DAT410 Assignment 8

Hongliang Zhong
zhongh@student.chalmers.se

Jianing Lei
jianingl@student.chalmers.se

March 21, 2025

1 Introduction

Remote sensing scene classification is a fundamental task in computer vision, aiming to automatically categorize aerial images into predefined scene classes. With the rapid development of satellite and drone-based imaging technologies, accurate classification of remote sensing images has become crucial for applications such as land use analysis, environmental monitoring, and urban planning. However, this task presents significant challenges due to the high intra-class variability (e.g., seasonal and weather changes) and low inter-class separability (e.g., visually similar scenes in different categories).

This project is directly related to **Module 5**: Computer Vision and Remote Sensing in our course. Similar to our coursework, we work with remote sensing data, but unlike Assignment 5, which focuses on a different task, our project primarily explores scene classification and evaluates the effectiveness of different deep learning models. In this work, we investigate and compare four different deep-learning architectures:

- Basic CNN – A simple convolutional neural network as a baseline.
- ResNet-50 – A widely used deep residual network with pre-trained ImageNet weights.
- Transformer-based models – Exploring vision transformers (ViT) for scene classification.
- Self-Supervised Learning (SSL) using SimCLR – A contrastive learning approach that learns representations from unlabeled data before fine-tuning on labeled samples.

To conduct this study, we use the **RSSCN7 dataset** [1], which consists of 2,800 remote sensing images from 7 typical scene categories: grassland, forest, farmland, parking lot, residential region, industrial region, and river & lake. The images are 400 × 400 pixels, covering diverse landscapes and environmental conditions. The variability in seasons, lighting conditions, and scales make RSSCN7 a challenging yet practical dataset for benchmarking remote sensing classification models.

By evaluating these models, we aim to analyze the impact of network depth, self-supervised learning, and transformer-based architectures on remote sensing image classification. Additionally, we explore whether self-supervised learning can reduce labeled data dependency and how Vision Transformers (ViTs) compare against traditional CNNs in this domain.

2 Methodology

2.1 Data preprocessing

We construct the dataset into a structured format with three columns: `image` (file path), `class_name` (scene label), and `class_num` (numerical label). To ensure fair and consistent evaluation, we randomly split the dataset into training, validation, and test sets with a ratio of **70%**,**15%**, and **15%**, respectively, while maintaining balanced class distributions. A fixed random seed was used to ensure reproducibility.

Data augmentation was applied to the training set to improve the model's generalization ability and robustness to variations in lighting, orientation, and composition. Specifically, the training images were randomly rotated, horizontally flipped, and color jittered in terms of brightness, contrast, saturation, and hue. All images were then converted to tensors and normalized using ImageNet mean and standard deviation values. For the validation and test sets, only resizing and normalization were applied, without additional augmentations, to ensure fair evaluation.

The training process was configured with consistent hyperparameters across all experiments: the batch size was set to 32, the number of training epochs was 30, and the learning rate was 0.0001. To avoid overfitting and reduce unnecessary computation, early stopping was employed with patience of 5 epochs, monitoring the validation loss throughout training.

2.2 Basic CNN

As a baseline for comparison, we implemented a simple convolutional neural network (CNN) architecture to perform scene classification on the RSSCN7 dataset. The network includes three convolutional blocks, each comprising a convolutional layer with increasing channel depth, a ReLU activation function, and a max pooling operation to progressively reduce the spatial resolution while capturing hierarchical features. The output feature maps are then flattened and passed through two fully connected layers, with the final layer producing logits corresponding to the seven scene categories. This model serves as an essential reference point to evaluate the effectiveness of more advanced models.

2.3 ResNet-50

ResNet-50 is a widely used deep convolutional neural network known for its residual learning framework. It consists of 50 layers with identity shortcut connections that allow the network to mitigate the vanishing gradient problem during backpropagation. These residual connections enable the training of significantly deeper networks without degradation in performance, making ResNet-50 a powerful choice for image classification tasks.

In this project, we used the ImageNet-pretrained version of ResNet-50 and replaced its final fully connected (FC) layer to adapt it to our specific classification task.

Several key design choices influence the fine-tuning process and impact the model's ability to generalize while maintaining efficient training. First, we freeze the initial 140 layers of ResNet-50 to preserve the fundamental visual features learned from ImageNet, such as edges and textures, while allowing only the final layers to adapt to the new dataset. This approach reduces the risk of overfitting and significantly decreases computational costs. Additionally, we replace the fully connected (FC) layer to match the number of target classes. A 256-unit hidden layer with ReLU activation is introduced to enhance learning capacity while maintaining efficiency. To further mitigate overfitting, we apply a dropout rate of 0.3, randomly deactivating neurons during training. The final output layer is adjusted to align with the number of classes in our classification task, ensuring compatibility and effective adaptation.

2.4 Transformer-based model

In addition, we explored a transformer-based model for our remote sensing scene classification task. We utilized the ViT-Base-Patch16-224 model introduced by Google, which applies the transformer architecture—originally developed for natural language processing—to image classification tasks.

The Vision Transformer (ViT) divides each input image into fixed-size patches (in our case, 16×16 pixels). Each patch is flattened and linearly projected into an embedding vector, forming a sequence that is fed into a standard transformer encoder. For the ViT-Base-Patch16-224 model, each image must be resized to 224×224 pixels before being split into patches. This is a notable distinction from the other models in our study, which implement directly on the original 400×400

resolution. Therefore, a key preprocessing step specific to the transformer model was resizing the original images to 224×224 prior to training and inference.

The model also combines positional encodings to retain spatial information and uses a learnable class token to aggregate global features for classification. Unlike convolutional networks, ViT relies on self-attention mechanisms to capture global context information. In our implementation, we fine-tuned the pre-trained model on the RSSCN7 dataset after replacing its classification head with a new fully connected layer suitable for seven classes.

2.5 Self-Supervised Learning (SSL)

2.5.1 SimCLR pretraining

Self-supervised learning has emerged as a powerful approach for representation learning, especially in scenarios with limited labeled data. In this project, we adopt SimCLR (Simple Framework for Contrastive Learning of Visual Representations) as our self-supervised method. SimCLR learns visual representations by maximizing the agreement between differently augmented views of the same image, without relying on labels during pretraining.

The core idea of SimCLR is to treat each augmented image pair as a positive pair and all other images in the batch as negatives. For each image, two augmented views are generated through a stochastic data augmentation pipeline. These are passed through a shared encoder (in our case, ResNet-50) and a projection head to obtain two latent representations. The model is trained using the Normalized Temperature-scaled Cross Entropy Loss (NT-Xent Loss), which encourages positive pairs to be close in the latent space while pushing apart the negative pairs [2]. We used a temperature parameter of 0.3 in the loss function to control the sharpness of similarity distribution.

In our implementation, the data augmentation for contrastive learning includes random resized cropping with a scale range of (0.2, 1.0), horizontal flipping, random application of color jittering (brightness, contrast, saturation, hue), and random grayscale conversion. These augmentations are useful for encouraging the model to learn invariant representations. All images are normalized using the standard ImageNet statistics.

2.5.2 Fine-tuning on RSSCN7

After pretraining, we evaluated the learned representations through supervised fine-tuning on the RSSCN7 dataset. We removed the projection head from the pre-trained SimCLR model and appended a new classification head consisting of a fully connected layer with 256 hidden units, a ReLU activation, dropout regularization, and a final output layer with seven units corresponding to the scene categories.

To investigate whether SimCLR pretraining can reduce the dependency on large amounts of labeled samples while still achieving competitive performance, we conducted two fine-tuning experiments. In the first setting, we used the entire training set (70% of the dataset), while in the second setting, we randomly sampled 50% of the training set. In both cases, the model was trained by using the labeled data.

3 Results and discussion

3.1 Performance Comparison Based on Test Accuracy

Table 1 presents the test accuracy of different models on the RSSCN7 dataset. Among the evaluated models, ResNet-50 achieves the highest accuracy (95.71%), followed closely by ViT-Base-Patch16-224 (94.05%). This suggests that deep convolutional networks and transformer-based architectures are highly effective for remote sensing image classification. The Basic CNN, serving as a baseline, performs significantly worse (76.19%), highlighting the benefits of deeper architectures and pretraining.

The SimCLR-based models, which employ self-supervised learning (SSL) followed by fine-tuning, exhibit lower accuracy than fully supervised models. The SimCLR model fine-tuned with 100% of the training data achieves 77.62%, slightly better than the Basic CNN but still well below ResNet-50 and ViT. When the labeled training data is reduced to 50%, the performance drops further to 68.33%, indicating that self-supervised pretraining alone does not fully compensate for the reduced supervision in this setting.

Model	**Test Accuracy (%)**
Basic CNN	76.19
ResNet-50	95.71
ViT-Base-Patch16-224	94.05
SimCLR + Fine-tuning (100% training data)	77.62
SimCLR + Fine-tuning (50% training data)	68.33

Table 1: Test accuracy comparison of different models

3.2 Training and Validation Loss/Accuracy Analysis

Examining the training and validation loss curves A reveals notable differences in convergence behavior across models. ResNet-50 demonstrates stable training, reaching near-optimal performance without significant overfitting. ViT, however, appears to require more epochs to converge, aligning with previous findings that transformer-based models often need extensive training. Additionally, ViT exhibits a larger gap between training and validation accuracy, suggesting a tendency to overfit.

The Basic CNN model aligns with expectations, as simpler networks with fewer parameters may struggle to generalize well without advanced architectural components like residual connections.

For SimCLR, the training curves indicate that pretraining learns feature representations, but fine-tuning still lags behind fully supervised models. The lower performance with 50% labeled data suggests that while SSL can reduce dependency on labels, it may require larger datasets or stronger augmentations to be more competitive in remote sensing tasks.

3.3 Confusion Matrix and Per-Class Performance

The confusion matrices B provide further insights into model performance. ResNet-50 and ViT show strong classification ability across all seven scene categories, though minor confusion occurs between visually similar classes, such as ViT model couldn't distinguish industry and parking very well. The Basic CNN exhibits more pronounced misclassifications, especially in similar scenes like residential and industrial area, grass and field, indicating weaker feature extraction capabilities.

Interestingly, the SimCLR-based models show slightly different misclassification patterns, possibly due to the nature of the learned representations. The performance drop with reduced training data is more evident in certain classes, especially industry, suggesting that SSL pretraining does not generalize equally across all scene types.

3.4 Limitations and Future Work

While this study provides valuable insights, certain limitations should be acknowledged. First, the dataset size may not be sufficient for fully leveraging the advantages of transformer-based models and self-supervised learning. Future work could explore whether pretraining on larger remote sensing datasets improves performance.

Additionally, experimenting with alternative SSL approaches, such as MoCo or SwAV, may provide a better understanding of the role of contrastive learning in remote sensing. Further architectural improvements, such as hybrid CNN-ViT models, could also be considered to combine the strengths of both paradigms.

References

[1] Google Drive Repository. *RSSCN7 Dataset for remote sensing scene classification*. Online. Accessed: 20-Mar-2025. 2025. URL: `https://drive.google.com/drive/folders/1A05g8YONj2YZ7XdoJMA9p3rVsN4OsvCx`.

[2] Ting Chen et al. "A Simple Framework for Contrastive Learning of Visual Representations". In: *International conference on machine learning (ICML)*. 2020.

A Training and validation loss/accuracy curves

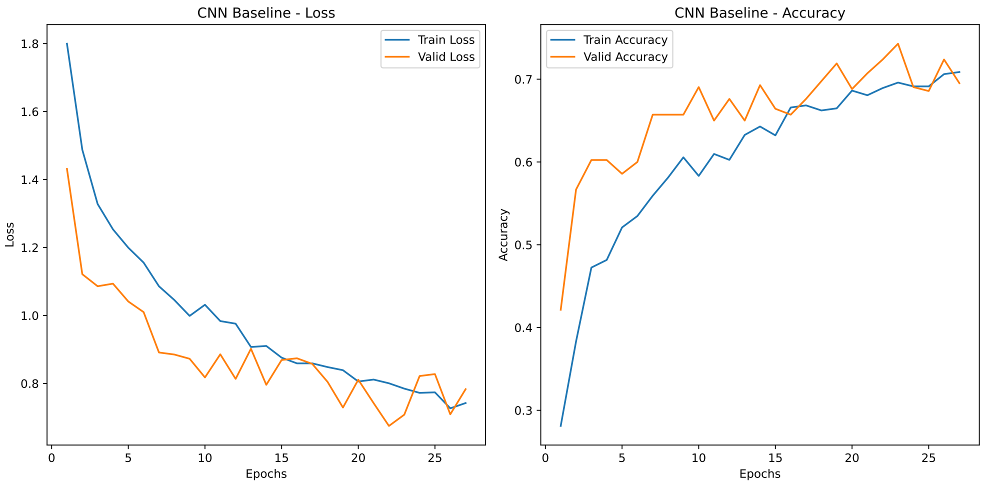

Figure 1: Training and validation loss/accuracy curves for the CNN baseline model.

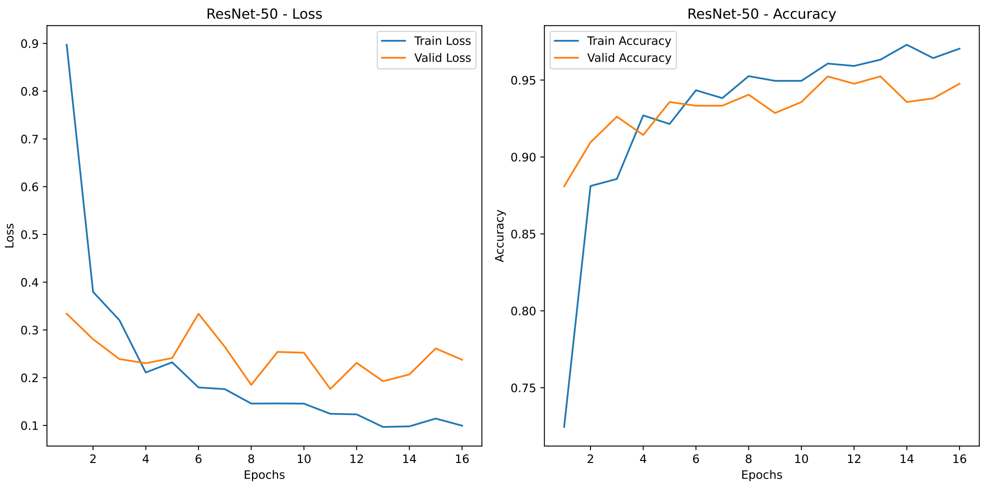

Figure 2: Training and validation loss/accuracy curves for the ResNet-50.

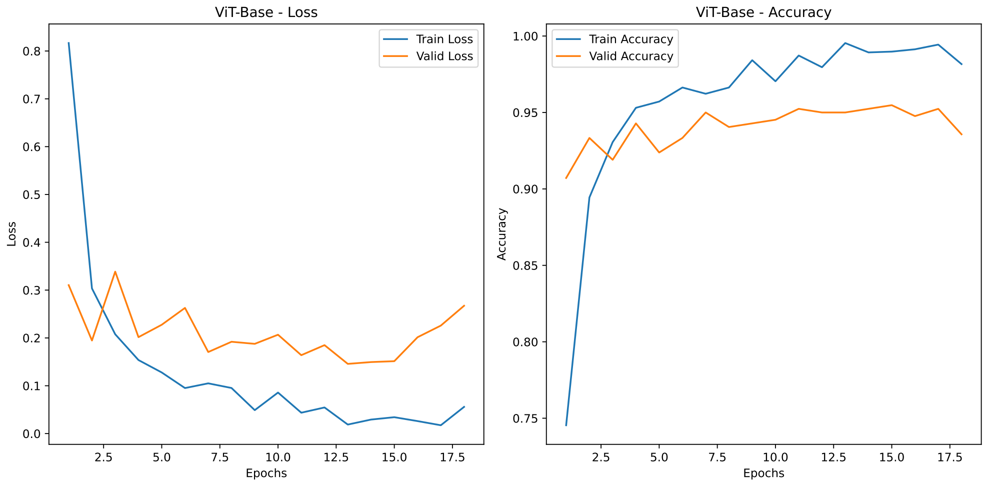

Figure 3: Training and validation loss/accuracy curves for the ViT-Base model.

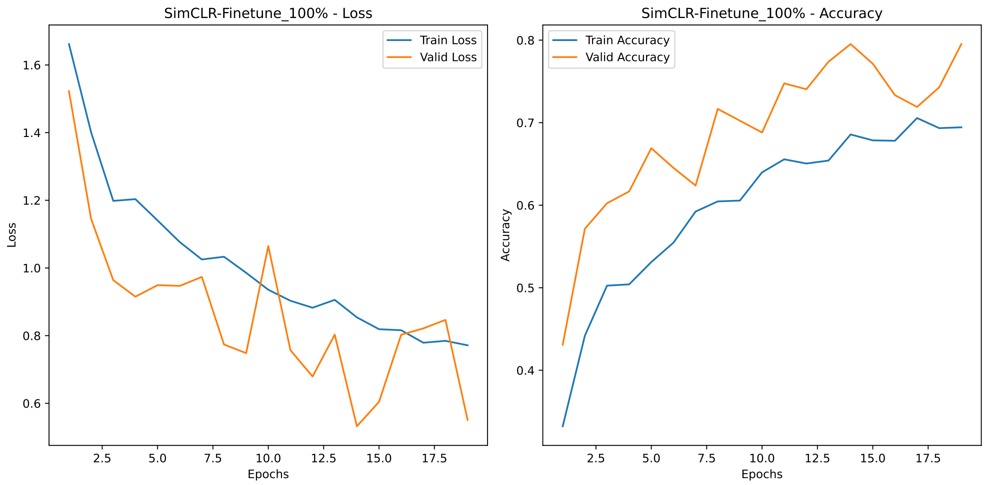

Figure 4: Training and validation loss/accuracy curves for the SimCLR-Finetune(Entrie training set).

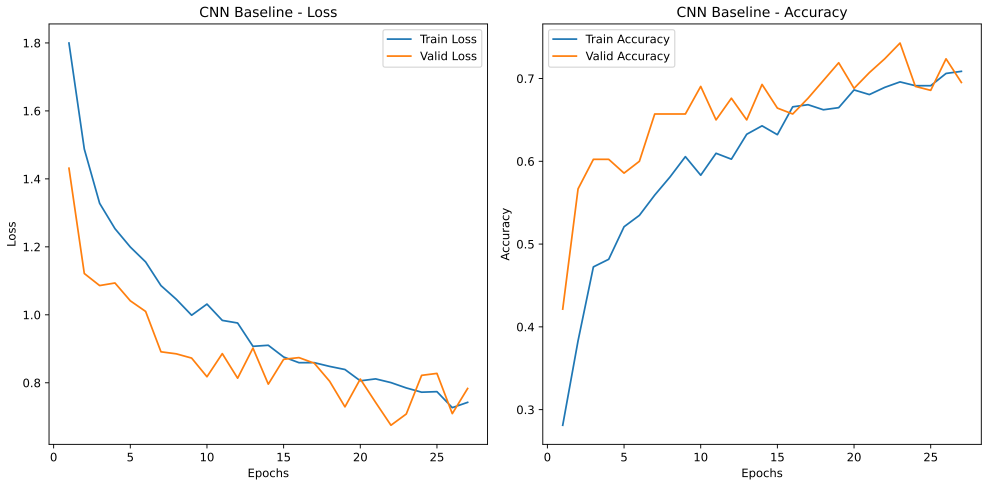

Figure 5: Training and validation loss/accuracy curves for the SimCLR-Finetune(Half of the training set).

B Confusion matrix

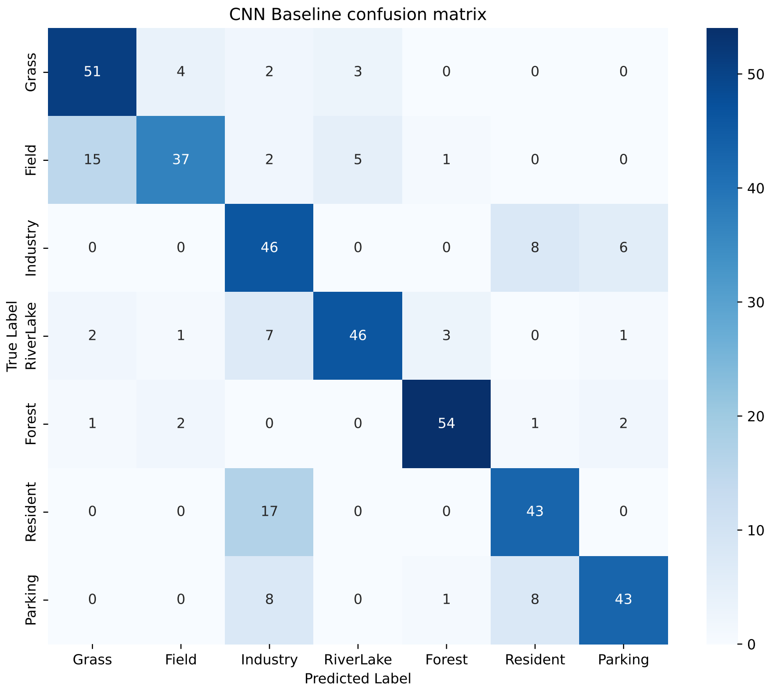

Figure 6: CNN Baseline confusion matrix

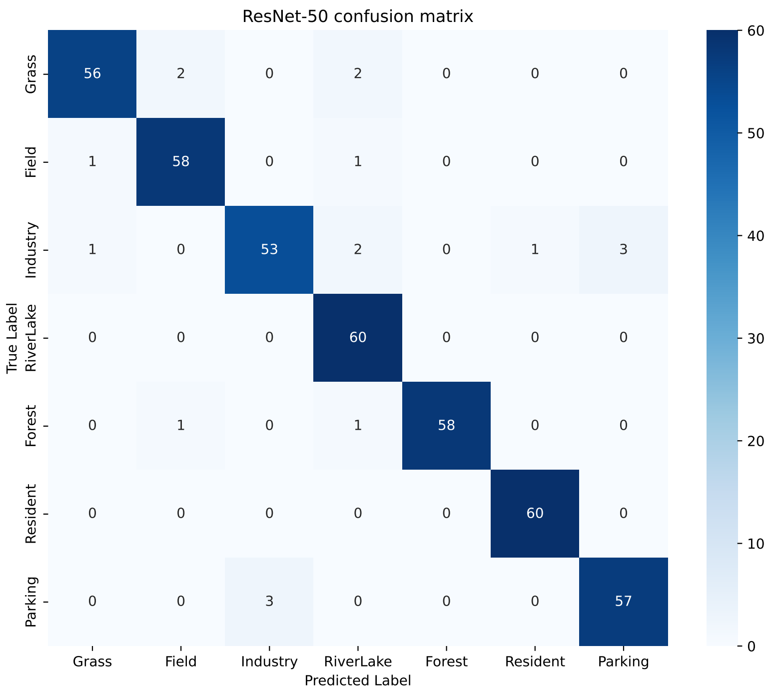

Figure 7: ResNet-50 confusion matrix

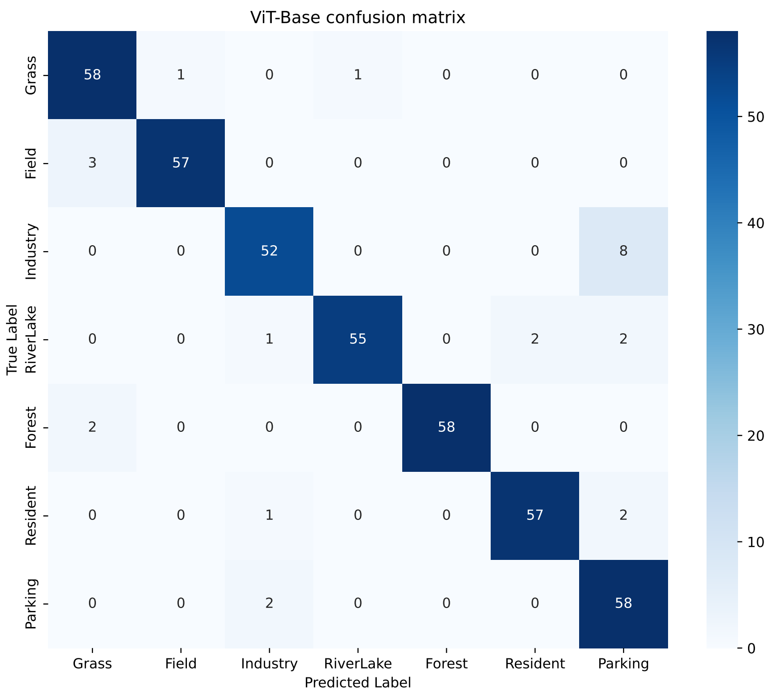

Figure 8: ViT-Base confusion matrix

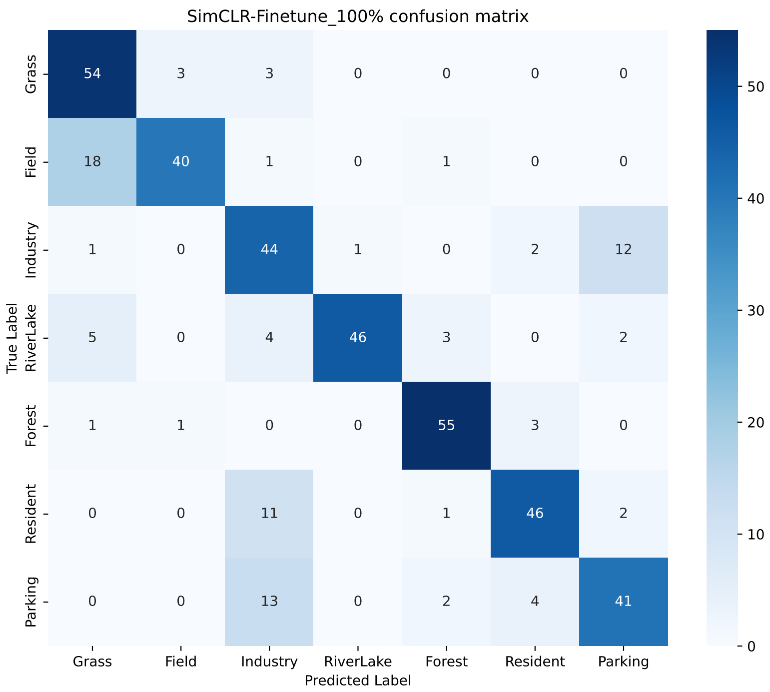

Figure 9: SimCLR-Finetune(Entrie training set) confusion matrix

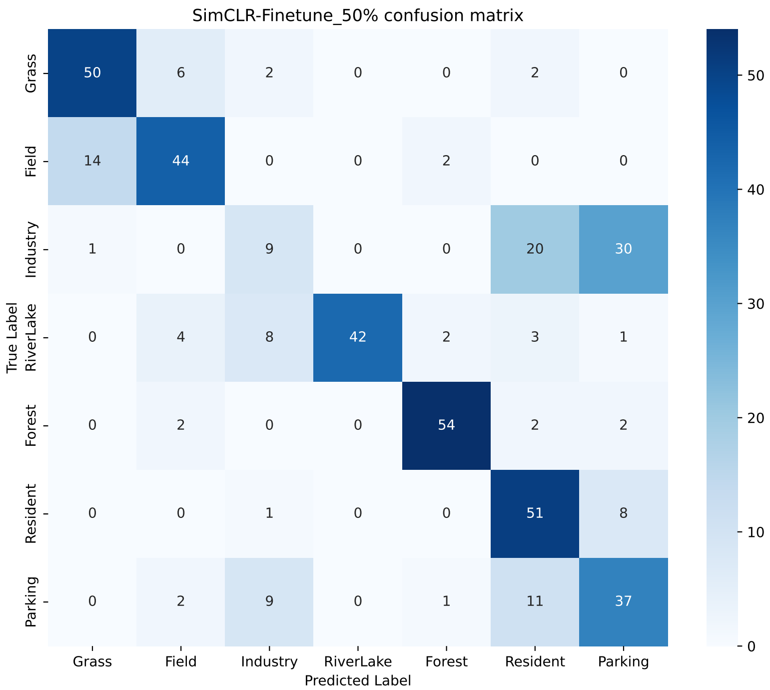

Figure 10: SimCLR-Finetune(Half of the training set) confusion matrix

C Python code

Colab code click **here**.

C.1 Data preparation

```python
import gdown

# Google Drive file ID
file_id = "1ExI4cImHyoFPL0anBBHePui5zhRWKQvj"
output_path = "RS_images_2800.rar"

# Use gdown to download
gdown.download(f"https://drive.google.com/uc?id={file_id}", output_path, quiet=False)

!apt-get install unrar
!unrar x RS_images_2800.rar -o+ # unzip


import os

dataset_path = "/content/RS_images_2800"
print(os.listdir(dataset_path))


import random
import pandas as pd


random.seed(410)

dataset_path = "./RS_images_2800"

# Split the dataset proportionally
train_ratio = 0.7 # Training set
valid_ratio = 0.15 # Validation set
test_ratio = 0.15 # Test set

# Read all categories
categories = sorted(os.listdir(dataset_path))
categories_cleaned = ["".join(c[1:]) for c in categories]
class_mapping = {c: i for i, c in enumerate(categories_cleaned)}

# Store
dataset = {"train": [], "valid": [], "test": []}

for category in categories:
    category_path = os.path.join(dataset_path, category)
    images = [os.path.join(category_path, img) for img in os.listdir(category_path)]

    # Random shuffle
    random.shuffle(images)

    # Calculate partition index
    total_count = len(images)
    train_count = int(total_count * train_ratio)
    valid_count = int(total_count * valid_ratio)

    # Split
    dataset["train"].extend([(img, category[1:], class_mapping[category[1:]]) for img
        in images[:train_count]])
```

```
    dataset["valid"].extend([(img, category[1:], class_mapping[category[1:]]) for img in images[train_count:train_count + valid_count]])
    dataset["test"].extend([(img, category[1:], class_mapping[category[1:]]) for img in images[train_count + valid_count:]])

# Convert to Pandas DataFrame
df_train = pd.DataFrame(dataset["train"], columns=["image", "class_name", "class_num"])
df_valid = pd.DataFrame(dataset["valid"], columns=["image", "class_name", "class_num"])
df_test = pd.DataFrame(dataset["test"], columns=["image", "class_name", "class_num"])

# Save
df_train.to_csv("train_data.csv", index=False)
df_valid.to_csv("valid_data.csv", index=False)
df_test.to_csv("test_data.csv", index=False)


import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
from PIL import Image
import matplotlib.pyplot as plt

# Set global random seed
seed = 410
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed_all(seed)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

# Use GPU
device = torch.device("cuda" if torch.cuda.is_available() else "CPU")


df_train = pd.read_csv("train_data.csv")
df_valid = pd.read_csv("valid_data.csv")
df_test = pd.read_csv("test_data.csv")

# Data enhancement
train_transform = transforms.Compose([
    transforms.Resize((400, 400)),
    transforms.RandomRotation(30), # Rotation
    transforms.RandomHorizontalFlip(), # Horizontal flip
    transforms.ColorJitter(brightness=0.5, contrast=0.5, saturation=0.5, hue=0.1), # Brightness change
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

valid_test_transform = transforms.Compose([
    transforms.Resize((400, 400)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

class LoadDataset(Dataset):
```

```python
    def __init__(self, dataframe, transform=None):
        self.dataframe = dataframe
        self.transform = transform

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

    def __getitem__(self, idx):
        img_path, class_name, class_num = self.dataframe.iloc[idx]
        image = Image.open(img_path).convert("RGB")
        if self.transform:
            image = self.transform(image)
        return image, class_num

# Create DataLoader
batch_size = 32

train_dataset = LoadDataset(df_train, transform=train_transform)
valid_dataset = LoadDataset(df_valid, transform=valid_test_transform)
test_dataset = LoadDataset(df_test, transform=valid_test_transform)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
valid_loader = DataLoader(valid_dataset, batch_size=batch_size, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

C.2 Basic CNN

```python
class CNNBaseline(nn.Module):
    def __init__(self, num_classes):
        super(CNNBaseline, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.fc1 = nn.Linear(128 * 50 * 50, 256) #400x400 input, after 3 MaxPool
            iterations, becomes 50x50
        self.fc2 = nn.Linear(256, num_classes)
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x


num_classes = len(df_train["class_num"].unique())

model = CNNBaseline(num_classes).to(device)



import matplotlib.pyplot as plt

def plot_training_results(train_losses, valid_losses, train_accs, valid_accs,
    model_name="Model"):
```

```
    epochs = range(1, len(train_losses) + 1)

    plt.figure(figsize=(12, 6))

    # Draw loss
    plt.subplot(1, 2, 1)
    plt.plot(epochs, train_losses, label="Train Loss")
    plt.plot(epochs, valid_losses, label="Valid Loss")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.title(f"{model_name} - Loss")
    plt.legend()

    # Draw accuracy
    plt.subplot(1, 2, 2)
    plt.plot(epochs, train_accs, label="Train Accuracy")
    plt.plot(epochs, valid_accs, label="Valid Accuracy")
    plt.xlabel("Epochs")
    plt.ylabel("Accuracy")
    plt.title(f"{model_name} - Accuracy")
    plt.legend()

    plt.tight_layout()
    plt.savefig(f"{model_name}_training_results.pdf")
    plt.show()



def train_model(model, train_loader, valid_loader, device, num_epochs=30,
     learning_rate=0.0001, early_stop_patience=5, model_name="Model"):
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    # Early stopping
    best_val_loss = float("inf")
    early_stop_counter = 0

    # Record Loss and Accuracy
    train_losses, valid_losses = [], []
    train_accs, valid_accs = [], []

    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0

        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)

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

            running_loss += loss.item()
            _, predicted = torch.max(outputs, 1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)

```

```
        train_loss = running_loss / len(train_loader)
        train_acc = correct / total
        train_losses.append(train_loss)
        train_accs.append(train_acc)

        # Validation
        model.eval()
        running_val_loss = 0.0
        correct = 0
        total = 0

        with torch.no_grad():
            for images, labels in valid_loader:
                images, labels = images.to(device), labels.to(device)
                outputs = model(images)
                loss = criterion(outputs, labels)
                running_val_loss += loss.item()

                _, predicted = torch.max(outputs, 1)
                correct += (predicted == labels).sum().item()
                total += labels.size(0)

        val_loss = running_val_loss / len(valid_loader)
        val_acc = correct / total
        valid_losses.append(val_loss)
        valid_accs.append(val_acc)

        print(f"Epoch {epoch+1}/{num_epochs}: Train Loss: {train_loss:.4f}, Train Acc:
            {train_acc:.4f}, Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f}")

        # Early stoping & Save best model
        if val_loss < best_val_loss:
            best_val_loss = val_loss
            early_stop_counter = 0
            torch.save(model.state_dict(), f"{model_name}_best.pth")
            print("Best model saved!")
        else:
            early_stop_counter += 1
            if early_stop_counter >= early_stop_patience:
                print("Early stopping triggered!")
                break

    # plot loss and acc
    plot_training_results(train_losses, valid_losses, train_accs, valid_accs,
        model_name)


train_model(model, train_loader, valid_loader, device, num_epochs=30, model_name="CNN
    Baseline")

def evaluate_model(model, test_loader, device):
    model.eval()
    correct = 0
    total = 0
    all_predictions = []
    all_labels = []

    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
```

```
            _, predicted = torch.max(outputs, 1)

            all_predictions.extend(predicted.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

            correct += (predicted == labels).sum().item()
            total += labels.size(0)

    accuracy = correct / total
    print(f"Test Accuracy: {accuracy:.4f}")

    return accuracy, all_predictions, all_labels


# Load model
best_model = CNNBaseline(num_classes).to(device)
best_model.load_state_dict(torch.load("best_model.pth"))

# Compute test accuracy
test_accuracy, predictions, labels = evaluate_model(best_model, test_loader, device)


import seaborn as sns
from sklearn.metrics import confusion_matrix

def plot_confusion_matrix(y_true, y_pred, class_names, model_name):
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=class_names,
        yticklabels=class_names)
    plt.xlabel("Predicted Label")
    plt.ylabel("True Label")
    plt.title(f"{model_name} confusion matrix")
    plt.savefig(f"{model_name}_confusion_matrix.pdf")
    plt.show()

# Draw confusion matrix
plot_confusion_matrix(labels, predictions, df_train["class_name"].unique(),
    model_name="CNN Baseline")
```

C.3 ResNet-50

```
from torchvision import models

# Load ResNet-50 pre trained model
model = models.resnet50(weights=models.ResNet50_Weights.IMAGENET1K_V1)

# Freeze the first 140 layers (only train the last few layers)
for param in list(model.parameters())[:140]:
    param.requires_grad = False

# Replace the fully connected layer of ResNet-50
num_features = model.fc.in_features
model.fc = nn.Sequential(
    nn.Linear(num_features, 256),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(256, num_classes)
)

```

```python
model = model.to(device)

train_model(model, train_loader, valid_loader, device, num_epochs=30,
    model_name="ResNet-50")

# Load best model
best_resnet_model = models.resnet50(weights=models.ResNet50_Weights.IMAGENET1K_V1)
best_resnet_model.fc = nn.Sequential(
    nn.Linear(num_features, 256),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(256, num_classes)
)
best_resnet_model.load_state_dict(torch.load("ResNet-50_best.pth"))
best_resnet_model.to(device)

# Compute test accuracy
test_accuracy, predictions, labels = evaluate_model(best_resnet_model, test_loader,
    device)

# Deaw confusion matrix
plot_confusion_matrix(labels, predictions, df_train["class_name"].unique(),
    model_name="ResNet-50")
```

C.4 ViT

```python
from transformers import ViTModel, ViTFeatureExtractor

# Load ViT feature extractor
feature_extractor = ViTFeatureExtractor.from_pretrained("google/vit-base-patch16-224")
# Load ViT model (without classification head)
class ViTClassifier(nn.Module):
    def __init__(self, num_classes=7):
        super(ViTClassifier, self).__init__()
        self.vit = ViTModel.from_pretrained("google/vit-base-patch16-224")
        self.classifier = nn.Sequential(
            nn.Linear(self.vit.config.hidden_size, 256),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(256, num_classes)
        )

    def forward(self, x):
        outputs = self.vit(x) # Extract features
        cls_token = outputs.last_hidden_state[:, 0, :] # CLS token
        return self.classifier(cls_token)

# Initialize model
model = ViTClassifier(num_classes=7).to(device)


train_transform_vit = transforms.Compose([
    transforms.Resize((224, 224)), # Vit needs 224x224 input
    transforms.RandomRotation(30),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(brightness=0.5, contrast=0.5, saturation=0.5, hue=0.1),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

```

```python
valid_test_transform_vit = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

class ViTDataset(Dataset):
    def __init__(self, dataframe, transform):
        self.dataframe = dataframe
        self.transform = transform

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

    def __getitem__(self, idx):
        img_path, class_num = self.dataframe.iloc[idx][["image", "class_num"]]

        image = Image.open(img_path).convert("RGB")

        image = self.transform(image)

        # Ensure the shape is (3, 224, 224)
        if image.shape != (3, 224, 224):
            print(f"Shape mismatch: {image.shape} at index {idx}")

        # Convert class_num to PyTorch tensor
        class_num = torch.tensor(class_num, dtype=torch.long)

        return image, class_num


train_dataset_vit = ViTDataset(df_train, transform=train_transform_vit)
valid_dataset_vit = ViTDataset(df_valid, transform=valid_test_transform_vit)
test_dataset_vit = ViTDataset(df_test, transform=valid_test_transform_vit)

train_loader_vit = DataLoader(train_dataset_vit, batch_size=batch_size, shuffle=True)
valid_loader_vit = DataLoader(valid_dataset_vit, batch_size=batch_size, shuffle=False)
test_loader_vit = DataLoader(test_dataset_vit, batch_size=batch_size, shuffle=False)


train_model(model, train_loader_vit, valid_loader_vit, device, num_epochs=30,
    model_name="ViT-Base")


# Load best model
best_vit_model = ViTClassifier(num_classes=7).to(device)
best_vit_model.load_state_dict(torch.load("ViT-Base_best.pth"))

# Compute test accuracy
test_accuracy, predictions, labels = evaluate_model(best_vit_model, test_loader_vit,
    device)

# Draw confusion matrix
plot_confusion_matrix(labels, predictions, df_train["class_name"].unique(),
    model_name="ViT-Base")
```

C.5 SSL

```python
contrastive_transforms = transforms.Compose([
    transforms.RandomResizedCrop(400, scale=(0.2, 1.0)),
```

```
    transforms.RandomHorizontalFlip(),
    transforms.RandomApply([transforms.ColorJitter(0.4, 0.4, 0.4, 0.1)], p=0.8),
    transforms.RandomGrayscale(p=0.2),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

def get_augmented_views(img_path):
    img = Image.open(img_path).convert("RGB")
    view1 = contrastive_transforms(img)
    view2 = contrastive_transforms(img)
    return view1, view2

class SimCLRDataset(Dataset):
    def __init__(self, df):
        self.image_paths = df["image"].values

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

    def __getitem__(self, idx):
        img_path = self.image_paths[idx]
        img1, img2 = get_augmented_views(img_path)
        return img1, img2

train_dataset_sim = SimCLRDataset(df_train)
train_loader_sim = DataLoader(train_dataset, batch_size=32, shuffle=True, num_workers=2)


from torchvision.models import ResNet50_Weights

class SimCLR(nn.Module):
    def __init__(self, feature_dim=128):
        super(SimCLR, self).__init__()
        base_model = models.resnet50(weights=ResNet50_Weights.DEFAULT)

        # Remove ResNet-50 Classification layer
        self.encoder = nn.Sequential(*list(base_model.children())[:-1])

        # Projection Head
        self.projector = nn.Sequential(
            nn.Linear(2048, 512),
            nn.ReLU(),
            nn.Linear(512, feature_dim)
        )

    def forward(self, x):
        h = self.encoder(x).squeeze()
        z = self.projector(h)
        return z


model = SimCLR().cuda()


def nt_xent_loss(z1, z2, temperature=0.3):
    z1 = F.normalize(z1, dim=1)
    z2 = F.normalize(z2, dim=1)
    logits = torch.matmul(z1, z2.T) / temperature
    labels = torch.arange(z1.shape[0]).cuda()
    loss = F.cross_entropy(logits, labels)
```

```
    return loss


import gc
import time
import json
from torch.amp import autocast, GradScaler

# Record time
start_time = time.time()

# Record training history
history = {"train_loss": [], "learning_rate": []}


scaler = GradScaler(device="cuda")
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=50, gamma=0.5)

best_train_loss = float("inf")
patience = 5
early_stop_counter = 0


for epoch in range(30):
    total_loss = 0
    model.train()

    epoch_start = time.time()

    for img1, img2 in train_loader:
        img1, img2 = img1.cuda(), img2.cuda()

        optimizer.zero_grad()

        with autocast(device_type="cuda"):
            z1, z2 = model(img1), model(img2)
            loss = nt_xent_loss(z1, z2)

        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()

        total_loss += loss.item()

        del img1, img2, z1, z2, loss
        torch.cuda.empty_cache()
        gc.collect()

    avg_train_loss = total_loss / len(train_loader)
    history["train_loss"].append(avg_train_loss)
    history["learning_rate"].append(optimizer.param_groups[0]['lr'])

    # update learning rate
    scheduler.step()

    # Compute training time
    epoch_time = time.time() - epoch_start

    print(f"Epoch [{epoch+1}/30] | Train Loss: {avg_train_loss:.4f} | LR:
        {optimizer.param_groups[0]['lr']:.6f} | Time: {epoch_time:.2f}s")
```

```python

    # Save best model
    if avg_train_loss < best_train_loss:
        best_train_loss = avg_train_loss
        torch.save(model.encoder.state_dict(), "simclr_best_model.pth")
        early_stop_counter = 0
    else:
        early_stop_counter += 1

    # Early Stopping
    if early_stop_counter >= patience:
        break


#=========================================================
# Fine-tuning
#=========================================================
def balanced_sample(df, frac=0.5, random_state=410):
    return df.groupby("class_name", group_keys=False).apply(lambda x:
        x.sample(frac=frac, random_state=random_state)).reset_index(drop=True)

df_train_sample = balanced_sample(df_train)

# Check
print(df_train["class_name"].value_counts())
print(df_train_sample["class_name"].value_counts())


train_dataset_full = LoadDataset(df_train, transform=train_transform)
train_dataset_sample = LoadDataset(df_train_sample, transform=train_transform)
valid_dataset = LoadDataset(df_valid, transform=valid_test_transform)
test_dataset = LoadDataset(df_test, transform=valid_test_transform)

train_loader_full = DataLoader(train_dataset_full, batch_size=batch_size, shuffle=True)
train_loader_sample = DataLoader(train_dataset_sample, batch_size=batch_size,
    shuffle=True)
valid_loader = DataLoader(valid_dataset, batch_size=batch_size, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)



# Load ResNet-50 pre trained with SimCLR
simclr_model = models.resnet50()
num_features = simclr_model.fc.in_features
simclr_model.fc = nn.Identity() # Remove the projection head
simclr_model.load_state_dict(torch.load("simclr_best_model.pth"), strict=False)
simclr_model = simclr_model.cuda()


# Add classification head
simclr_model.fc = nn.Sequential(
    nn.Linear(num_features, 256),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(256, num_classes)
)

simclr_model = simclr_model.to(device)

# Full training set
```

```
train_model(simclr_model, train_loader_full, valid_loader, device, num_epochs=30,
     model_name="SimCLR-Finetune_100%")


# Load the best model
best_simclr_model = models.resnet50()
best_simclr_model.fc = nn.Sequential(
    nn.Linear(num_features, 256),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(256, num_classes)
)
best_simclr_model.load_state_dict(torch.load("SimCLR-Finetune_100%_best.pth"))
best_simclr_model.to(device)

# Compute test accuracy
test_accuracy, predictions, labels = evaluate_model(best_simclr_model, test_loader,
     device)

# Draw confusion matrix
plot_confusion_matrix(labels, predictions, df_train["class_name"].unique(),
     model_name="SimCLR-Finetune_100%")

#----------------------------------------------------
# Load ResNet-50 pre trained with SimCLR
simclr_model = models.resnet50()
num_features = simclr_model.fc.in_features
simclr_model.fc = nn.Identity() # Remove the projection head
simclr_model.load_state_dict(torch.load("simclr_best_model.pth"), strict=False)
simclr_model = simclr_model.cuda()

# Add classification head
simclr_model.fc = nn.Sequential(
    nn.Linear(num_features, 256),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(256, num_classes)
)

simclr_model = simclr_model.to(device)

# Half of training set
train_model(simclr_model, train_loader_sample, valid_loader, device, num_epochs=30,
     model_name="SimCLR-Finetune_50%")


# Load the best model
best_simclr_model = models.resnet50()
best_simclr_model.fc = nn.Sequential(
    nn.Linear(num_features, 256),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(256, num_classes)
)
best_simclr_model.load_state_dict(torch.load("SimCLR-Finetune_50%_best.pth"))
best_simclr_model.to(device)

# Compute test accuracy
test_accuracy, predictions, labels = evaluate_model(best_simclr_model, test_loader,
     device)

```

```
# Draw confusion matrix
plot_confusion_matrix(labels, predictions, df_train["class_name"].unique(),
    model_name="SimCLR-Finetune_50%")
```