance == "4-1":
  train_count = 250
elif args.imblance == "8-1":
  train_count = 125

trainCT_COVID = CT_COVID_name[0:train_count]
valCT_COVID = CT_COVID_name[1000:]
trainCT_NonCOVID = CT_NonCOVID_name[0:1000]
valCT_NonCOVID = CT_NonCOVID_name[1000:]

normalize = transforms.Normalize(mean=[0.45271412, 0.45271412, 0.45271412],
                                 std=[0.33165374, 0.33165374, 0.33165374])

#数据增强
train_transformer = transforms.Compose([
    transforms.Resize(256),
    transforms.RandomResizedCrop((224),scale=(0.5,1.0)),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(brightness=0.2, contrast=0.2),
    transforms.ToTensor(),
    normalize
])

val_transformer = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.ToTensor(),
    normalize
])

trainset = CovidCTDataset_for_down_task(root_dir='/content/',
                           txt_COVID= trainCT_COVID,
                           txt_NonCOVID=trainCT_NonCOVID,
```

```
                            transform= train_transformer)
valset = CovidCTDataset_for_down_task(root_dir='/content/',
                            txt_COVID=valCT_COVID,
                            txt_NonCOVID=valCT_NonCOVID,
                            transform= val_transformer)


#load data
train_loader = DataLoader(trainset, batch_size=load_batch_size, drop_last=False, shuffle=True)
val_loader = DataLoader(valset, batch_size=load_batch_size, drop_last=False, shuffle=True)


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


model = torch.load(args.save_path + args.pretrained_model_path, map_location='cpu')
writer = SummaryWriter(args.save_path + os.path.join('logs','pretrain_mae_mdoel'))

model = ViT_Classifier(model.encoder, num_classes=2).to(device)

loss_fn = torch.nn.CrossEntropyLoss()
acc_fn = lambda logit, label: torch.mean((logit.argmax(dim=-1) == label).float())

optim = torch.optim.AdamW(model.parameters(), lr=args.base_learning_rate * args.batch_size / 256,
                            betas=(0.9, 0.999), weight_decay=args.weight_decay)
lr_func = lambda epoch: min((epoch + 1) / (args.warmup_epoch + 1e-8),
                              0.5 * (math.cos(epoch / args.total_epoch * math.pi) + 1))
lr_scheduler = torch.optim.lr_scheduler.LambdaLR(optim, lr_lambda=lr_func, verbose=True)

best_val_acc = 0
step_count = 0
optim.zero_grad()
# 检查是否存在之前的检查点文件
if os.path.exists(args.save_path + 'PF_checkpoint.pth'):
    # 加载之前的检查点

    checkpoint = torch.load(args.save_path + 'PF_checkpoint.pth')
    model.load_state_dict(checkpoint['model_state_dict'])
    optim.load_state_dict(checkpoint['optimizer_state_dict'])
    start_epoch = checkpoint['epoch']
    loss = checkpoint['loss']
    print("load previous fineturn model epoch {}".format(checkpoint['epoch']))
else:
    start_epoch = 0

for e in range(start_epoch, args.total_epoch):
```

```
        model.train()
        losses = []
        acces = []
        for batch_samples in tqdm(iter(train_loader)):
            step_count += 1
            img = batch_samples['img'].to(device)
            label = batch_samples['label'].to(device)
            logits = model(img)
            loss = loss_fn(logits, label)
            acc = acc_fn(logits, label)
            loss.backward()
            if step_count % steps_per_update == 0:
                optim.step()
                optim.zero_grad()
            losses.append(loss.item())
            acces.append(acc.item())
        lr_scheduler.step()
        avg_train_loss = sum(losses) / len(losses)
        avg_train_acc = sum(acces) / len(acces)
        print(f'In epoch {e}, average training loss is {avg_train_loss}, average training acc is
{avg_train_acc}.')

        model.eval()
        with torch.no_grad():
            losses = []
            acces = []
            f1s = []
            AUCs = []
            for batch_samples in tqdm(iter(val_loader)):
                img = batch_samples['img'].to(device)
                label = batch_samples['label'].to(device)
                logits = model(img)
                loss = loss_fn(logits, label)
                acc = acc_fn(logits, label)
                logits_np = logits.cpu().detach().numpy()
                label_np = label.cpu().detach().numpy()
                auc = roc_auc_score(label_np,logits_np[:, 1])
                f1 = f1_score(label_np,np.argmax(logits_np, axis=1))
                losses.append(loss.item())
                acces.append(acc.item())
                f1s.append(f1)
                AUCs.append(auc)
            avg_val_loss = sum(losses) / len(losses)
            avg_val_acc = sum(acces) / len(acces)
            avg_f1 =  sum(f1s) / len(f1s)
            avg_auc = sum(AUCs) / len(AUCs)
```

```
            print(f'In epoch {e}, average validation loss is {avg_val_loss}, average validation acc
is {avg_val_acc},average f1 {avg_f1},avg auc {avg_auc}')

        if avg_val_acc > best_val_acc:
            best_val_acc = avg_val_acc
            print(f'saving best model with acc {best_val_acc} at {e} epoch!')
            torch.save(model, args.save_path + args.output_model_path)

        writer.add_scalars('cls/loss', {'train': avg_train_loss, 'val': avg_val_loss}, global_step=e)
        writer.add_scalars('cls/acc', {'train': avg_train_acc, 'val': avg_val_acc}, global_step=e)
        writer.add_scalars('cls/f1', {'val': avg_f1}, global_step=e)
        writer.add_scalars('cls/auc', {'val': avg_auc}, global_step=e)
        # save checkpoint
        # 保存检查点
        checkpoint = {
            'epoch': e + 1,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optim.state_dict(),
            'loss': loss
        }
        torch.save(checkpoint, args.save_path + 'PF_checkpoint.pth')
```

# APPENDIX B

## List of publications

1. **Jiashu Xu** and Sergii Stirenko, (2023) "Mixup Feature: A Pretext Task Self-Supervised Learning Method for Enhanced Visual Feature Learning," in IEEE Access, vol. 11, pp. 82400-82409, IEEE, ISSN: 2169-3536, DOI: 10.1109/ACCESS.2023.3301561 (Scopus Q1,WoS Q2).
2. **Jiashu Xu**, Sergii Stirenko, (2023) "Denoising Self-Distillation Masked Autoencoder for Self-Supervised Learning", International Journal of Image, Graphics and Signal Processing (IJIGSP), Vol.15, No.5, pp. 29-38. MECS Press, ISSN:2074-9074, DOI:10.5815/ijigsp.2023.05.03 (Scopus)
3. **Jiashu Xu**, Sergii Stirenko, (2022) "Self-Supervised Model Based on Masked Autoencoders Advance CT Scans Classification", International Journal of Image, Graphics and Signal Processing (IJIGSP), Vol.14, No.5, pp. 1-9. MECS Press, ISSN:2074-9074, DOI:10.5815/ijigsp.2022.05.01 (Scopus)
4. **Jiashu Xu**. (2021) "A review of self-supervised learning methods in the field of medical image analysis." International Journal of Image, Graphics and Signal Processing (IJIGSP) 13, no. 4: 33-46. MECS Press, ISSN:2074-9074, 10.5815/ijigsp.2021.04.03 (Scopus).
5. Yahu Yang, Hu Zhang, **Jiashu Xu**, Shenmin Song, (2023), “MATEKG: A Large-scale Multi-class Equipment Knowledge Graph for Military Auxiliary Tasks.” 2023 IEEE 6th International Conference on Information Systems and Computer Aided Education (ICISCAE), Dalian, China. (Scopus).

6. Yang, Ya-Hu, **Jiashu Xu**, Yuri Gordienko, and Sergii Stirenko. (2021). "Abnormal Interference Recognition Based on Rolling Prediction Average Algorithm." Advances in Computer Science for Engineering and Education III. ICCSEEA 2020. Advances in Intelligent Systems and Computing, vol 1247. Springer, Cham. https://doi.org/10.1007/978-3-030-55506-1_28 (Scopus)
7. **Jiashu Xu**, Sergii Stirenko, (2020), "FACIAL EXPRESSION RECOGNITION SYSTEM BASED ON GAN NETWORK DATA AUGMENTATION", The International Conference on Security, Fault Tolerance, Intelligence 2020, pp. 144-149.