1.0 Problem Definition and Dataset Selection

1.1 Problem Definition

Image classification is a fundamental task in computer vision with widespread applications, including scene recognition, surveillance, and autonomous navigation. This project addresses the challenge of **automatically classifying natural scenes into six categories**: buildings, forests, glaciers, mountains, sea, and street. Manual classification of such images can be time-consuming and inconsistent, especially when dealing with large volumes of visual data. A deep learning-based classification model can provide efficient and scalable solutions for automated image tagging and analysis.

Scene image classification plays a vital role across various domains, including environmental monitoring, urban planning, and content-based image retrieval. From my professional experience, non-profit organizations such as Earthworm Foundation, Global Forest Watch, and the World Wide Fund for Nature often depend on manual interpretation of satellite imagery for monitoring purposes. While this method can be effective, it is labor-intensive and prone to human error, underscoring the need for automated and scalable approaches. This study aims to evaluate multiple deep learning models to determine the most effective architecture for accurately distinguishing between different scene types based on their visual features.

1.2 Dataset Selection

The **Intel Image Classification dataset**, available on Kaggle, has been selected for this study. It contains over **25,000 labeled RGB images**, each resized to 150×150 pixels, categorized into six scene classes: buildings, forests, glaciers, mountains, sea, and street. The dataset is organized into separate training, testing, and prediction sets, making it convenient for supervised learning tasks.

This dataset is well-suited for the problem due to the following reasons:

- It provides a balanced distribution of images across all six classes, minimizing the risk of class imbalance.
- The image categories represent a variety of natural and urban environments, making the classification task non-trivial and realistic.
- The resolution and size of the images are compatible with most convolutional neural network (CNN) architectures without the need for extensive computational resources.
- The dataset's relevance to real-world applications enhances the practical value of the model evaluation.

Overall, the Intel dataset offers a reliable and well-structured benchmark for developing and comparing deep learning models for scene classification.

In [1]:
```
import os
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
```

```
C:\Users\user\anaconda3\envs\rl-assignment\lib\site-packages\tqdm\auto.py:21: TqdmWarning: IProgr
ess not found. Please update jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/sta
ble/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

In [62]:
```
# Define the root directories
train_dir = r"C:\Users\user\OneDrive - Universiti Teknologi Malaysia (UTM)\MRTB2153 Advanced Art
test_dir = r"C:\Users\user\OneDrive - Universiti Teknologi Malaysia (UTM)\MRTB2153 Advanced Arti

# Define image transformations
transform_train = transforms.Compose([
    transforms.Resize((150, 150)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(15),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])

transform_test = transforms.Compose([
    transforms.Resize((150, 150)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])

# Load datasets using ImageFolder
train_dataset = datasets.ImageFolder(root=train_dir, transform=transform_train)
test_dataset = datasets.ImageFolder(root=test_dir, transform=transform_test)

# Create DataLoaders
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

# Get class names
class_names = train_dataset.classes
print(f"Class names: {class_names}")
```

```
Class names: ['buildings', 'forest', 'glacier', 'mountain', 'sea', 'street']
```

In [4]:
```
import matplotlib.pyplot as plt
import numpy as np
```

In [5]:
```
# Helper to unnormalize and show image
def imshow(img):
    img = img.numpy().transpose((1, 2, 0))
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    img = std * img + mean  # unnormalize
    img = np.clip(img, 0, 1)
    plt.imshow(img)
    plt.axis('off')
    plt.show()

# Display a batch of training images
dataiter = iter(train_loader)
images, labels = next(dataiter)

print(f"Image tensor shape: {images.shape}")  # [batch_size, channels, height, width]
```

```
imshow(images[0])  # Show first image in the batch
print(f"Label: {class_names[labels[0]]}")
```

```
Image tensor shape: torch.Size([32, 3, 150, 150])
```

```
Label: forest
```

2.0 Methodology

This section outlines the end-to-end approach used to develop, tune, and evaluate image classification models for the Intel Image Classification dataset. The workflow is divided into several key stages:

2.1 Data Preprocessing

The dataset used is the **Intel Image Classification** dataset, consisting of six classes: `buildings`, `forest`, `glacier`, `mountain`, `sea`, and `street`. It is split into three folders:

- `seg_train`: Labelled images used for training and validation
- `seg_test`: Labelled images used to evaluate final model performance
- `seg_pred`: Unlabelled images used for inference

The following preprocessing steps were applied:

- **Resize**: All images were resized to 150x150 pixels.
- **Normalization**: Pixel values were normalized using ImageNet mean and standard deviation.
- **Augmentation** (training set only): Horizontal flipping and small random rotations were applied to improve generalization.

2.2 Model Selection

Three architectures were implemented and compared:

- **Simple CNN**: A custom convolutional neural network with tunable dropout.
- **ResNet50**: A pre-trained residual network from PyTorch's `torchvision.models`.
- **EfficientNet-B0**: A lightweight yet powerful image classification model.

2.3 Hyperparameter Tuning with Optuna

To optimize each model's performance, **Optuna** was used for automated hyperparameter tuning. For each model, a search space was defined, and trials were run to identify the best combination of:

- Learning rate (`lr`)
- Weight decay
- Dropout rate (for CNN)
- Optimizer (`Adam` or `SGD`)
- Batch size
- Momentum (for SGD-based optimizers)

The objective function was based on validation accuracy, and trials were tracked using Optuna's `study.optimize`.

2.4 Training Procedure

For each model, the following training pipeline was followed:

- **Loss Function**: CrossEntropyLoss was used for multi-class classification.
- **Optimizers**: Optimizer selection (Adam or SGD) was based on Optuna tuning.
- **Epochs**: All models were trained for 10 epochs.
- **Device**: Training was conducted on GPU.

Additionally:

- For **ResNet50** and **EfficientNet-B0**, feature extraction was used by freezing the pretrained base layers and fine-tuning the classifier head.
- For **CNN**, dropout was applied to prevent overfitting.

2.5 Evaluation and Inference

- **Metrics**: Models were evaluated using precision, recall, accuracy, F1-score and confusion matrices on the `seg_test` dataset.
- **Loss Curves**: Training and validation loss were plotted to monitor learning behavior.
- **Class Distribution Analysis**: Predictions on the unlabelled `seg_pred` images were analyzed for each model to assess class balance and visual accuracy.
- **Qualitative Analysis**: Sample predictions were manually reviewed to identify strengths and weaknesses in classifying confusing scenes (urban or snowy landscapes).

3.0 Data Preprocessing and Exploration

3.1 Exploratory Data Analysis (EDA)

To begin the analysis, the dataset was explored to understand its structure, class distribution, and image properties. The training dataset (`seg_train`) and testing dataset (`seg_test`) are organized into six subfolders representing the classes: **buildings**, **forest**, **glacier**, **mountain**, **sea**, and **street**. This folder structure is compatible with PyTorch's `ImageFolder` for supervised learning tasks.

A bar chart of image counts per class reveals a **balanced dataset**, with no significant class imbalance. Sample images were also visualized to observe the variety in visual textures, lighting conditions, and scene content. All images are in color (RGB) and of manageable resolution, enabling efficient training on most deep learning models without resizing overhead.

Class Distribution Analysis

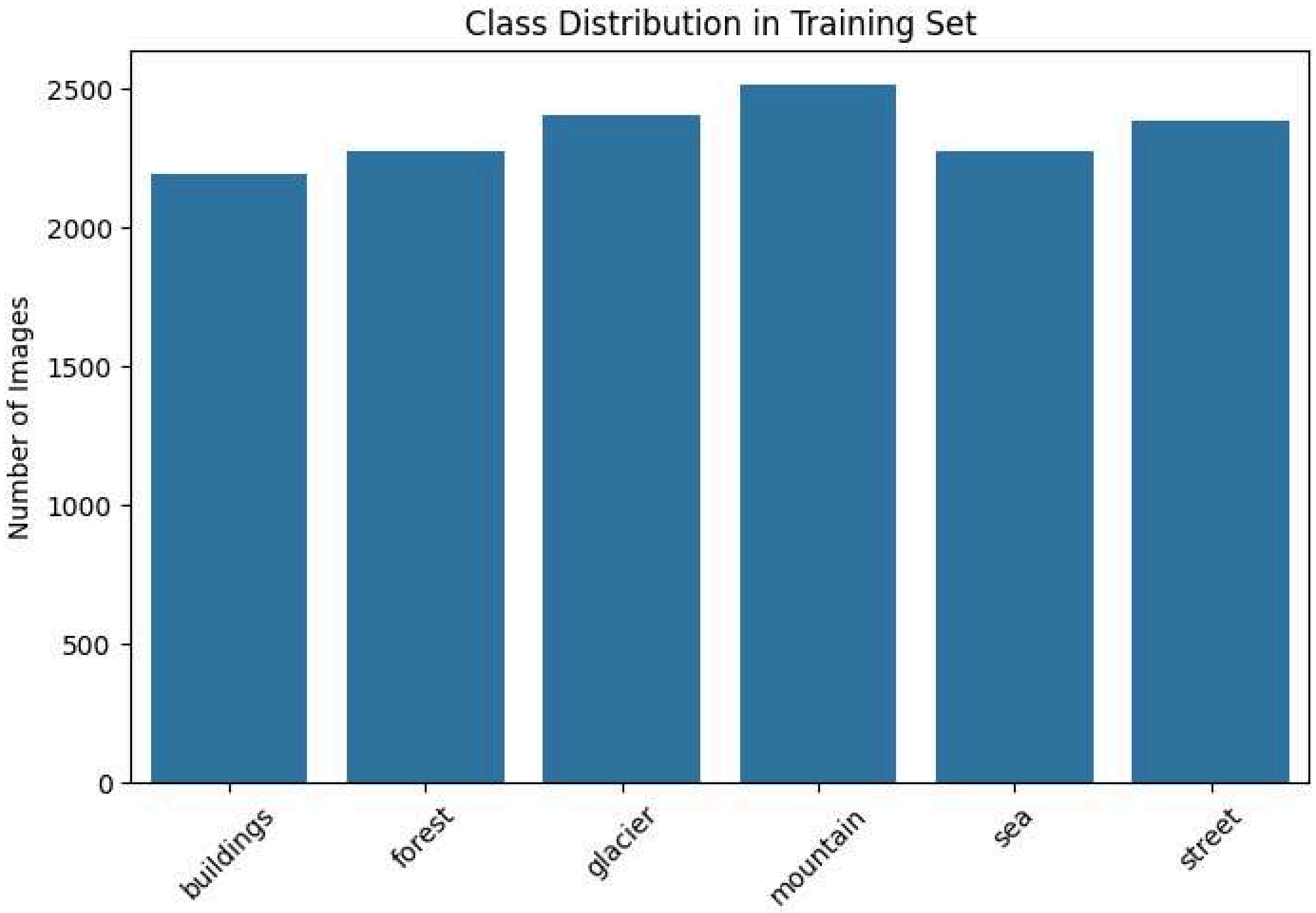
The bar chart reveals a balanced dataset with all six classes (`buildings` , `forest` , `glacier` , `mountain` , `sea` , and `street`) containing between 2200 and 2500 images each. This eliminates the need for additional sampling strategies and allows the model to train fairly across categories.

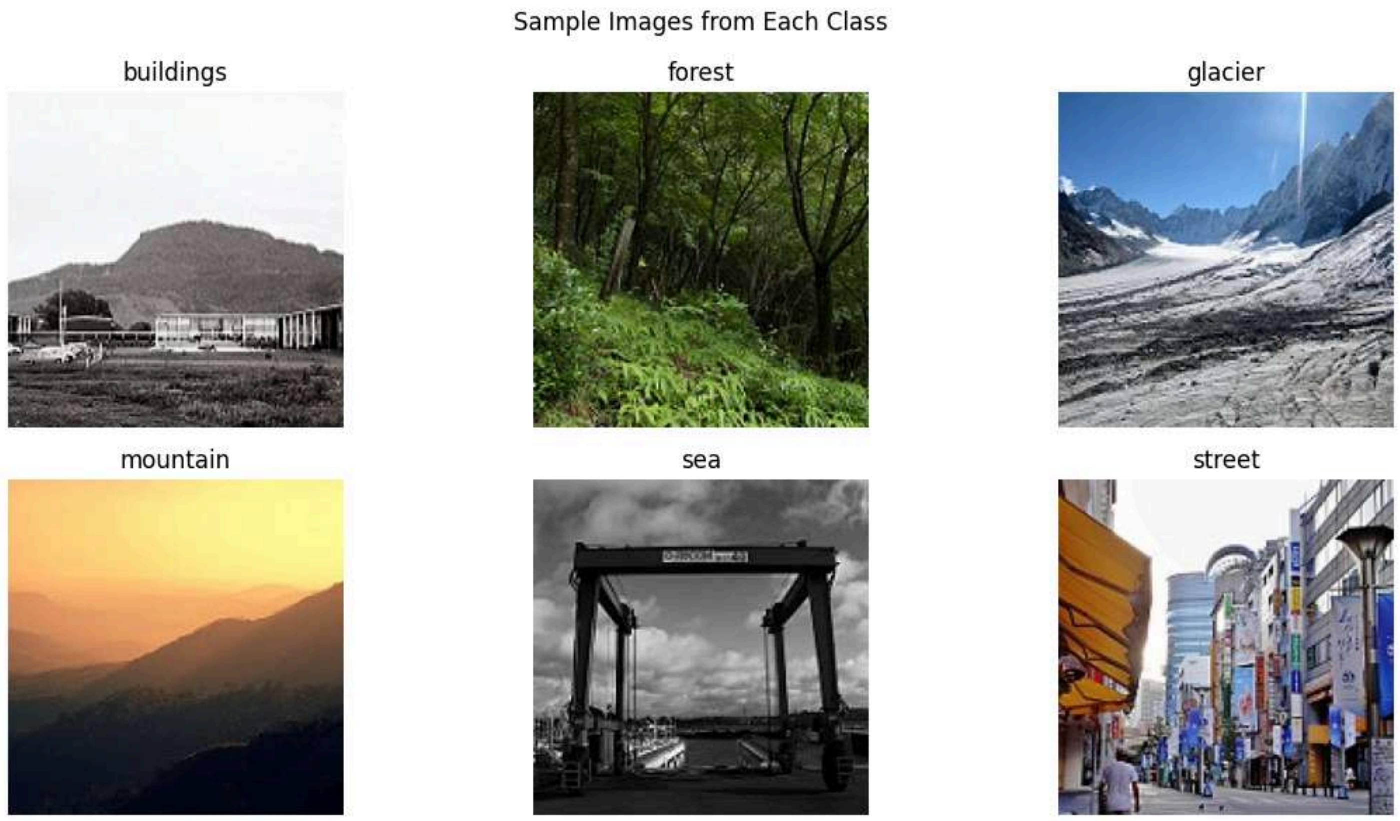
Sample Image Inspection

Sample images from each class show meaningful variation in visual features such as texture, color, and structure. For example, forests exhibit dense greenery, while glaciers feature high-contrast icy terrain. This diversity supports the relevance and complexity of the classification task, indicating that the model will need to extract meaningful spatial and contextual features to distinguish between categories.

In [10]:
```python
import seaborn as sns
```

In [11]:
```python
# Count images per class
class_counts = {cls: len(os.listdir(os.path.join(train_dir, cls))) for cls in os.listdir(train_d
plt.figure(figsize=(8, 5))
sns.barplot(x=list(class_counts.keys()), y=list(class_counts.values()))
plt.title("Class Distribution in Training Set")
plt.ylabel("Number of Images")
plt.xticks(rotation=45)
plt.show()

# Show sample images from each class
fig, axs = plt.subplots(2, 3, figsize=(12, 6))
for idx, cls in enumerate(class_counts.keys()):
    img_path = os.path.join(train_dir, cls, os.listdir(os.path.join(train_dir, cls))[0])
    img = Image.open(img_path)
    axs[idx//3, idx%3].imshow(img)
    axs[idx//3, idx%3].set_title(cls)
    axs[idx//3, idx%3].axis('off')
plt.suptitle("Sample Images from Each Class")
plt.tight_layout()
plt.show()
```





3.2 Preprocessing Steps

To prepare the data for training deep learning models, the following preprocessing steps were applied:

- `Resize`: All images were resized to 150×150 pixels to ensure consistent input dimensions across the model.
- `Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]`: Pixel values were normalized using ImageNet mean and standard deviation to accelerate convergence during training and ensure numerical stability. This ensures the input distribution matches what many pretrained CNN models expect, facilitating more stable and effective training.
- **Data Augmentation** via `RandomHorizontalFlip()` and `RandomRotation(15)`: Random horizontal flipping and slight rotation were introduced in the training pipeline to improve generalization and reduce overfitting. The `RandomRotation(15)` transformation introduces a small degree of variation (±15°) in the training images, helping the model become more robust to slight changes in camera angle or perspective. The 15° range is large enough to promote generalization, yet small enough to preserve the original scene semantics.
- **Tensor Conversion** using `ToTensor()`: Images were converted from PIL format to PyTorch tensors for model consumption.
- **Dataset Splitting**: The dataset was already separated into training (`seg_train`), testing (`seg_test`), and prediction (`seg_pred`) folders.

In [60]:
```
transform_train
```

Out[60]:
```
Compose(
    Resize(size=(150, 150), interpolation=bilinear, max_size=None, antialias=warn)
    RandomHorizontalFlip(p=0.5)
    RandomRotation(degrees=[-15.0, 15.0], interpolation=nearest, expand=False, fill=0)
    ToTensor()
    Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
)
```

In [63]:
```
transform_test
```

Out[63]:
```
Compose(
    Resize(size=(150, 150), interpolation=bilinear, max_size=None, antialias=warn)
    ToTensor()
    Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
)
```

3.3 Handling Data Imbalance or Missing Data

A check was conducted to ensure the dataset was not affected by class imbalance. Each of the six scene categories contained a comparable number of images, making it suitable for training without additional oversampling or undersampling techniques.

Additionally, the dataset was scanned for **corrupted or unreadable image files**. No such images were found. A try-except block was implemented during data loading to verify image integrity.

In [7]:
```
from PIL import Image
```

In [8]:
```
corrupted = []
for cls in os.listdir(train_dir):
    for img_file in os.listdir(os.path.join(train_dir, cls)):
        try:
            img = Image.open(os.path.join(train_dir, cls, img_file))
            img.verify()
        except:
            corrupted.append((cls, img_file))

print(f"Corrupted images found: {len(corrupted)}")
```

```
Corrupted images found: 0
```

3.4 Justification of Preprocessing Steps

- **Resizing** ensures all images have the same input shape, which is essential for convolutional neural networks (CNNs).
- **Normalization** helps stabilize and speed up the training process by bringing pixel values to a standard range.
- **Augmentation techniques** such as random flipping and rotation introduce variability into the training set, improving the model's ability to generalize to unseen data.
- **Data splitting** into training and test sets allows the model to be evaluated fairly on unseen examples.
- Checking for **missing/corrupt images** prevents training failures due to unreadable files.

These preprocessing decisions align with best practices for image classification using deep learning models.

4.0 Model Implementation

4.1 Model Selection

For the Intel Image Classification task, the following three models were selected based on peer-reviewed literature and their effectiveness in scene classification:

4.1.1. Convolutional Neural Networks (CNNs)

CNNs form the foundation of modern image classification. A custom CNN built from scratch provides insights into how deep models learn hierarchical visual features (edges, textures, patterns) directly from raw image data (Zhou et al., 2014).

- CNNs are widely used in remote sensing and scene classification tasks.
- They automatically extract translation-invariant features without manual feature engineering.
- Useful as a lightweight baseline model for understanding the effects of architectural depth and complexity.

Justification: CNNs are a core building block for visual scene understanding, and their relevance in scene classification is well-established in recent research.

4.1.2. ResNet (Residual Networks)

ResNet introduces skip connections to solve vanishing gradients and enables effective training of very deep networks (Shabbir et al., 2021).

- ResNet50, pretrained on ImageNet, is widely adopted in scene and satellite image classification.
- Fine-tuning allows leveraging high-level features learned from millions of images, accelerating training and boosting accuracy.
- ResNet has achieved 92%–99.6% accuracy on related scene image datasets.

Justification: ResNet50 is a proven architecture in scene classification literature due to its robust performance and excellent transfer learning capabilities.

4.1.3. EfficientNet

EfficientNet uses compound scaling to balance depth, width, and resolution, achieving high accuracy with fewer parameters and less computation (Tan et al., 2021; Jain et al., 2024).

- EfficientNet achieved an impressive accuracy rate of 94.31% across the dataset.
- EfficientNet has been recognised for reaching "new highs in the efficiency and accuracy of model architecture".
- The compound scaling strategy enables high performance with fewer parameters.

Justification: EfficientNet offers state-of-the-art performance with high computational efficiency, making it ideal for large-scale image scene classification under resource constraints.

The combination of:

- A **custom CNN** for foundational understanding,
- A **ResNet50** for deep residual learning and transfer learning performance,
- An **EfficientNet** for state-of-the-art efficiency and accuracy

provides a diverse and well-justified model selection strategy for benchmarking scene classification performance on the Intel dataset.

4.1.4 Architectural Comparison: CNN vs ResNet vs EfficientNet

Feature	CNN	ResNet	EfficientNet
Core Idea	Basic deep learning model for visual data	Adds residual connections to train deeper models	Uses compound scaling for optimal accuracy/efficiency
Main Innovation	Hierarchical convolution + pooling layers	Skip connections (Residual Blocks)	Compound scaling of depth, width, and resolution
Problem Addressed	Feature extraction and classification	Vanishing gradients in deep networks	Balancing model size and efficiency
Structure	Conv → Pool → FC → Softmax	Residual blocks with identity mappings	MBConv + SE blocks + compound scaling
Gradient Flow	Gradients pass through each layer sequentially	Skip connections ease gradient flow	Optimized baseline ensures stable scaling
Scalability	Manually add layers	Easily scales with deeper blocks	Automatically scales all dimensions (d, w, r)

Feature	CNN	ResNet	EfficientNet
Model Examples	Custom CNN, LeNet, AlexNet	ResNet18, ResNet50, ResNet101	EfficientNet-B0 to B7
Pretraining Use	Optional	Common in transfer learning (ImageNet)	Designed for pretrained efficiency
Purpose	Foundational model for image tasks	Enables very deep networks without degradation	Achieves high accuracy with fewer parameters

References:

- Jain, R., Sharma, R., Tiwari, D., Joshi, K., & Jain, V. (2024). Enhanced Classification of Intel Images Using Refined EfficientNet and MobileNetV2 Frameworks. In 2023 4th International Conference on Intelligent Technologies (CONIT) (pp. 1-4). IEEE.
- Shabbir, A., Ali, N., Ahmed, J., Zafar, B., Rasheed, A., Sajid, M., ... & Dar, S. H. (2021). Satellite and scene image classification based on transfer learning and fine tuning of ResNet50. Mathematical Problems in Engineering, 2021(1), 5843816.
- Tan, M., & Le, Q. (2019, May). Efficientnet: Rethinking model scaling for convolutional neural networks. In International conference on machine learning (pp. 6105-6114). PMLR.
- Zhou, B., Lapedriza, A., Xiao, J., Torralba, A., & Oliva, A. (2014). Learning deep features for scene recognition using places database. Advances in neural information processing systems, 27.

4.2 Model Implementation & Training

In [13]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import models
```

4.2.1 Model Definition

1. CNN

The `SimpleCNN` model is a custom convolutional neural network designed for image classification with 6 output classes. It consists of three main convolutional blocks followed by fully connected layers:

- **Convolutional Layers (`conv1`, `conv2`, `conv3`)**: Each layer applies 2D convolution with ReLU activation and padding to preserve spatial dimensions. These layers progressively learn low- to high-level visual features.
- **Max Pooling (`pool`)**: After each convolution, max pooling with a 2×2 kernel downsamples the feature maps, reducing spatial dimensions and computational cost.
- **Dropout**: A dropout layer is applied before the final classification layer to prevent overfitting by randomly deactivating neurons during training.
- **Fully Connected Layers (`fc1`, `fc2`)**:
  - `fc1`: Flattens and reduces features from the last convolutional block to 256 neurons.
  - `fc2`: Maps the 256 features to 6 output classes (scene labels).

This architecture is relatively shallow and efficient, making it suitable as a baseline model for image classification tasks.

In [18]:
```
class SimpleCNN(nn.Module):
    def __init__(self, num_classes=6, dropout_rate=0.5):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.dropout = nn.Dropout(dropout_rate)
        self.fc1 = nn.Linear(128 * 18 * 18, 256)
        self.fc2 = nn.Linear(256, num_classes)

    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 = x.view(-1, 128 * 18 * 18)
        x = self.dropout(F.relu(self.fc1(x)))
        x = self.fc2(x)
        return x
```

2. ResNet50

The `build_resnet50()` function loads a pretrained ResNet50 model and adapts it for the Intel scene classification task (6 classes):

- **Pretrained Weights**: The model uses weights pretrained on ImageNet, enabling it to leverage learned visual features.
- **Feature Freezing**: All layers are frozen to retain the pretrained feature extractor and reduce training time.
- **Classifier Replacement**: The original fully connected layer (`model.fc`) is replaced with a new linear layer with 6 output units to match the number of scene categories.

This transfer learning approach allows the model to generalize well to new image classification tasks with limited training data.

In [15]:
```
def build_resnet50(num_classes=6):
    model = models.resnet50(pretrained=True)

    # Freeze earlier layers (optional)
    for param in model.parameters():
        param.requires_grad = False

    # Replace the classifier
    in_features = model.fc.in_features
    model.fc = nn.Linear(in_features, num_classes)
    return model
```

3. EfficientNet-B0

The `build_efficientnet_b0()` function loads a pretrained EfficientNet-B0 model and customizes it for the 6-class scene classification task:

- **Pretrained Weights**: The model is initialized with weights pretrained on ImageNet, which helps transfer rich feature representations to the new task.
- **Feature Extractor Freezing**: All convolutional feature extraction layers are frozen to preserve learned representations and reduce training time.
- **Classifier Replacement**: The final classification layer is replaced with a new fully connected layer with 6 output neurons to match the number of scene categories.

EfficientNet-B0 is known for its high accuracy and computational efficiency, making it a strong candidate for transfer learning in image classification.

In [16]:
```
def build_efficientnet_b0(num_classes=6):
    model = models.efficientnet_b0(pretrained=True)

    # Freeze feature extractor
    for param in model.features.parameters():
        param.requires_grad = False

    # Replace classifier
    in_features = model.classifier[1].in_features
    model.classifier[1] = nn.Linear(in_features, num_classes)
    return model
```

4.2.2 Hyperparameter Tuning Strategy

`train_dataset` is split into two parts: 80% train and 20% validation using `random_split`.

In [23]:
```
from torch.utils.data import random_split

# Split train_dataset into train + val
train_size = int(0.8 * len(train_dataset))
val_size = len(train_dataset) - train_size
train_dataset_split, val_dataset = random_split(train_dataset, [train_size, val_size])

# Create loaders
train_loader = DataLoader(train_dataset_split, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
```

Optuna Objective Function

The `objective()` function defines the hyperparameter tuning process for Optuna. This function enables automated tuning across different architectures to identify the best-performing configuration. It supports three model types: `SimpleCNN`, `ResNet50`, and `EfficientNet-B0`. The key components are:

- **Hyperparameter Sampling**: Hyperparameter tuning was conducted using Optuna for all three models: **CNN**, **ResNet50**, and **EfficientNet-B0**.
    - For **ResNet50** and **EfficientNet-B0**, two hyperparameters were tuned:
        - `learning_rate`
        - `optimizer`

- For the **CNN model**, an additional `dropout` parameter was included to help regularize and reduce overfitting, making it three tunable hyperparameters in total.

The best *1st trial* results for each model are as follows:

Best CNN Trial 1 :

- **Accuracy**: `0.8297`
- **Parameters**:
  - `lr` : `0.000609`
  - `optimizer` : `Adam`
  - `dropout` : `0.5697`

Best ResNet50 Trial 1 :

- **Accuracy**: `0.8828`
- **Parameters**:
  - `lr` : `0.000214`
  - `optimizer` : `Adam`

Best EfficientNet-B0 Trial 1 :

- **Accuracy**: `0.8557`
- **Parameters**:
  - `lr` : `0.001659`
  - `optimizer` : `SGD`

> To further improve the tuning process, **additional hyperparameters** such as `weight_decay` , `momentum` (for SGD), and `batch_size` were later introduced in subsequent tuning iterations:

- **Weight Decay**: Helps prevent overfitting by penalizing large weights.
- **Batch Size**: Affects gradient estimates and memory efficiency.
- **Momentum**: (For SGD only) Accelerates convergence.

- **Model Initialization**:
  - Based on the selected `model_type` , the appropriate model is created and transferred to GPU.
  - For ResNet and EfficientNet, pretrained weights are used and feature extractor layers are frozen.
  - The final classifier layers are replaced with a new layer matching the number of classes (6).

- **Training Loop**:
  - Each model is trained for 5 short epochs using the sampled hyperparameters.
  - The `CrossEntropyLoss` is used for multi-class classification.

- **Validation Evaluation**:
  - After training, the model's performance is evaluated on the validation set.
  - Validation accuracy is returned as the objective to be maximized by Optuna.

In [19]:
```python
import optuna
```

In [28]:
```python
# Optuna Objective Function
def objective(trial, model_type, train_dataset, val_dataset, device):
    # Suggested hyperparameters
    lr = trial.suggest_float("lr", 1e-4, 1e-2, log=True)
    weight_decay = trial.suggest_float("weight_decay", 1e-5, 1e-3, log=True)
    batch_size = trial.suggest_categorical("batch_size", [16, 32, 64])
    optimizer_name = trial.suggest_categorical("optimizer", ["Adam", "SGD"])

    # Optional dropout (for CNN only)
    if model_type == "cnn":
        dropout = trial.suggest_float("dropout", 0.3, 0.6)
        model = SimpleCNN(dropout_rate=dropout).to(device)
    elif model_type == "resnet":
        model = build_resnet50().to(device)
    elif model_type == "efficientnet":
        model = build_efficientnet_b0().to(device)

    # Optimizer setup
    if optimizer_name == "Adam":
        optimizer = torch.optim.Adam(model.parameters(), lr=lr, weight_decay=weight_decay)
    else:
        momentum = trial.suggest_float("momentum", 0.8, 0.99)
        optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=momentum, weight_decay=we

    # Data loaders with variable batch size
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
    val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)

    # Training loop (short version)
    criterion = nn.CrossEntropyLoss()
    for epoch in range(5):
        model.train()
        for inputs, labels in train_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

    # Validation accuracy
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in val_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)
            correct += (preds == labels).sum().item()
            total += labels.size(0)

    return correct / total
```

In [27]:
```python
# Run the Optuna Study for Each Model
device = torch.device("cuda")
```

```python
for model_type in ['cnn', 'resnet', 'efficientnet']:
    print(f"\n🔍 Tuning {model_type.upper()} model with Optuna...")

    study = optuna.create_study(direction="maximize")
    study.optimize(lambda trial: objective(trial, model_type, train_dataset, val_dataset, device

    print(f"🏆 Best {model_type} trial:")
    print("  Accuracy:", study.best_value)
    print("  Params:", study.best_params)
```

```
[I 2025-07-07 20:50:59,504] A new study created in memory with name: no-name-e6486aad-d976-44cd-9
7cc-e3716252c146
🔍 Tuning CNN model with Optuna...
```

```
[I 2025-07-07 20:54:13,317] Trial 0 finished with value: 0.809048806555041 and parameters: {'lr':
0.00527139450593836, 'weight_decay': 0.0008854795048194025, 'batch_size': 64, 'optimizer': 'SGD',
'dropout': 0.36386660688067474, 'momentum': 0.9513299179724435}. Best is trial 0 with value: 0.80
9048806555041.
[I 2025-07-07 20:55:33,819] Trial 1 finished with value: 0.6416102600641254 and parameters: {'l
r': 0.0002525345011057191, 'weight_decay': 0.0006159680136507453, 'batch_size': 64, 'optimizer':
'SGD', 'dropout': 0.4002333656398484, 'momentum': 0.9158190742112617}. Best is trial 0 with valu
e: 0.809048806555041.
[I 2025-07-07 20:56:55,035] Trial 2 finished with value: 0.6661916636978981 and parameters: {'l
r': 0.0009278243879406056, 'weight_decay': 0.00017405079196587312, 'batch_size': 64, 'optimizer':
'SGD', 'dropout': 0.574860375230559, 'momentum': 0.8343903737544937}. Best is trial 0 with value:
0.809048806555041.
[I 2025-07-07 20:58:15,033] Trial 3 finished with value: 0.8019237620235127 and parameters: {'l
r': 0.004399230153127017, 'weight_decay': 0.00018180054226878877, 'batch_size': 64, 'optimizer':
'SGD', 'dropout': 0.5468623463867117, 'momentum': 0.9373230905819381}. Best is trial 0 with valu
e: 0.809048806555041.
[I 2025-07-07 20:59:46,585] Trial 4 finished with value: 0.8489490559315995 and parameters: {'l
r': 0.00014380273627461281, 'weight_decay': 6.056717975434195e-05, 'batch_size': 16, 'optimizer':
'Adam', 'dropout': 0.34640433656245717}. Best is trial 4 with value: 0.8489490559315995.
[I 2025-07-07 21:01:09,793] Trial 5 finished with value: 0.17955112219451372 and parameters: {'l
r': 0.009911786626182391, 'weight_decay': 0.00036607937140662936, 'batch_size': 32, 'optimizer':
'SGD', 'dropout': 0.3054160963021719, 'momentum': 0.9792025045005662}. Best is trial 4 with valu
e: 0.8489490559315995.
[I 2025-07-07 21:02:41,192] Trial 6 finished with value: 0.4624153900961881 and parameters: {'l
r': 0.009740696709699579, 'weight_decay': 1.6642675195482235e-05, 'batch_size': 16, 'optimizer':
'Adam', 'dropout': 0.5716733536462921}. Best is trial 4 with value: 0.8489490559315995.
[I 2025-07-07 21:04:12,206] Trial 7 finished with value: 0.6316351977199858 and parameters: {'l
r': 0.0001297919357603355, 'weight_decay': 0.0007115988045286953, 'batch_size': 16, 'optimizer':
'SGD', 'dropout': 0.5945324470930805, 'momentum': 0.8205076775492061}. Best is trial 4 with valu
e: 0.8489490559315995.
[I 2025-07-07 21:05:35,498] Trial 8 finished with value: 0.6914855717848236 and parameters: {'l
r': 0.005388289576870666, 'weight_decay': 1.813456577293424e-05, 'batch_size': 32, 'optimizer':
'Adam', 'dropout': 0.5913634340219229}. Best is trial 4 with value: 0.8489490559315995.
[I 2025-07-07 21:07:06,821] Trial 9 finished with value: 0.817598859992875 and parameters: {'lr':
0.001855679118678683, 'weight_decay': 0.00015867041881960099, 'batch_size': 16, 'optimizer': 'Ada
m', 'dropout': 0.3260932934235975}. Best is trial 4 with value: 0.8489490559315995.
[I 2025-07-07 21:08:38,523] Trial 10 finished with value: 0.8778054862842892 and parameters: {'l
r': 0.0004397830540797077, 'weight_decay': 4.729585299481905e-05, 'batch_size': 16, 'optimizer':
'Adam', 'dropout': 0.469694857002729}. Best is trial 10 with value: 0.8778054862842892.
[I 2025-07-07 21:10:10,038] Trial 11 finished with value: 0.8517990737442109 and parameters: {'l
r': 0.00047621924094838353, 'weight_decay': 4.4971248128587765e-05, 'batch_size': 16, 'optimize
r': 'Adam', 'dropout': 0.4769917573583805}. Best is trial 10 with value: 0.8778054862842892.
[I 2025-07-07 21:11:41,500] Trial 12 finished with value: 0.8685429283933025 and parameters: {'l
r': 0.0005103683256011872, 'weight_decay': 4.646781722193487e-05, 'batch_size': 16, 'optimizer':
'Adam', 'dropout': 0.48739084101095176}. Best is trial 10 with value: 0.8778054862842892.
[I 2025-07-07 21:13:13,547] Trial 13 finished with value: 0.8621303883149269 and parameters: {'l
r': 0.0005658146632902774, 'weight_decay': 3.499940493166068e-05, 'batch_size': 16, 'optimizer':
'Adam', 'dropout': 0.4962685729865812}. Best is trial 10 with value: 0.8778054862842892.
[I 2025-07-07 21:14:44,661] Trial 14 finished with value: 0.853580334877093 and parameters: {'l
r': 0.0003521594585789136, 'weight_decay': 8.280073674432183e-05, 'batch_size': 16, 'optimizer':
'Adam', 'dropout': 0.4308834232239646}. Best is trial 10 with value: 0.8778054862842892.
[I 2025-07-07 21:16:16,133] Trial 15 finished with value: 0.8250801567509797 and parameters: {'l
r': 0.0012891212459071252, 'weight_decay': 2.8839213292293783e-05, 'batch_size': 16, 'optimizer':
'Adam', 'dropout': 0.513935811127507}. Best is trial 10 with value: 0.8778054862842892.
[I 2025-07-07 21:17:40,151] Trial 16 finished with value: 0.8382614891343071 and parameters: {'l
r': 0.00023558979250658414, 'weight_decay': 1.0158431335825086e-05, 'batch_size': 32, 'optimize
r': 'Adam', 'dropout': 0.4538467760701363}. Best is trial 10 with value: 0.8778054862842892.
[I 2025-07-07 21:19:11,846] Trial 17 finished with value: 0.8361239757748485 and parameters: {'l
r': 0.0010039589402367894, 'weight_decay': 0.00010118316062069908, 'batch_size': 16, 'optimizer':
```

```
'Adam', 'dropout': 0.5259353831415208}. Best is trial 10 with value: 0.8778054862842892.
[I 2025-07-07 21:20:43,640] Trial 18 finished with value: 0.809048806555041 and parameters: {'l
r': 0.0020105305613189726, 'weight_decay': 2.433741613160891e-05, 'batch_size': 16, 'optimizer':
'Adam', 'dropout': 0.41263709392347586}. Best is trial 10 with value: 0.8778054862842892.
[I 2025-07-07 21:22:07,107] Trial 19 finished with value: 0.8500178126113288 and parameters: {'l
r': 0.000680587738242438, 'weight_decay': 7.06209326606556e-05, 'batch_size': 32, 'optimizer': 'A
dam', 'dropout': 0.46685805143070047}. Best is trial 10 with value: 0.8778054862842892.
[I 2025-07-07 21:22:07,107] A new study created in memory with name: no-name-0986cbbc-81ef-49fd-9
db0-613aa7d33718
🏆 Best cnn trial:
  Accuracy: 0.8778054862842892
  Params: {'lr': 0.0004397830540797077, 'weight_decay': 4.729585299481905e-05, 'batch_size': 16,
'optimizer': 'Adam', 'dropout': 0.469694857002729}

🔍 Tuning RESNET model with Optuna...
```

```
[I 2025-07-07 21:24:22,236] Trial 0 finished with value: 0.8892055575347346 and parameters: {'l
r': 0.0012360167218039273, 'weight_decay': 2.752083144185464e-05, 'batch_size': 16, 'optimizer':
'SGD', 'momentum': 0.9227370099488055}. Best is trial 0 with value: 0.8892055575347346.
[I 2025-07-07 21:26:36,746] Trial 1 finished with value: 0.8863555397221232 and parameters: {'l
r': 0.0013575320751067243, 'weight_decay': 0.0001842578591050113, 'batch_size': 16, 'optimizer':
'SGD', 'momentum': 0.9212306595280184}. Best is trial 0 with value: 0.8892055575347346.
[I 2025-07-07 21:28:51,332] Trial 2 finished with value: 0.826148913430709 and parameters: {'lr':
0.003106703521844512, 'weight_decay': 0.00019393919661012923, 'batch_size': 16, 'optimizer': 'SG
D', 'momentum': 0.9729274493333744}. Best is trial 0 with value: 0.8892055575347346.
[I 2025-07-07 21:30:38,080] Trial 3 finished with value: 0.8838617741360884 and parameters: {'l
r': 0.0001910145119798502, 'weight_decay': 0.00013990832595977962, 'batch_size': 32, 'optimizer':
'Adam'}. Best is trial 0 with value: 0.8892055575347346.
[I 2025-07-07 21:32:25,696] Trial 4 finished with value: 0.8845742785892412 and parameters: {'l
r': 0.0017460771180567143, 'weight_decay': 4.2110455924369406e-05, 'batch_size': 32, 'optimizer':
'SGD', 'momentum': 0.8884244094336284}. Best is trial 0 with value: 0.8892055575347346.
[I 2025-07-07 21:34:41,019] Trial 5 finished with value: 0.8827930174563591 and parameters: {'l
r': 0.0016069988178358408, 'weight_decay': 0.0009185686969089284, 'batch_size': 16, 'optimizer':
'Adam'}. Best is trial 0 with value: 0.8892055575347346.
[I 2025-07-07 21:36:11,950] Trial 6 finished with value: 0.8785179907374421 and parameters: {'l
r': 0.00031617922356072384, 'weight_decay': 0.0004101083284530018, 'batch_size': 64, 'optimizer':
'SGD', 'momentum': 0.9797428078347781}. Best is trial 0 with value: 0.8892055575347346.
[I 2025-07-07 21:38:30,706] Trial 7 finished with value: 0.8489490559315995 and parameters: {'l
r': 0.004323799644582862, 'weight_decay': 0.00010490301455378904, 'batch_size': 16, 'optimizer':
'Adam'}. Best is trial 0 with value: 0.8892055575347346.
[I 2025-07-07 21:40:05,251] Trial 8 finished with value: 0.8941930887068044 and parameters: {'l
r': 0.004598938965786921, 'weight_decay': 1.1254502589625614e-05, 'batch_size': 64, 'optimizer':
'SGD', 'momentum': 0.8323628393531479}. Best is trial 8 with value: 0.8941930887068044.
[I 2025-07-07 21:41:55,433] Trial 9 finished with value: 0.8835055219095119 and parameters: {'l
r': 0.003282437965857967, 'weight_decay': 0.00016134655603376125, 'batch_size': 32, 'optimizer':
'SGD', 'momentum': 0.8252331029848555}. Best is trial 8 with value: 0.8941930887068044.
[I 2025-07-07 21:43:30,251] Trial 10 finished with value: 0.8692554328464553 and parameters: {'l
r': 0.008349926097717038, 'weight_decay': 1.0127762089954046e-05, 'batch_size': 64, 'optimizer':
'Adam'}. Best is trial 8 with value: 0.8941930887068044.
[I 2025-07-07 21:45:04,962] Trial 11 finished with value: 0.8585678660491628 and parameters: {'l
r': 0.000593039510904168, 'weight_decay': 1.1930078078220512e-05, 'batch_size': 64, 'optimizer':
'SGD', 'momentum': 0.8111144390068357}. Best is trial 8 with value: 0.8941930887068044.
[I 2025-07-07 21:46:39,984] Trial 12 finished with value: 0.8859992874955468 and parameters: {'l
r': 0.009910422834614616, 'weight_decay': 3.4578881712562506e-05, 'batch_size': 64, 'optimizer':
'SGD', 'momentum': 0.8772956660946556}. Best is trial 8 with value: 0.8941930887068044.
[I 2025-07-07 21:48:59,118] Trial 13 finished with value: 0.8774492340577129 and parameters: {'l
r': 0.0007207735249359813, 'weight_decay': 2.692773206233824e-05, 'batch_size': 16, 'optimizer':
'SGD', 'momentum': 0.9290527773455698}. Best is trial 8 with value: 0.8941930887068044.
[I 2025-07-07 21:50:34,380] Trial 14 finished with value: 0.8724617028856431 and parameters: {'l
r': 0.00043301652418547805, 'weight_decay': 1.944865754059119e-05, 'batch_size': 64, 'optimizer':
'SGD', 'momentum': 0.849218176659077З}. Best is trial 8 with value: 0.8941930887068044.
[I 2025-07-07 21:53:02,635] Trial 15 finished with value: 0.8696116850730317 and parameters: {'l
r': 0.00012491409839642074, 'weight_decay': 5.8811436652396386e-05, 'batch_size': 16, 'optimize
r': 'SGD', 'momentum': 0.923990894502146}. Best is trial 8 with value: 0.8941930887068044.
[I 2025-07-07 21:54:44,005] Trial 16 finished with value: 0.8963306020662629 and parameters: {'l
r': 0.005874727198538598, 'weight_decay': 1.8000310277495893e-05, 'batch_size': 64, 'optimizer':
'SGD', 'momentum': 0.8495487276964692}. Best is trial 16 with value: 0.8963306020662629.
[I 2025-07-07 21:56:18,542] Trial 17 finished with value: 0.8859992874955468 and parameters: {'l
r': 0.005519923037381905, 'weight_decay': 1.9206703017625043e-05, 'batch_size': 64, 'optimizer':
'SGD', 'momentum': 0.8563178009633323}. Best is trial 16 with value: 0.8963306020662629.
[I 2025-07-07 21:57:52,933] Trial 18 finished with value: 0.8421802636266477 and parameters: {'l
r': 0.0025575399569297995, 'weight_decay': 6.7242569512722e-05, 'batch_size': 64, 'optimizer': 'A
dam'}. Best is trial 16 with value: 0.8963306020662629.
[I 2025-07-07 21:59:27,556] Trial 19 finished with value: 0.8831492696829355 and parameters: {'l
r': 0.006194030003646635, 'weight_decay': 1.5314269259893173e-05, 'batch_size': 64, 'optimizer':
```

```
'SGD', 'momentum': 0.8398267541365352}. Best is trial 16 with value: 0.8963306020662629.
[I 2025-07-07 21:59:27,557] A new study created in memory with name: no-name-f3a4e176-3f70-4524-a
44f-0a7fa6a2d9eb
```

```
🏆 Best resnet trial:
  Accuracy: 0.8963306020662629
  Params: {'lr': 0.005874727198538598, 'weight_decay': 1.8000310277495893e-05, 'batch_size': 64,
'optimizer': 'SGD', 'momentum': 0.8495487276964692}

🔍 Tuning EFFICIENTNET model with Optuna...
```

```
[I 2025-07-07 22:01:04,073] Trial 0 finished with value: 0.8635553972212326 and parameters: {'l
r': 0.0008307701907002361, 'weight_decay': 4.160674731559605e-05, 'batch_size': 64, 'optimizer':
'Adam'}. Best is trial 0 with value: 0.8635553972212326.
[I 2025-07-07 22:02:40,649] Trial 1 finished with value: 0.8496615603847524 and parameters: {'l
r': 0.007743285836786923, 'weight_decay': 0.0005637225075632932, 'batch_size': 64, 'optimizer':
'Adam'}. Best is trial 0 with value: 0.8635553972212326.
[I 2025-07-07 22:05:18,771] Trial 2 finished with value: 0.8460990381189882 and parameters: {'l
r': 0.001327179751940l518, 'weight_decay': 3.1004841721040866e-05, 'batch_size': 16, 'optimizer':
'SGD', 'momentum': 0.8851438043443012}. Best is trial 0 with value: 0.8635553972212326.
[I 2025-07-07 22:07:13,423] Trial 3 finished with value: 0.8589241182757392 and parameters: {'l
r': 0.0005227024835840878, 'weight_decay': 1.33042915263065e-05, 'batch_size': 32, 'optimizer':
'SGD', 'momentum': 0.9737121355775504}. Best is trial 0 with value: 0.8635553972212326.
[I 2025-07-07 22:09:51,905] Trial 4 finished with value: 0.8403990024937655 and parameters: {'l
r': 0.00016022654010377105, 'weight_decay': 2.190579917960545e-05, 'batch_size': 16, 'optimizer':
'SGD', 'momentum': 0.9668653299355159}. Best is trial 0 with value: 0.8635553972212326.
[I 2025-07-07 22:11:27,690] Trial 5 finished with value: 0.8307801923762024 and parameters: {'l
r': 0.0003772041743245481, 'weight_decay': 0.00045374309047425346, 'batch_size': 64, 'optimizer':
'SGD', 'momentum': 0.9224246721360804}. Best is trial 0 with value: 0.8635553972212326.
[I 2025-07-07 22:14:09,663] Trial 6 finished with value: 0.8418240114000712 and parameters: {'l
r': 0.00017976404119347887, 'weight_decay': 2.6848926201939666e-05, 'batch_size': 16, 'optimize
r': 'Adam'}. Best is trial 0 with value: 0.8635553972212326.
[I 2025-07-07 22:16:07,697] Trial 7 finished with value: 0.8215176344852155 and parameters: {'l
r': 0.00010386045250729953, 'weight_decay': 0.0002802891579896262, 'batch_size': 32, 'optimizer':
'SGD', 'momentum': 0.928837107891128}. Best is trial 0 with value: 0.8635553972212326.
[I 2025-07-07 22:18:53,261] Trial 8 finished with value: 0.8688991806198789 and parameters: {'l
r': 0.0027108336171661885, 'weight_decay': 3.248858080604843e-05, 'batch_size': 16, 'optimizer':
'SGD', 'momentum': 0.912922678112737}. Best is trial 8 with value: 0.8688991806198789.
[I 2025-07-07 22:21:14,261] Trial 9 finished with value: 0.8610616316351977 and parameters: {'l
r': 0.0034174807256856722, 'weight_decay': 0.00012255641136592582, 'batch_size': 32, 'optimizer':
'SGD', 'momentum': 0.9175739223363242}. Best is trial 8 with value: 0.8688991806198789.
[I 2025-07-07 22:24:05,151] Trial 10 finished with value: 0.8585678660491628 and parameters: {'l
r': 0.0022945033642523473, 'weight_decay': 8.722657253042298e-05, 'batch_size': 16, 'optimizer':
'Adam'}. Best is trial 8 with value: 0.8688991806198789.
[I 2025-07-07 22:25:41,561] Trial 11 finished with value: 0.8674741717135732 and parameters: {'l
r': 0.0009235711844907976, 'weight_decay': 6.348628249421892e-05, 'batch_size': 64, 'optimizer':
'Adam'}. Best is trial 8 with value: 0.8688991806198789.
[I 2025-07-07 22:27:18,630] Trial 12 finished with value: 0.8639116494478091 and parameters: {'l
r': 0.004581120063792124, 'weight_decay': 7.295240242610951e-05, 'batch_size': 64, 'optimizer':
'Adam'}. Best is trial 8 with value: 0.8688991806198789.
[I 2025-07-07 22:28:55,436] Trial 13 finished with value: 0.8617741360883505 and parameters: {'l
r': 0.001632967172381866, 'weight_decay': 1.1715076622423518e-05, 'batch_size': 64, 'optimizer':
'Adam'}. Best is trial 8 with value: 0.8688991806198789.
[I 2025-07-07 22:31:34,496] Trial 14 finished with value: 0.8393302458140364 and parameters: {'l
r': 0.0007981003701818353, 'weight_decay': 0.00022203798362270652, 'batch_size': 16, 'optimizer':
'SGD', 'momentum': 0.8072243189253736}. Best is trial 8 with value: 0.8688991806198789.
[I 2025-07-07 22:33:10,505] Trial 15 finished with value: 0.85785536159601 and parameters: {'lr':
0.006445019340011479, 'weight_decay': 4.472985482775856e-05, 'batch_size': 64, 'optimizer': 'Ada
m'}. Best is trial 8 with value: 0.8688991806198789.
[I 2025-07-07 22:35:51,494] Trial 16 finished with value: 0.8589241182757392 and parameters: {'l
r': 0.002601532396857496, 'weight_decay': 0.0001405960202021294, 'batch_size': 16, 'optimizer':
'SGD', 'momentum': 0.8562842266742332}. Best is trial 8 with value: 0.8688991806198789.
[I 2025-07-07 22:37:34,930] Trial 17 finished with value: 0.8656929105806911 and parameters: {'l
r': 0.0003679110712798055, 'weight_decay': 5.959086569988057e-05, 'batch_size': 64, 'optimizer':
'Adam'}. Best is trial 8 with value: 0.8688991806198789.
[I 2025-07-07 22:40:14,778] Trial 18 finished with value: 0.8496615603847524 and parameters: {'l
r': 0.0012085141350595548, 'weight_decay': 1.720566758826368e-05, 'batch_size': 16, 'optimizer':
'SGD', 'momentum': 0.8544013254641721}. Best is trial 8 with value: 0.8688991806198789.
[I 2025-07-07 22:42:09,736] Trial 19 finished with value: 0.8664054150338439 and parameters: {'l
r': 0.0020219082911031878, 'weight_decay': 0.000822977801751363, 'batch_size': 32, 'optimizer':
'Adam'}. Best is trial 8 with value: 0.8688991806198789.
🏆 Best efficientnet trial:
  Accuracy: 0.8688991806198789
  Params: {'lr': 0.0027108336171661885, 'weight_decay': 3.248858080604843e-05, 'batch_size': 16,
'optimizer': 'SGD', 'momentum': 0.912922678112737}
```

Final Optuna Tuning Results (After Expanding Search Space)

To improve model performance, additional hyperparameters were tuned for each model:

- **All models**: `learning_rate`, `weight_decay`, `batch_size`, `optimizer`
- **CNN only**: `dropout` (specific to the architecture)
- **ResNet50 & EfficientNet-B0 with SGD only**: `momentum`

Each model was tuned over 20 trials. Below are the best results obtained from the Optuna studies:

Best CNN Trial

- **Accuracy**: `0.8778`
- **Best Parameters**:
  - `lr`: `0.00043978`
  - `weight_decay`: `4.73e-05`
  - `batch_size`: `16`
  - `optimizer`: `Adam`
  - `dropout`: `0.4697`

Best ResNet50 Trial

- **Accuracy**: `0.8963`
- **Best Parameters**:
  - `lr`: `0.00587473`
  - `weight_decay`: `1.80e-05`
  - `batch_size`: `64`
  - `optimizer`: `SGD`
  - `momentum`: `0.8495`

Best EfficientNet-B0 Trial

- **Accuracy**: `0.8689`
- **Best Parameters**:
  - `lr`: `0.00271083`
  - `weight_decay`: `3.25e-05`
  - `batch_size`: `16`
  - `optimizer`: `SGD`
  - `momentum`: `0.9129`

These results indicate that **ResNet50** achieved the best validation accuracy, followed by **EfficientNet-B0** and then **CNN**. The expanded hyperparameter search helped uncover better configurations, particularly for SGD-based models where momentum played a significant role.

4.2.3 Retrain the Model with Best Hyperparameters

This code block retrains each model from scratch using the full training dataset (`train_dataset`, not just `train_loader` from the split) and the best hyperparameters found via Optuna. It includes both training and validation phases and logs loss values for each epoch.

1. CNN

Data Loaders

- `train_loader` and `val_loader` load the training and validation datasets respectively using the best batch size (`16`).

Model Definition

- `SimpleCNN` is initialized with a dropout rate of `0.4697` and moved to the appropriate device (GPU).

Optimizer and Loss

- Uses **Adam** optimizer with the best learning rate (`0.00043978`) and weight decay (`4.73e-05`).
- Applies **CrossEntropyLoss**, which is commonly used for multi-class classification problems.

Training Loop (10 Epochs)

Each epoch involves:

- **Training Phase:**
  - Sets the model to training mode.
  - Iterates through mini-batches:
    - Performs a forward pass.
    - Computes loss.
    - Backpropagates the error.
    - Updates weights using the optimizer.
  - Accumulates and stores average training loss.
- **Validation Phase:**
  - Sets the model to evaluation mode.
  - Disables gradient computation.
  - Runs forward passes on validation data.
  - Calculates and stores average validation loss.

Logging and Monitoring

- After every epoch, both training and validation losses are printed.
- The losses are stored in `cnn_train_losses` and `cnn_val_losses` for further visualization.
- These curves help diagnose **overfitting** (if val loss increases while train loss decreases) or **underfitting** (if both remain high).

This setup ensures proper monitoring of model performance and generalization capability during retraining.

In [30]:
```python
import torch.optim as optim
```

In [35]:
```python
# Best CNN Hyperparameters
cnn_params = {
    'lr': 0.00043978,
    'weight_decay': 4.73e-05,
    'batch_size': 16,
    'dropout': 0.4697,
    'optimizer': 'Adam'
}

# Define loaders
train_loader = DataLoader(train_dataset, batch_size=cnn_params['batch_size'], shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=cnn_params['batch_size'], shuffle=False)

# Define model
cnn_model = SimpleCNN(dropout_rate=cnn_params['dropout']).to(device)

# Optimizer
if cnn_params['optimizer'] == 'Adam':
    optimizer = optim.Adam(cnn_model.parameters(), lr=cnn_params['lr'], weight_decay=cnn_params[
else:
    optimizer = optim.SGD(cnn_model.parameters(), lr=cnn_params['lr'], weight_decay=cnn_params['

# Loss function
criterion = nn.CrossEntropyLoss()

# Initialize lists to store training and validation losses
cnn_train_losses = []
cnn_val_losses = []

# Training loop
for epoch in range(10):
    # ---- Training ----
    cnn_model.train()
    running_train_loss = 0
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = cnn_model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_train_loss += loss.item()
    epoch_train_loss = running_train_loss / len(train_loader)
    cnn_train_losses.append(epoch_train_loss)

    # ---- Validation ----
    cnn_model.eval()
    running_val_loss = 0
    with torch.no_grad():
        for inputs, labels in val_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = cnn_model(inputs)
            val_loss = criterion(outputs, labels)
            running_val_loss += val_loss.item()
    epoch_val_loss = running_val_loss / len(val_loader)
    cnn_val_losses.append(epoch_val_loss)

    # ---- Logging ----
    print(f"[CNN] Epoch {epoch+1} - Train Loss: {epoch_train_loss:.4f} | Val Loss: {epoch_val_lo
[CNN] Epoch 1 - Train Loss: 0.9366 | Val Loss: 0.6673
[CNN] Epoch 2 - Train Loss: 0.6413 | Val Loss: 0.5413
[CNN] Epoch 3 - Train Loss: 0.5439 | Val Loss: 0.4800
[CNN] Epoch 4 - Train Loss: 0.4896 | Val Loss: 0.4185
[CNN] Epoch 5 - Train Loss: 0.4583 | Val Loss: 0.3928
[CNN] Epoch 6 - Train Loss: 0.4221 | Val Loss: 0.3860
[CNN] Epoch 7 - Train Loss: 0.3971 | Val Loss: 0.3206
[CNN] Epoch 8 - Train Loss: 0.3771 | Val Loss: 0.3166
[CNN] Epoch 9 - Train Loss: 0.3522 | Val Loss: 0.2940
[CNN] Epoch 10 - Train Loss: 0.3264 | Val Loss: 0.3037
```

2. ResNet50

Data Loaders

- `train_loader` and `val_loader` are initialized using the best batch size (`64`).
- These loaders feed images to the model in mini-batches for training and validation.

Model Definition

- A pre-trained `ResNet50` model is loaded using `torchvision.models`.
- All layers are **frozen** (`requires_grad=False`) to retain learned ImageNet features.
- The final classification layer (`fc`) is **replaced** with a new fully connected layer to output 6 classes.

Optimizer and Loss Function

- **SGD** optimizer is used (as per best params) with:
  - Learning rate = `0.0058747`
  - Momentum = `0.8495`
  - Weight decay = `1.8e-05`
- **CrossEntropyLoss** is used, suitable for multi-class classification.

Training Loop (10 Epochs)

Each epoch has two phases:

- **Training Phase:**
  - Sets model to training mode.
  - Performs forward pass, loss calculation, backpropagation, and weight updates.
  - Accumulates average training loss.
- **Validation Phase:**
  - Sets model to evaluation mode (disables dropout, batch norm updates).
  - Disables gradient tracking for efficiency.
  - Computes average validation loss.

Logging and Monitoring

- At each epoch, prints both training and validation losses.
- Stores losses in `resnet_train_losses` and `resnet_val_losses` lists.
- These values can later be plotted to detect **Overfitting**, **Underfitting** or **Good fit**.

This process ensures that the model's generalization ability is tracked as it learns with the best hyperparameters.

```
In [36]: # Best ResNet Hyperparameters
resnet_params = {
    'lr': 0.0058747,
    'weight_decay': 1.8e-05,
    'batch_size': 64,
    'optimizer': 'SGD',
    'momentum': 0.8495
}

# Define loaders (use train/val split if available)
train_loader = DataLoader(train_dataset, batch_size=resnet_params['batch_size'], shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=resnet_params['batch_size'], shuffle=False)

# Load pre-trained ResNet
resnet_model = models.resnet50(pretrained=True)
for param in resnet_model.parameters():
    param.requires_grad = False

# Replace final layer
in_features = resnet_model.fc.in_features
resnet_model.fc = nn.Linear(in_features, 6)  # 6 classes
resnet_model = resnet_model.to(device)

# Optimizer
if resnet_params['optimizer'] == 'SGD':
    optimizer = optim.SGD(resnet_model.parameters(), lr=resnet_params['lr'],
                          momentum=resnet_params['momentum'], weight_decay=resnet_params['weight_
else:
    optimizer = optim.Adam(resnet_model.parameters(), lr=resnet_params['lr'],
                           weight_decay=resnet_params['weight_decay'])

# Loss function
criterion = nn.CrossEntropyLoss()

# Lists to store losses
resnet_train_losses = []
resnet_val_losses = []

# Training Loop
for epoch in range(10):
    # ---- Training ----
    resnet_model.train()
    running_train_loss = 0
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = resnet_model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_train_loss += loss.item()
    epoch_train_loss = running_train_loss / len(train_loader)
    resnet_train_losses.append(epoch_train_loss)

    # ---- Validation ----
    resnet_model.eval()
    running_val_loss = 0
    with torch.no_grad():
        for inputs, labels in val_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = resnet_model(inputs)
            val_loss = criterion(outputs, labels)
            running_val_loss += val_loss.item()
    epoch_val_loss = running_val_loss / len(val_loader)
    resnet_val_losses.append(epoch_val_loss)

    # ---- Logging ----
    print(f"[ResNet] Epoch {epoch+1} - Train Loss: {epoch_train_loss:.4f} | Val Loss: {epoch_val_
```

```
[ResNet] Epoch 1 - Train Loss: 0.5235 | Val Loss: 0.3869
[ResNet] Epoch 2 - Train Loss: 0.3635 | Val Loss: 0.3657
[ResNet] Epoch 3 - Train Loss: 0.3396 | Val Loss: 0.3267
[ResNet] Epoch 4 - Train Loss: 0.3349 | Val Loss: 0.3067
[ResNet] Epoch 5 - Train Loss: 0.3228 | Val Loss: 0.3022
[ResNet] Epoch 6 - Train Loss: 0.3237 | Val Loss: 0.3115
[ResNet] Epoch 7 - Train Loss: 0.3168 | Val Loss: 0.2782
[ResNet] Epoch 8 - Train Loss: 0.3187 | Val Loss: 0.2942
[ResNet] Epoch 9 - Train Loss: 0.3057 | Val Loss: 0.2813
[ResNet] Epoch 10 - Train Loss: 0.3095 | Val Loss: 0.3056
```

3. EfficientNet-B0

Data Loaders

- `train_loader` and `val_loader` are created using the best batch size (`16`).
- These feed the training and validation datasets into the model during training.

Model Definition

- A pre-trained `EfficientNet-B0` model is loaded using `torchvision.models` .
- The **feature extractor layers are frozen** to retain previously learned visual features from ImageNet.
- The classifier head is **replaced** with a fully connected layer that outputs predictions for 6 classes.

Optimizer and Loss Function

- **SGD** optimizer is used (based on best parameters) with:
  - Learning rate = `0.0027108`
  - Momentum = `0.9129`
  - Weight decay = `3.25e-05`
- **CrossEntropyLoss** is used for multi-class classification.

Training Loop (10 Epochs)

Each epoch consists of:

- **Training Phase:**
  - Sets model to training mode.
  - Performs forward pass, computes loss, backpropagates, and updates weights.
  - Accumulates average training loss.

- **Validation Phase:**
    - Disables gradient computation for efficiency.
    - Sets model to evaluation mode (no dropout, no batch norm updates).
    - Computes validation loss over the validation set.

Logging and Monitoring

- Logs and prints train and validation loss after each epoch.
- Losses are stored in `efficientnet_train_losses` and `efficientnet_val_losses` for visualization.
- These curves help identify Overfitting, Underfitting or Good fit.

This setup ensures efficient fine-tuning of EfficientNet while monitoring performance across epochs.

In [37]:
```
# Best EfficientNet Hyperparameters
efficientnet_params = {
    'lr': 0.0027108,
    'weight_decay': 3.25e-05,
    'batch_size': 16,
    'optimizer': 'SGD',
    'momentum': 0.9129
}

# DataLoaders (use val_dataset if available)
train_loader = DataLoader(train_dataset, batch_size=efficientnet_params['batch_size'], shuffle=Tr
val_loader = DataLoader(val_dataset, batch_size=efficientnet_params['batch_size'], shuffle=False

# Load pretrained EfficientNet-B0
efficientnet_model = models.efficientnet_b0(pretrained=True)
for param in efficientnet_model.features.parameters():
    param.requires_grad = False

# Replace classifier
in_features = efficientnet_model.classifier[1].in_features
efficientnet_model.classifier[1] = nn.Linear(in_features, 6)  # 6 classes
efficientnet_model = efficientnet_model.to(device)

# Optimizer
if efficientnet_params['optimizer'] == 'SGD':
    optimizer = optim.SGD(efficientnet_model.parameters(),
                          lr=efficientnet_params['lr'],
                          momentum=efficientnet_params['momentum'],
                          weight_decay=efficientnet_params['weight_decay'])
else:
    optimizer = optim.Adam(efficientnet_model.parameters(),
                           lr=efficientnet_params['lr'],
                           weight_decay=efficientnet_params['weight_decay'])

# Loss function
criterion = nn.CrossEntropyLoss()

# Store loss per epoch
efficientnet_train_losses = []
efficientnet_val_losses = []

# Training Loop
```

```python
for epoch in range(10):
    # --- Training ---
    efficientnet_model.train()
    running_train_loss = 0
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = efficientnet_model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_train_loss += loss.item()
    epoch_train_loss = running_train_loss / len(train_loader)
    efficientnet_train_losses.append(epoch_train_loss)

    # --- Validation ---
    efficientnet_model.eval()
    running_val_loss = 0
    with torch.no_grad():
        for inputs, labels in val_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = efficientnet_model(inputs)
            val_loss = criterion(outputs, labels)
            running_val_loss += val_loss.item()
    epoch_val_loss = running_val_loss / len(val_loader)
    efficientnet_val_losses.append(epoch_val_loss)

    # --- Logging ---
    print(f"[EfficientNet] Epoch {epoch+1} - Train Loss: {epoch_train_loss:.4f} | Val Loss: {epo
```

```
[EfficientNet] Epoch 1 - Train Loss: 0.6767 | Val Loss: 0.4627
[EfficientNet] Epoch 2 - Train Loss: 0.5456 | Val Loss: 0.4203
[EfficientNet] Epoch 3 - Train Loss: 0.5124 | Val Loss: 0.3997
[EfficientNet] Epoch 4 - Train Loss: 0.5057 | Val Loss: 0.4078
[EfficientNet] Epoch 5 - Train Loss: 0.4975 | Val Loss: 0.3925
[EfficientNet] Epoch 6 - Train Loss: 0.4989 | Val Loss: 0.3892
[EfficientNet] Epoch 7 - Train Loss: 0.4944 | Val Loss: 0.3932
[EfficientNet] Epoch 8 - Train Loss: 0.5005 | Val Loss: 0.3930
[EfficientNet] Epoch 9 - Train Loss: 0.4860 | Val Loss: 0.3882
[EfficientNet] Epoch 10 - Train Loss: 0.4887 | Val Loss: 0.3808
```

5.0 Model Evaluation and Performance Analysis

5.1 Evaluation Metrics Used

The image classification models were evaluated using the following metrics:

- Accuracy: Proportion of correctly classified images.
- Precision: Ability of the model to return only relevant images per class.
- Recall: Ability of the model to find all relevant images per class.
- F1-Score: Harmonic mean of precision and recall.
- Confusion Matrix: Visual breakdown of predicted vs. true labels.

These metrics are appropriate for multi-class classification, providing both overall and class-wise performance insights.

5.1.1 Evaluation Function

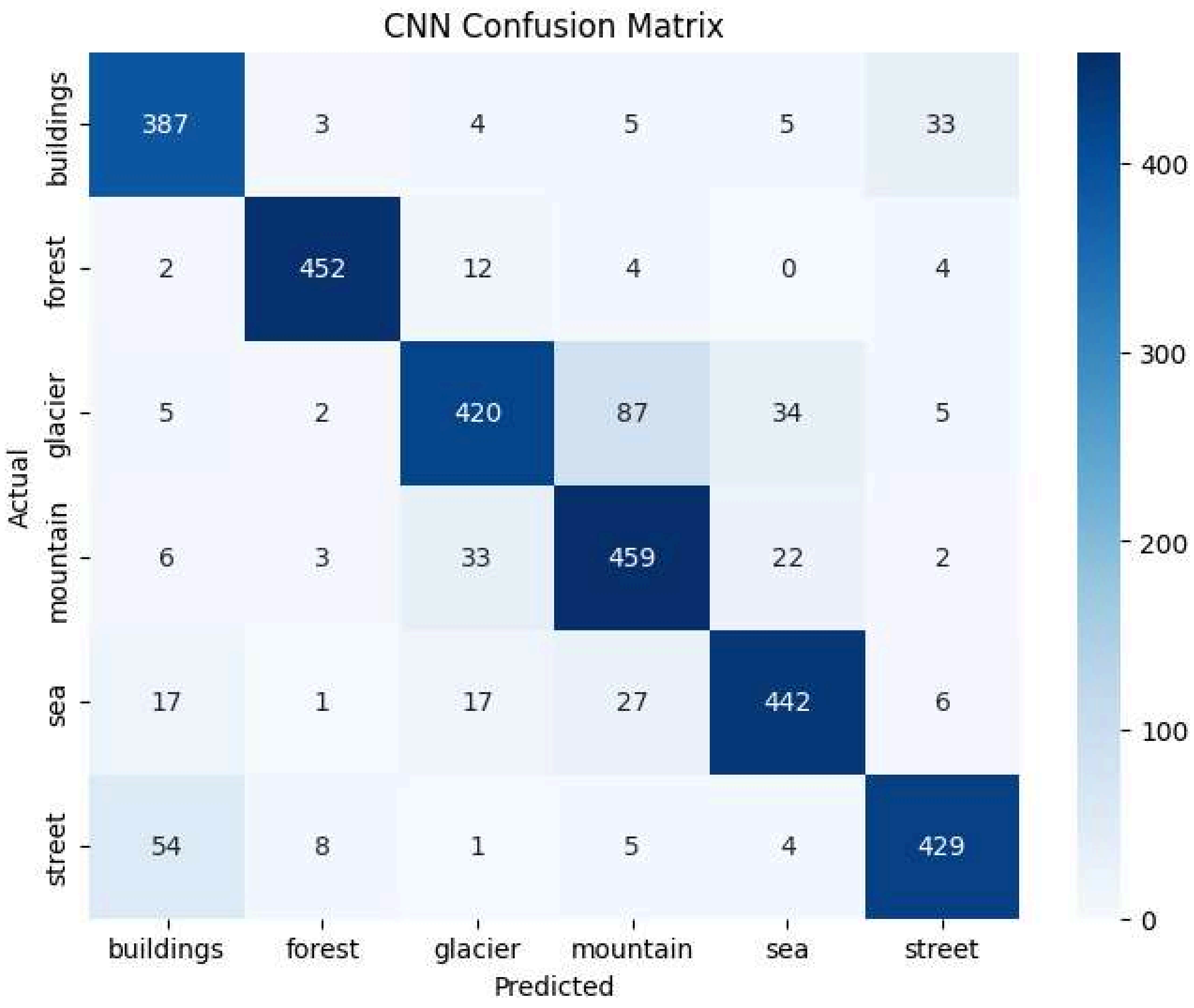
In [46]:
```
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, f1_score
import seaborn as sns
import matplotlib.pyplot as plt
```

In [56]:
```
def evaluate_model(model, test_loader, class_names, model_name):
    model.eval()
    all_preds = []
    all_labels = []

    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            preds = torch.argmax(outputs, dim=1)
            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    acc = accuracy_score(all_labels, all_preds)
    f1 = f1_score(all_labels, all_preds, average='macro')

    print(f"✅ {model_name} Accuracy: {acc:.4f}")
    print(f"\n📋 {model_name} Classification Report:")
    print(classification_report(all_labels, all_preds, target_names=class_names))

    cm = confusion_matrix(all_labels, all_preds)
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=class_names, yticklabels=class_names)
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.title(f"{model_name} Confusion Matrix")
    plt.show()

    return acc, f1
```

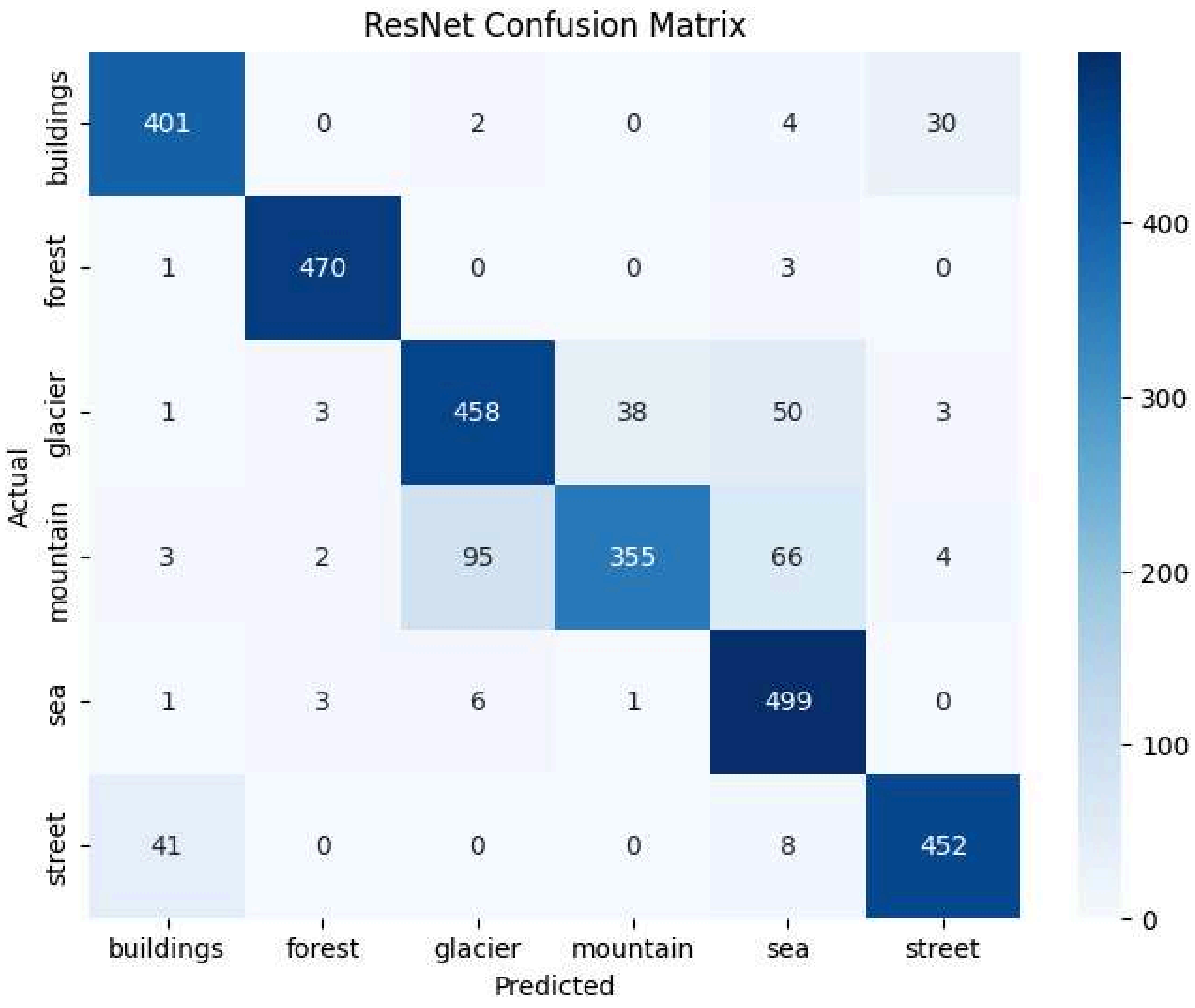
5.1.2 Apply Evaluation on Best Models

In [59]:
```
cnn_acc, cnn_f1 = evaluate_model(cnn_model, test_loader, class_names, model_name="CNN")
resnet_acc, resnet_f1 = evaluate_model(resnet_model, test_loader, class_names, model_name="ResNe
efficientnet_acc, efficientnet_f1 = evaluate_model(efficientnet_model, test_loader, class_names,
```

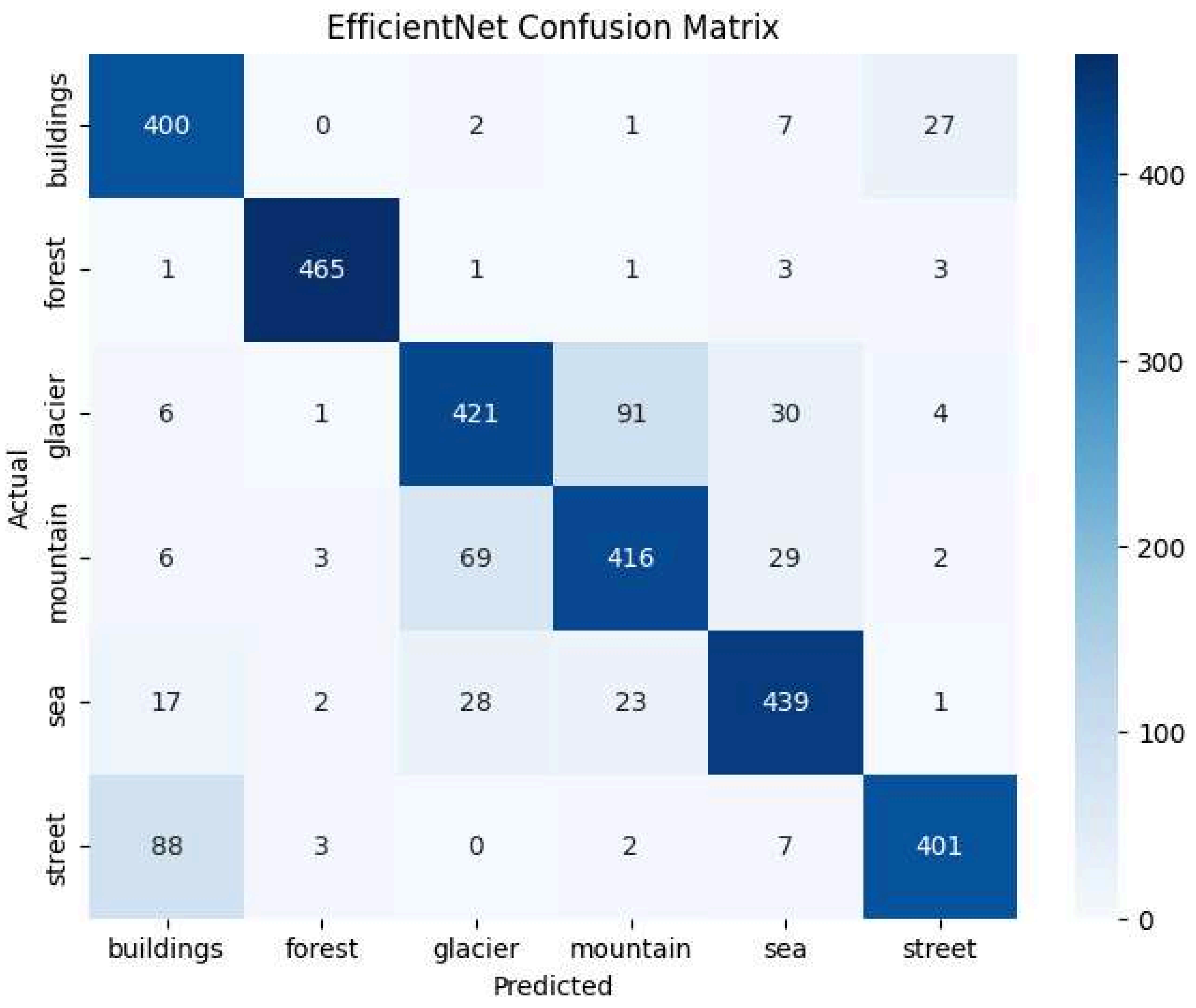
```
✅ CNN Accuracy: 0.8630

📋 CNN Classification Report:
              precision    recall  f1-score   support

   buildings       0.82      0.89      0.85       437
      forest       0.96      0.95      0.96       474
     glacier       0.86      0.76      0.81       553
    mountain       0.78      0.87      0.83       525
         sea       0.87      0.87      0.87       510
      street       0.90      0.86      0.88       501

    accuracy                           0.86      3000
   macro avg       0.87      0.87      0.86      3000
weighted avg       0.87      0.86      0.86      3000
```



```
✅ ResNet Accuracy: 0.8783

📋 ResNet Classification Report:
              precision    recall  f1-score   support

   buildings       0.90      0.92      0.91       437
      forest       0.98      0.99      0.99       474
     glacier       0.82      0.83      0.82       553
    mountain       0.90      0.68      0.77       525
         sea       0.79      0.98      0.88       510
      street       0.92      0.90      0.91       501

    accuracy                           0.88      3000
   macro avg       0.89      0.88      0.88      3000
weighted avg       0.88      0.88      0.88      3000
```



```
✅ EfficientNet Accuracy: 0.8473

📋 EfficientNet Classification Report:
              precision    recall  f1-score   support

   buildings       0.77      0.92      0.84       437
      forest       0.98      0.98      0.98       474
     glacier       0.81      0.76      0.78       553
    mountain       0.78      0.79      0.79       525
         sea       0.85      0.86      0.86       510
      street       0.92      0.80      0.85       501

    accuracy                           0.85      3000
   macro avg       0.85      0.85      0.85      3000
weighted avg       0.85      0.85      0.85      3000
```



Accuracy & Macro-Averaged F1-Score

Model	Accuracy	Macro F1-Score
CNN	86.30%	0.86
ResNet50	87.83%	0.88
EfficientNet	84.73%	0.85

- **ResNet** achieved the highest accuracy and macro F1-score, indicating better generalization and class-wise balance.
- **CNN** performs decently but lags slightly behind ResNet in most metrics.
- **EfficientNet** shows good precision but has slightly lower recall and accuracy.

Confusion Matrix Insights

- **CNN**:
    - Performs very well on `forest` (96% recall) and `street` (88% F1-score).
    - Struggles slightly with `glacier` (F1 = 0.81) with some misclassifications into `sea` and `mountain`. meanwhile, `mountain` (F1 = 0.83), with some misclassifications into `glacier` and `sea`.
- **ResNet**:
    - Near-perfect recall on `forest` (99%), `street` and `buildings` (91%).
    - Major improvement on `buildings` compared to CNN.
    - `Sea` class has slightly lower precision (0.79) due to confusion with `glacier` and `mountain`.
- **EfficientNet**:
    - Performs best on `forest` and `sea` (high recall and precision).
    - Noticeable drop in performance for `street` (only 80% recall), often misclassified as `buildings`.
    - Also more confusion between `glacier` and `mountain`, as seen by higher off-diagonal values.

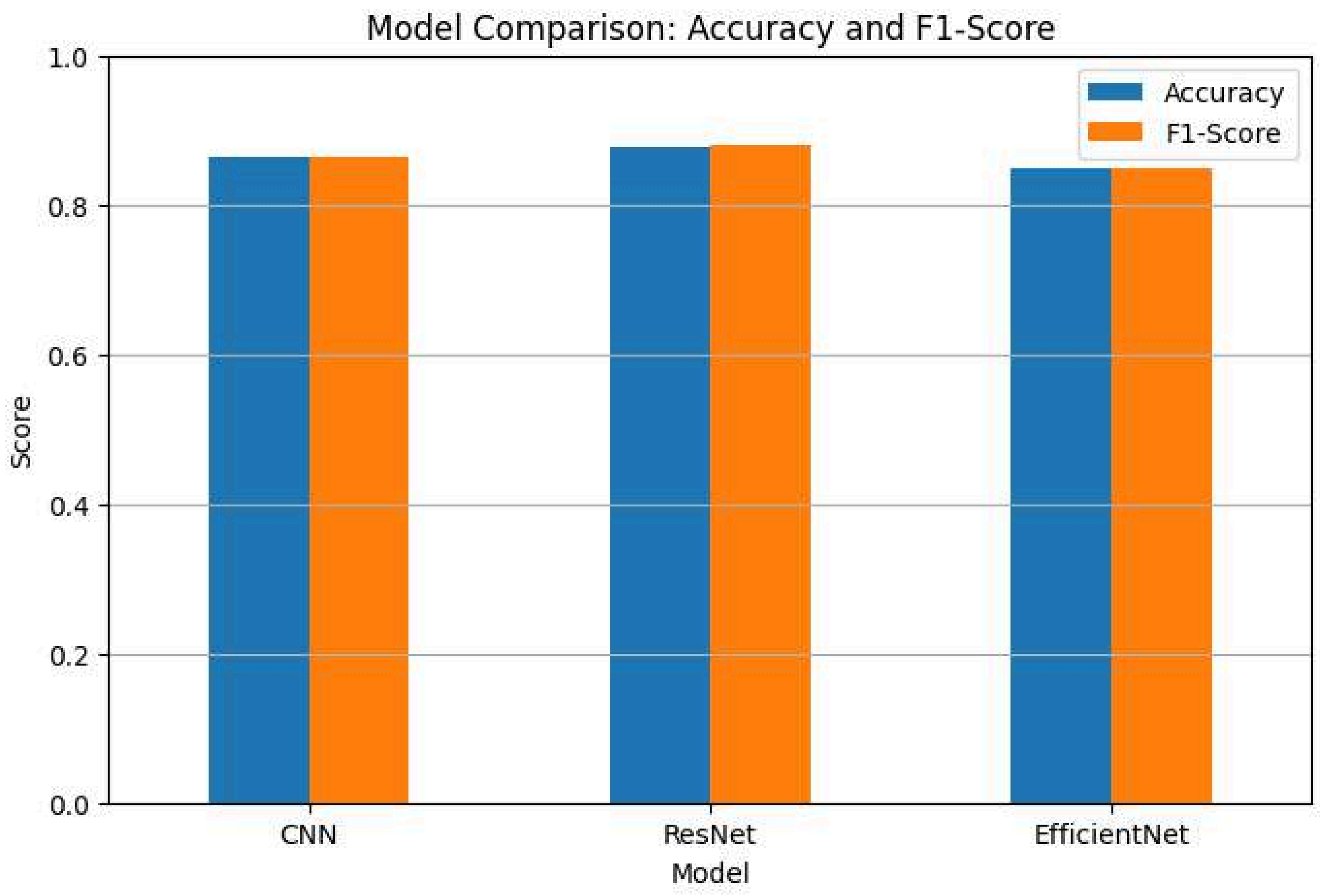
Final Remarks

- All models struggled at differentiating between `glacier` vs `mountain` vs `sea`, as well as `buildings` vs `street`, likely due to visual similarities in the images from these classes.
- **Best overall model**: `ResNet` due to consistent performance across all classes.
- **Most lightweight**: `CNN`, suitable for faster inference if slight accuracy trade-off is acceptable.
- **EfficientNet** may benefit from unfreezing more layers or longer training, as it currently underperforms despite its powerful architecture.

5.2 Visualizing Trends

5.2.1 Model Comparison: Accuracy & F1-Score

In [84]:
```python
import pandas as pd

results = pd.DataFrame({
    'Model': ['CNN', 'ResNet', 'EfficientNet'],
    'Accuracy': [cnn_acc, resnet_acc, efficientnet_acc],
    'F1-Score': [cnn_f1, resnet_f1, efficientnet_f1]
})

results.plot(kind='bar', x='Model', figsize=(8,5), legend=True)
plt.title("Model Comparison: Accuracy and F1-Score")
plt.ylabel("Score")
plt.ylim(0, 1.0)
plt.grid(axis='y')
plt.xticks(rotation=0)
plt.show()
```



The bar chart above compares the performance of three models; CNN, ResNet, and EfficientNet based on two metrics: Accuracy and F1-Score.

ResNet achieved the highest performance overall, with both accuracy and F1-score around 0.88, indicating strong and balanced classification across all classes.

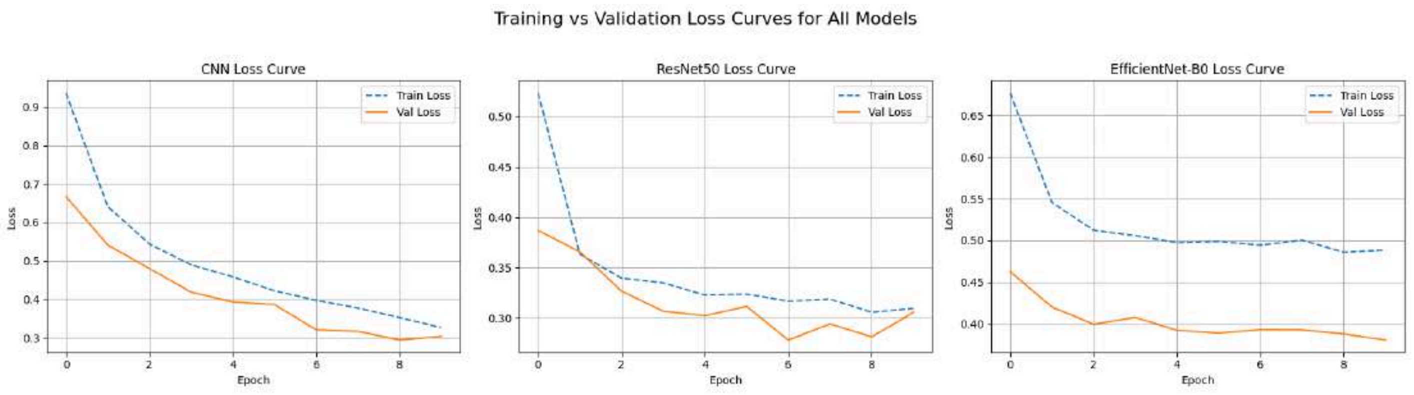
CNN performed slightly below ResNet, with an accuracy and F1-score of approximately 0.86. While it is a simpler model, it still managed competitive results.

EfficientNet scored the lowest among the three, with both accuracy and F1-score around 0.85. This suggests that while EfficientNet is generally powerful, it may have underperformed due to freezing too many layers or insufficient fine-tuning in this context.

> ResNet not only delivered the highest accuracy but also maintained strong F1-score consistency, making it the most reliable model for this image classification task. However, the margin of difference among the models is relatively small, showing that all three models are capable but may benefit from further tuning or deeper fine-tuning strategies.

5.2.2 Plotting Loss Curves

In [52]:
```
# Create subplots
fig, axes = plt.subplots(1, 3, figsize=(18, 5))

# CNN
axes[0].plot(cnn_train_losses, label='Train Loss', linestyle='--')
axes[0].plot(cnn_val_losses, label='Val Loss', linestyle='-')
axes[0].set_title('CNN Loss Curve')
axes[0].set_xlabel('Epoch')
axes[0].set_ylabel('Loss')
axes[0].legend()
axes[0].grid(True)

# ResNet
axes[1].plot(resnet_train_losses, label='Train Loss', linestyle='--')
axes[1].plot(resnet_val_losses, label='Val Loss', linestyle='-')
axes[1].set_title('ResNet50 Loss Curve')
axes[1].set_xlabel('Epoch')
axes[1].set_ylabel('Loss')
axes[1].legend()
axes[1].grid(True)

# EfficientNet
axes[2].plot(efficientnet_train_losses, label='Train Loss', linestyle='--')
axes[2].plot(efficientnet_val_losses, label='Val Loss', linestyle='-')
axes[2].set_title('EfficientNet-B0 Loss Curve')
axes[2].set_xlabel('Epoch')
axes[2].set_ylabel('Loss')
axes[2].legend()
axes[2].grid(True)

plt.suptitle('Training vs Validation Loss Curves for All Models', fontsize=16)
plt.tight_layout(rect=[0, 0, 1, 0.95])
plt.show()
```



The figure above presents the **training vs validation loss** curves across 10 epochs for the three models: **CNN**, **ResNet50**, and **EfficientNet-B0**.

Typically, validation loss is often higher than training loss curves (source) because:

1. Training data is seen repeatedly, so the model becomes good at minimizing loss on it.
2. Validation data is unseen, so performance tends to be worse (higher loss).
3. Data augmentation (`transform_train`) introduces randomness in training that helps generalize, but the clean validation set still differs.
4. Overfitting is more likely with image data if the model is large and the dataset is small.

However, all 3 models shown have validation loss curve lower than the training loss, because:

1. for the CNN model, dropout was applied (`cnn_model = SimpleCNN(dropout_rate=cnn_params['dropout']).to(device)`). Dropout randomly "drops" (disables) a fraction of neurons in the network during training only. But, during validation (and testing), dropout is turned off, so, all neurons are active. This results in lower loss on validation data than on dropout-weakened training data.
2. In both ResNet and EfficientNet models, the majority of layers are frozen to retain pretrained ImageNet features. As a result, only the final classification layer is being trained. The pretrained layers already generalize well to visual features, benefiting the validation set more than the training set.
3. The training set uses data augmentation to improve generalization (`transforms.RandomHorizontalFlip(), transforms.RandomRotation(15)`). These augmentations introduce variability, making the training task harder. The validation set is cleaner (no augmentations), leading to lower loss.

CNN Loss Curve:

- The training loss steadily decreases across epochs, indicating effective learning.
- Validation loss closely follows the training trend and improves consistently.
- No sign of overfitting is observed, as both curves appears smoooth and almost converge in the end, showing generalization is well maintained.

ResNet50 Loss Curve:

- ResNet50 starts with a low loss and continues to converge quickly within the first few epochs.
- Validation loss aligns well with training loss, with minor fluctuations suggesting good generalization.
- Slight gap between the curves indicates minimal overfitting, and the model maintains stable performance throughout.

EfficientNet-B0 Loss Curve:

- Shows the slowest improvement in training loss across epochs compared to CNN and ResNet.
- Validation loss improves rapidly and flattens, remaining consistently below training loss.
- The persistent gap between training and validation losses may indicate **underfitting**, possibly due to freezing too many layers or conservative training.

- Among the three models, **ResNet50** exhibits the most stable and flattened loss curves, suggesting consistent learning. While **none of the models demonstrate a clear convergence**, the training and validation losses maintain a small and stable gap, indicating steady progress.
- **CNN** also performs well and steadily improves, suitable for lightweight deployment.
- For **CNN** and **ResNet50**, extending training over more epochs could help determine whether the curves will converge or begin to diverge, offering better insight into their long-term learning behavior.
- In contrast, **EfficientNet-B0** shows a noticeably large and persistent gap between training and validation loss, which indicates underfitting. The model is potentially too constrained or insufficiently trained for the task and may require **unfreezing more layers** or **more epochs** to fully leverage its capacity.

5.3 Prediction Samples

This code block performs inference using three trained models (CNN, ResNet, EfficientNet) on images stored in the `seg_pred` directory.

- The model names are defined, corresponding model objects, class labels, and the image transform (`transform_test`) used for evaluation.
- The prediction directory path is specified.

The loop then:

1. Iterates over each model.
2. Loads and transforms each image (ignoring subdirectories).
3. Runs the model in evaluation mode without computing gradients.
4. Predicts the image class and maps the output index to the corresponding class name.
5. Stores all predictions and saves them into a CSV file for each model (`seg_pred_predictions_xx.csv`).

This modular code allows batch predictions and comparisons across different models on the same set of unseen images.

In [78]:
```
# List of models and their names
model_names = ['CNN', 'ResNet', 'EfficientNet']
models_list = [cnn_model, resnet_model, efficientnet_model]
```

In [65]:
```
# Class names
class_names
```

Out[65]:
```
['buildings', 'forest', 'glacier', 'mountain', 'sea', 'street']
```

In [81]:
```
# Transform
transform_test
```

Out[81]:
```
Compose(
    Resize(size=(150, 150), interpolation=bilinear, max_size=None, antialias=warn)
    ToTensor()
    Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
)
```

In [82]:
```
# Predict images in seg_pred
pred_dir = r"C:\Users\user\OneDrive - Universiti Teknologi Malaysia (UTM)\MRTB2153 Advanced Arti

# Evaluate each model
all_results = {}

for model_name, model in zip(model_names, models_list):
    print(f"🔍 Predicting with {model_name}...")
    model.eval()
    predicted_labels = []
    results = []

    for img_name in os.listdir(pred_dir):
        img_path = os.path.join(pred_dir, img_name)
        if os.path.isdir(img_path): continue
        try:
            img = Image.open(img_path).convert("RGB")
        except:
            continue

        input_tensor = transform_test(img).unsqueeze(0).to(device)

        with torch.no_grad():
            output = model(input_tensor)
            pred_class = torch.argmax(output, dim=1).item()
            predicted_label = class_names[pred_class]
            predicted_labels.append(predicted_label)
            results.append({'Filename': img_name, 'Predicted_Class': predicted_label})

    all_results[model_name] = results  # Save for second loop

    # Save CSV immediately
    df = pd.DataFrame(results)
    csv_path = f"seg_pred_predictions_{model_name.lower()}.csv"
    df.to_csv(csv_path, index=False)
    print(f"✅ Saved predictions to {csv_path}")
```

```
🔍 Predicting with CNN...
✅ Saved predictions to seg_pred_predictions_cnn.csv
🔍 Predicting with ResNet...
✅ Saved predictions to seg_pred_predictions_resnet.csv
🔍 Predicting with EfficientNet...
✅ Saved predictions to seg_pred_predictions_efficientnet.csv
```

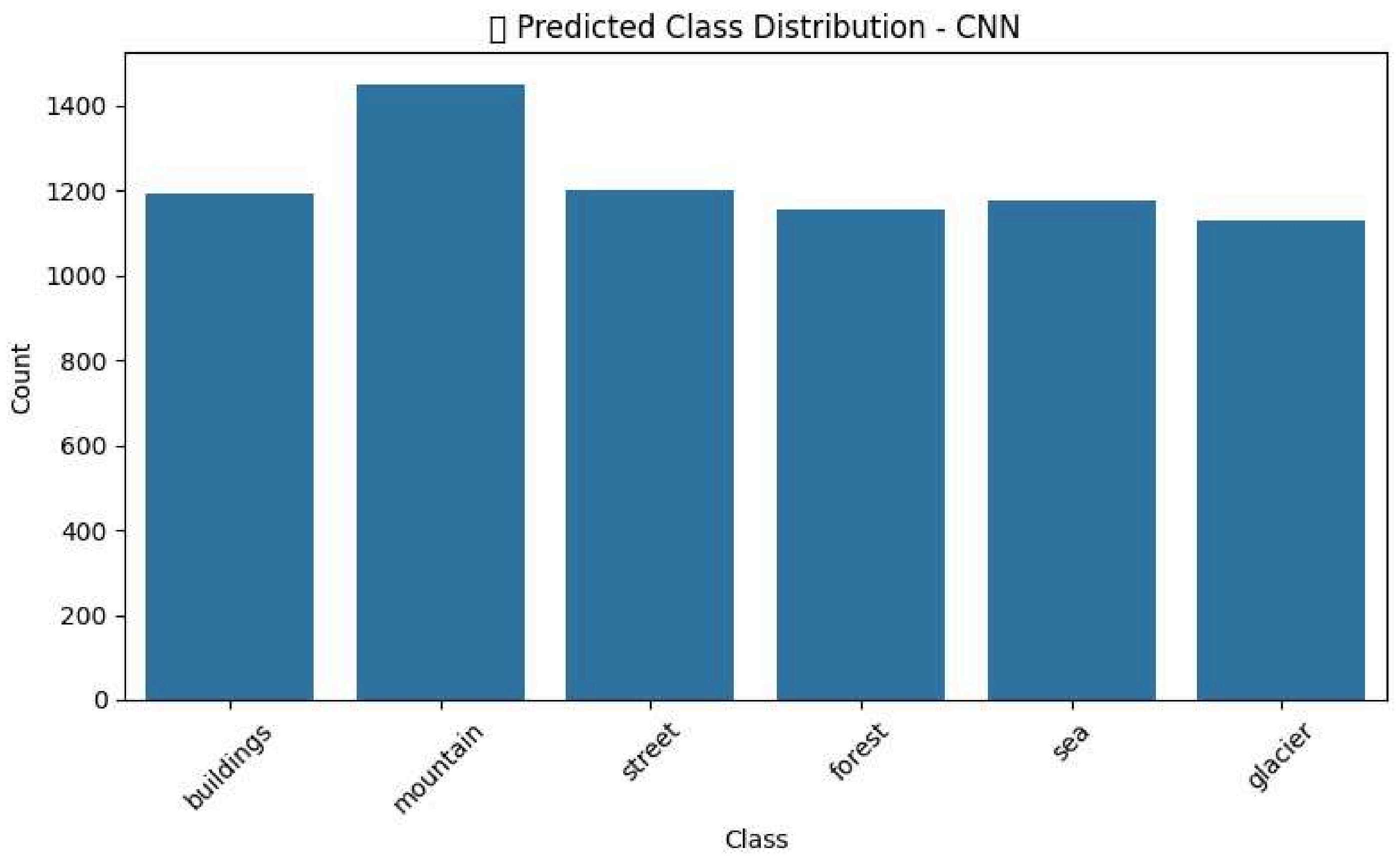
5.3.1 Plot & Visualize Predictions by Class

See how many images were predicted as each class

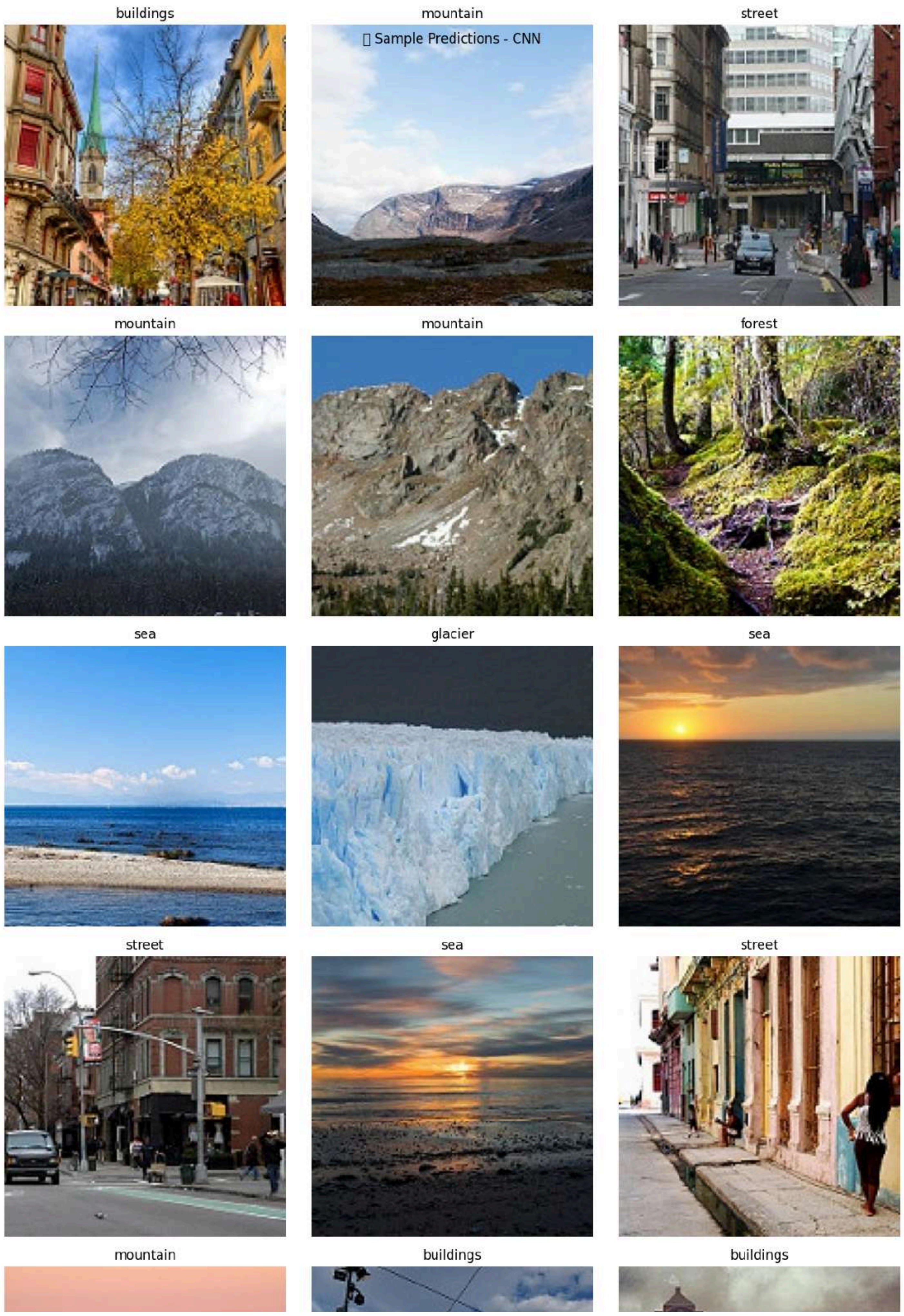
In [90]:

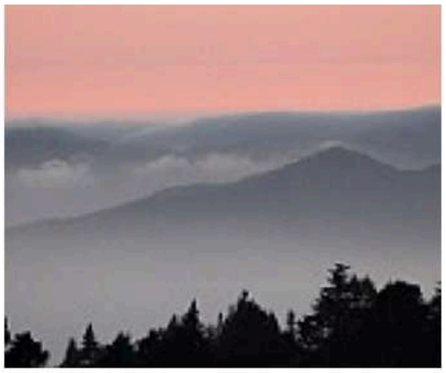
```
from collections import Counter
import math

for model_name in model_names:
    results = all_results[model_name]
    predicted_labels = [r['Predicted_Class'] for r in results]

    # 1. Plot class distribution
    counter = Counter(predicted_labels)
    plt.figure(figsize=(8,5))
    sns.barplot(x=list(counter.keys()), y=list(counter.values()))
    plt.title(f"📊 Predicted Class Distribution - {model_name}")
    plt.ylabel("Count")
    plt.xlabel("Class")
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.show()

    # 2. Show sample predictions
    sample_results = results[:30]
    n_samples = len(sample_results)
    n_cols = 3
    n_rows = math.ceil(n_samples / n_cols)

    fig, axs = plt.subplots(n_rows, n_cols, figsize=(12, 4 * n_rows))
    # If axs is 1D (for n_rows == 1), reshape for consistency
    axs = np.array(axs).reshape(n_rows, n_cols)

    for i, row in enumerate(sample_results):
        img_path = os.path.join(pred_dir, row['Filename'])
        img = Image.open(img_path)
        axs[i//n_cols, i%n_cols].imshow(img)
        axs[i//n_cols, i%n_cols].set_title(row['Predicted_Class'])
        axs[i//n_cols, i%n_cols].axis('off')

    # Hide unused axes
    for j in range(i + 1, n_rows * n_cols):
        axs[j // n_cols, j % n_cols].axis('off')

    plt.suptitle(f"🔍 Sample Predictions - {model_name}")
    plt.tight_layout()
    plt.show()
```

```
C:\Users\user\AppData\Local\Temp\ipykernel_38124\978616877.py:16: UserWarning: Glyph 128202 (\N{B
AR CHART}) missing from font(s) DejaVu Sans.
  plt.tight_layout()
C:\Users\user\anaconda3\envs\rl-assignment\lib\site-packages\IPython\core\pylabtools.py:170: User
Warning: Glyph 128202 (\N{BAR CHART}) missing from font(s) DejaVu Sans.
  fig.canvas.print_figure(bytes_io, **kw)
```



```
C:\Users\user\AppData\Local\Temp\ipykernel_38124\978616877.py:42: UserWarning: Glyph 128269 (\N{L
EFT-POINTING MAGNIFYING GLASS}) missing from font(s) DejaVu Sans.
  plt.tight_layout()
C:\Users\user\anaconda3\envs\rl-assignment\lib\site-packages\IPython\core\pylabtools.py:170: User
Warning: Glyph 128269 (\N{LEFT-POINTING MAGNIFYING GLASS}) missing from font(s) DejaVu Sans.
  fig.canvas.print_figure(bytes_io, **kw)
```


□ Sample Predictions - CNN
buildings
mountain
street
mountain
mountain
forest
sea
glacier
sea
street
sea
street
mountain
buildings
buildings

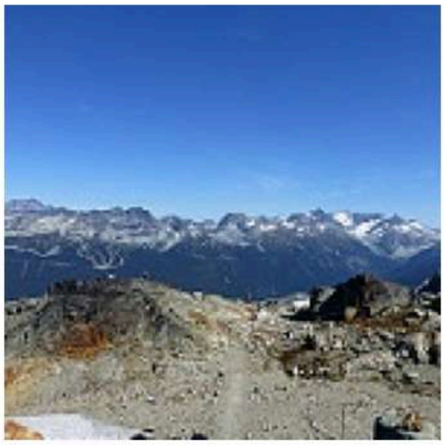


glacier

forest

forest

mountain



sea

street

street

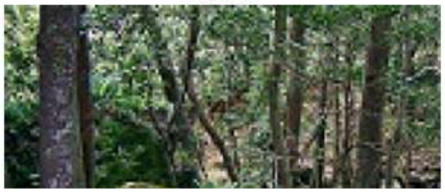
street

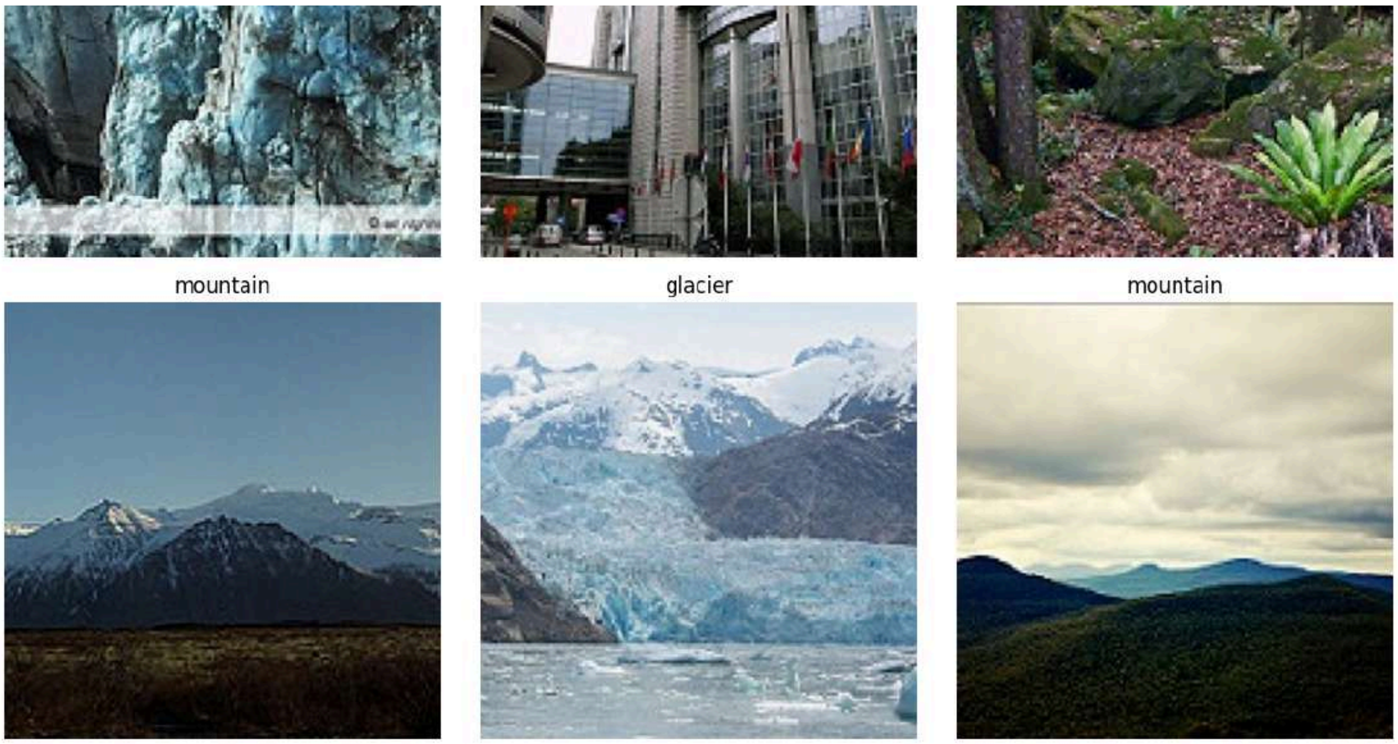
buildings

mountain

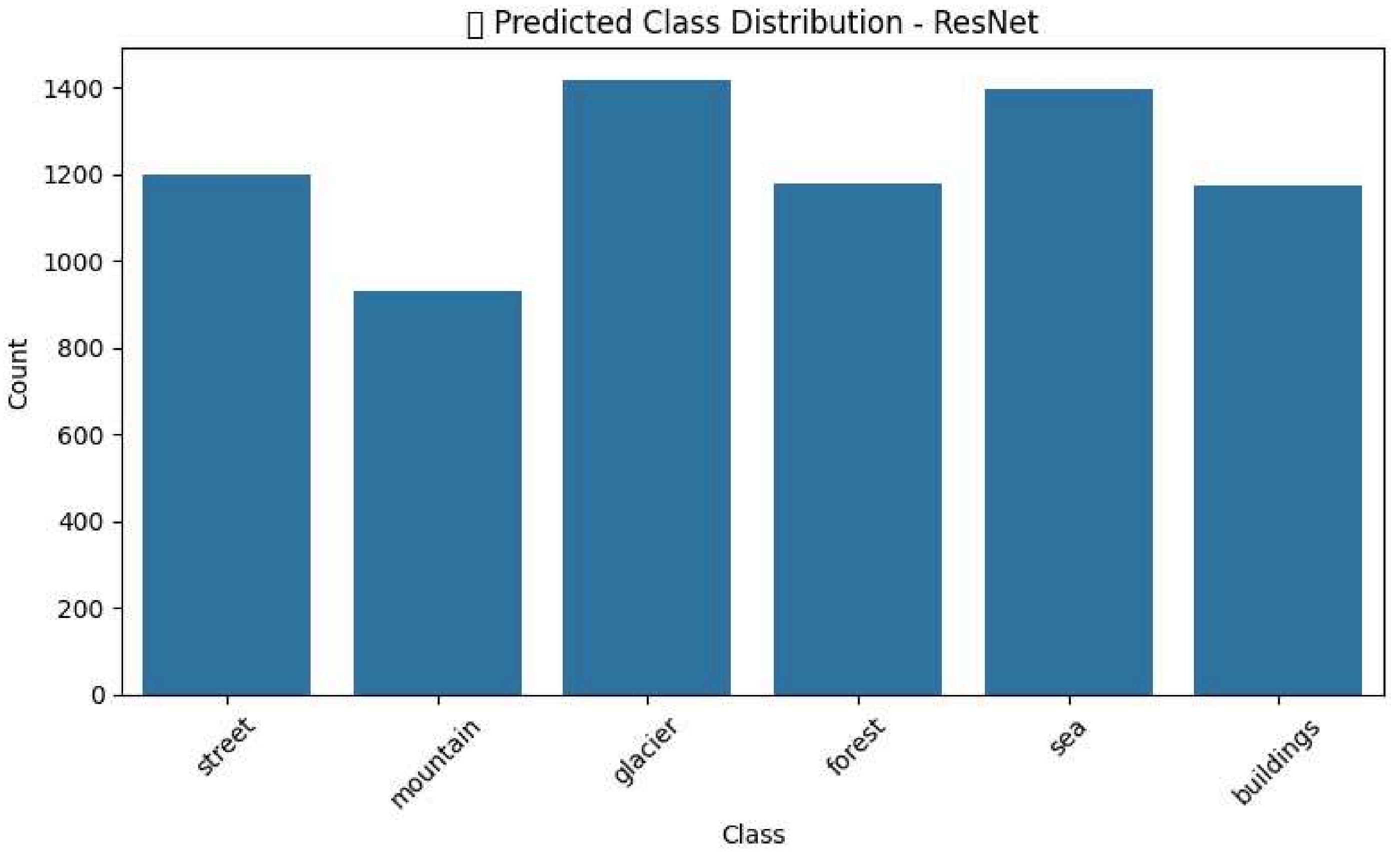
buildings

forest

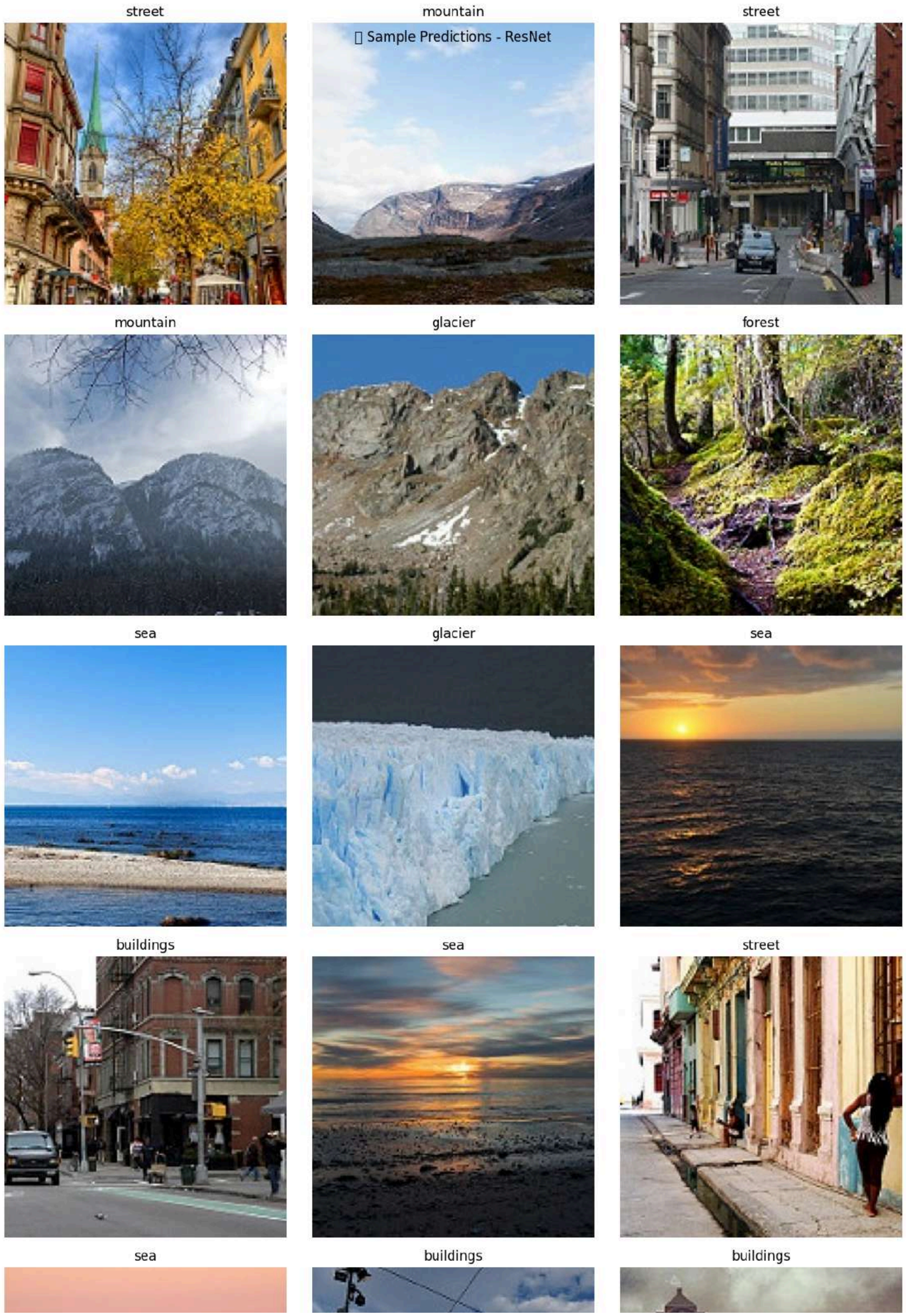


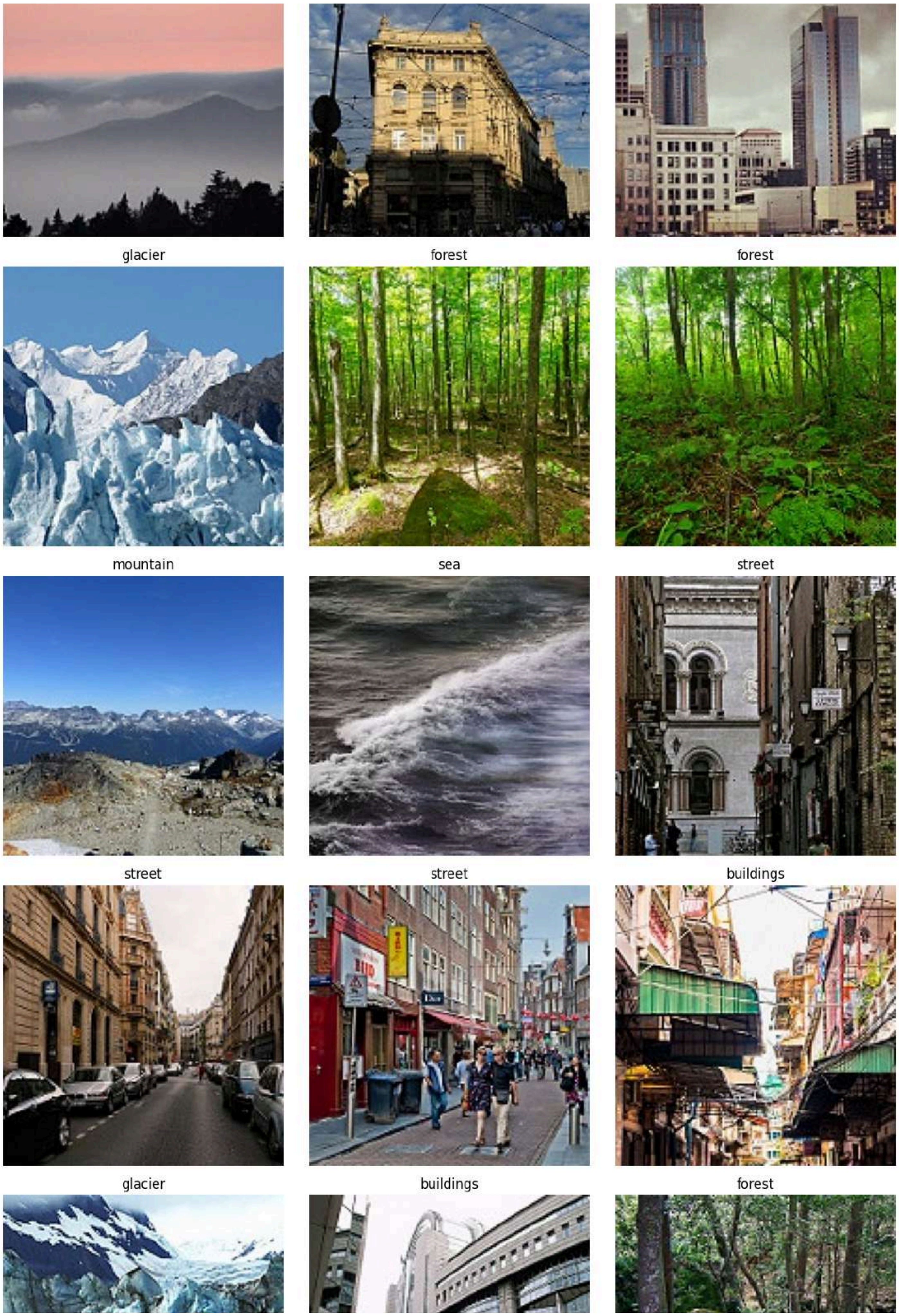


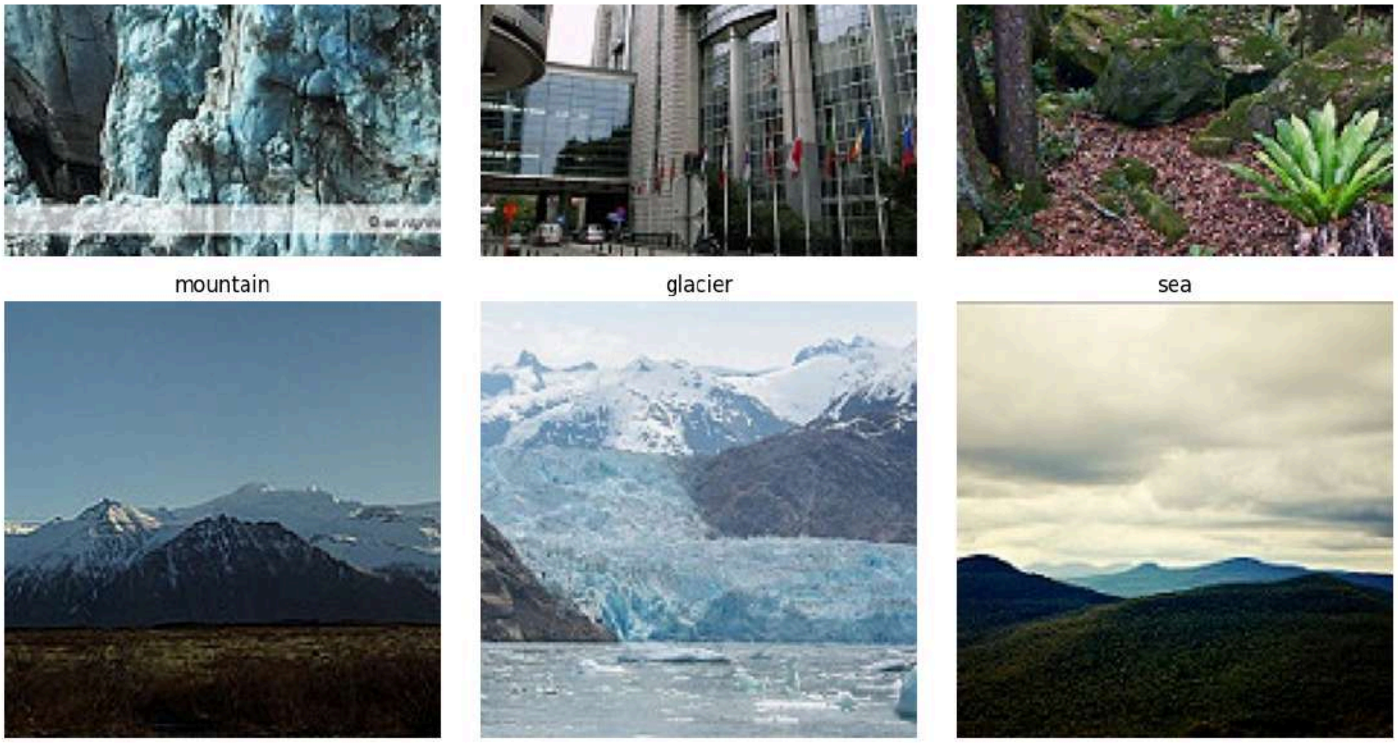
```
C:\Users\user\AppData\Local\Temp\ipykernel_38124\978616877.py:16: UserWarning: Glyph 128202 (\N{B
AR CHART}) missing from font(s) DejaVu Sans.
  plt.tight_layout()
C:\Users\user\anaconda3\envs\rl-assignment\lib\site-packages\IPython\core\pylabtools.py:170: User
Warning: Glyph 128202 (\N{BAR CHART}) missing from font(s) DejaVu Sans.
  fig.canvas.print_figure(bytes_io, **kw)
```



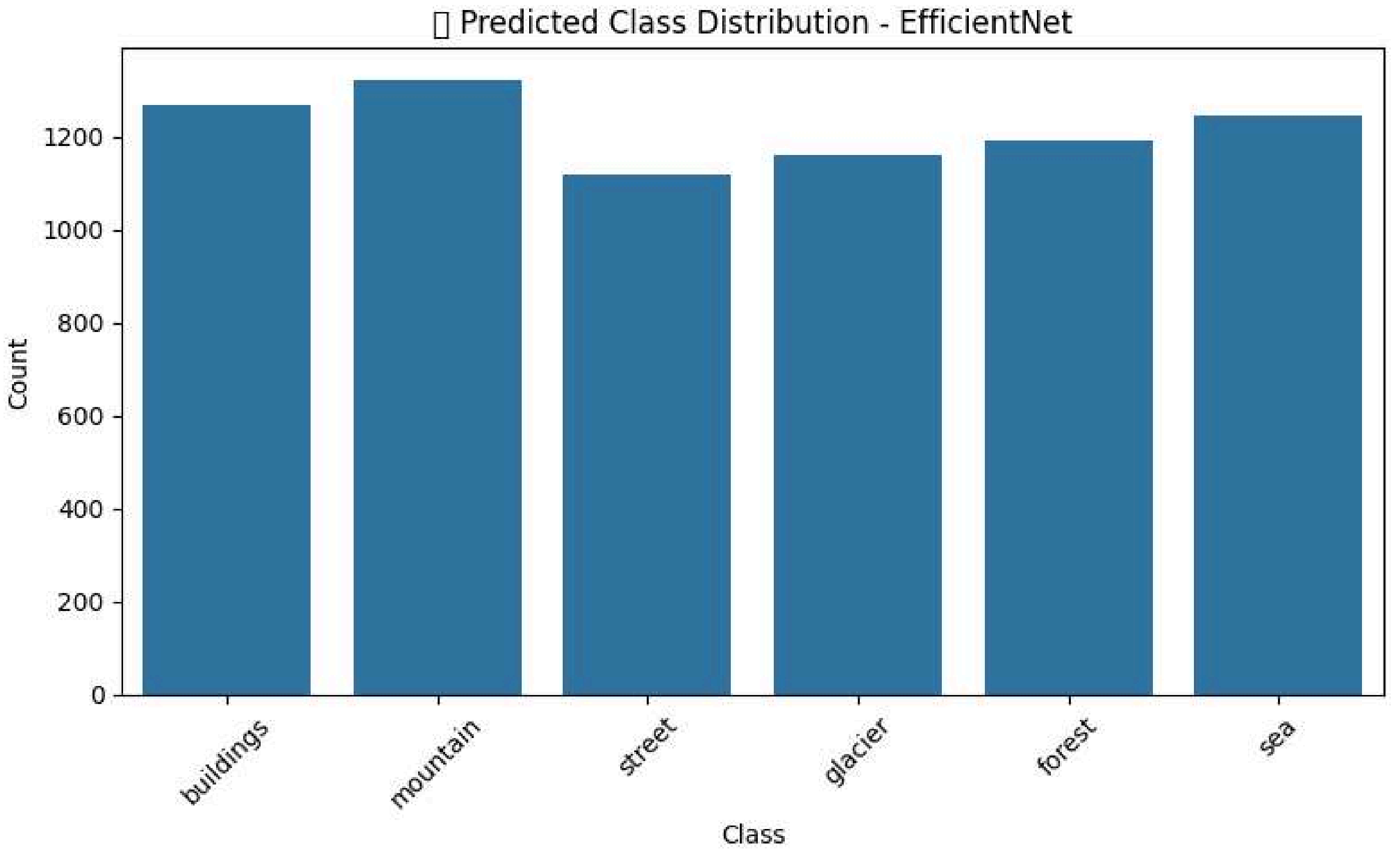
```
C:\Users\user\AppData\Local\Temp\ipykernel_38124\978616877.py:42: UserWarning: Glyph 128269 (\N{L
EFT-POINTING MAGNIFYING GLASS}) missing from font(s) DejaVu Sans.
  plt.tight_layout()
C:\Users\user\anaconda3\envs\rl-assignment\lib\site-packages\IPython\core\pylabtools.py:170: User
Warning: Glyph 128269 (\N{LEFT-POINTING MAGNIFYING GLASS}) missing from font(s) DejaVu Sans.
  fig.canvas.print_figure(bytes_io, **kw)
```


Sample Predictions - ResNet
street
mountain
street
mountain
glacier
forest
sea
glacier
sea
buildings
sea
street
sea
buildings
buildings

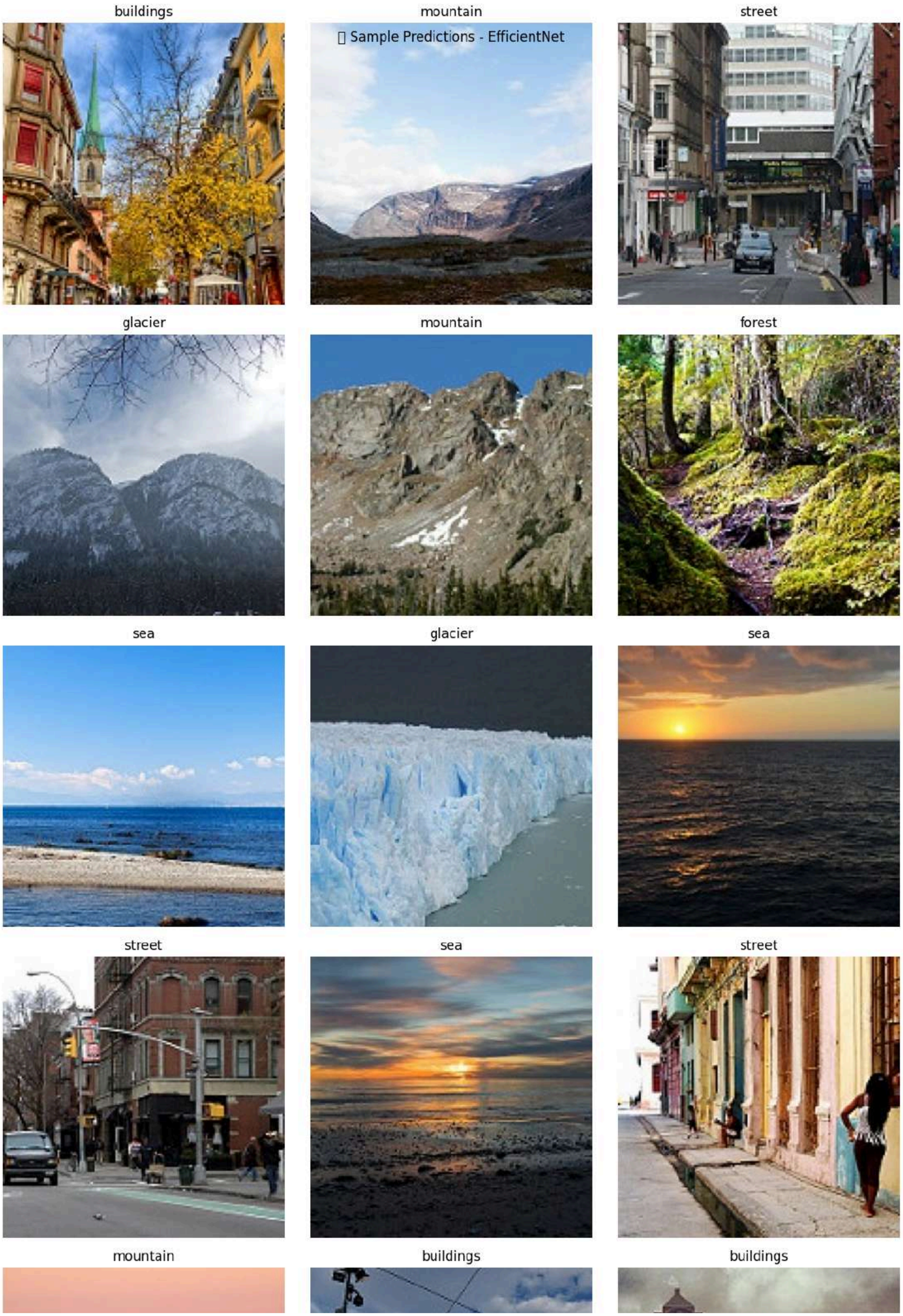

glacier
forest
forest
mountain
sea
street
street
street
buildings
glacier
buildings
forest

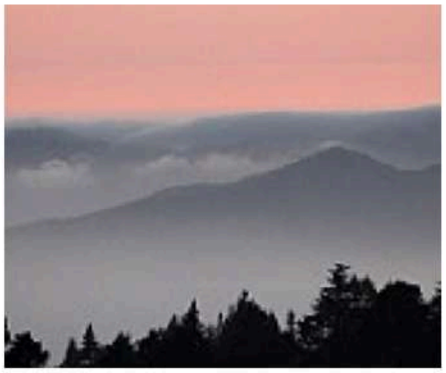


```
C:\Users\user\AppData\Local\Temp\ipykernel_38124\978616877.py:16: UserWarning: Glyph 128202 (\N{B
AR CHART}) missing from font(s) DejaVu Sans.
  plt.tight_layout()
C:\Users\user\anaconda3\envs\rl-assignment\lib\site-packages\IPython\core\pylabtools.py:170: User
Warning: Glyph 128202 (\N{BAR CHART}) missing from font(s) DejaVu Sans.
  fig.canvas.print_figure(bytes_io, **kw)
```



```
C:\Users\user\AppData\Local\Temp\ipykernel_38124\978616877.py:42: UserWarning: Glyph 128269 (\N{L
EFT-POINTING MAGNIFYING GLASS}) missing from font(s) DejaVu Sans.
  plt.tight_layout()
C:\Users\user\anaconda3\envs\rl-assignment\lib\site-packages\IPython\core\pylabtools.py:170: User
Warning: Glyph 128269 (\N{LEFT-POINTING MAGNIFYING GLASS}) missing from font(s) DejaVu Sans.
  fig.canvas.print_figure(bytes_io, **kw)
```


Sample Predictions - EfficientNet
buildings
mountain
street
glacier
mountain
forest
sea
glacier
sea
street
sea
street
mountain
buildings
buildings

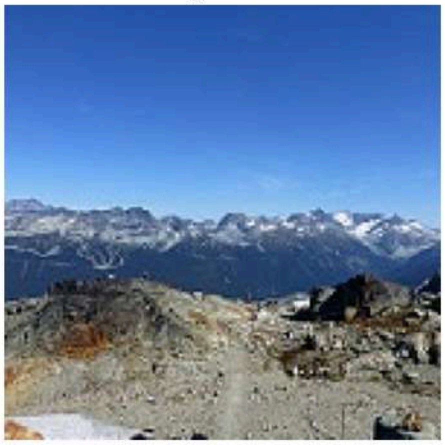


glacier

forest

forest

glacier



sea

street

street

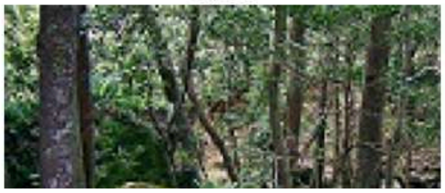
street

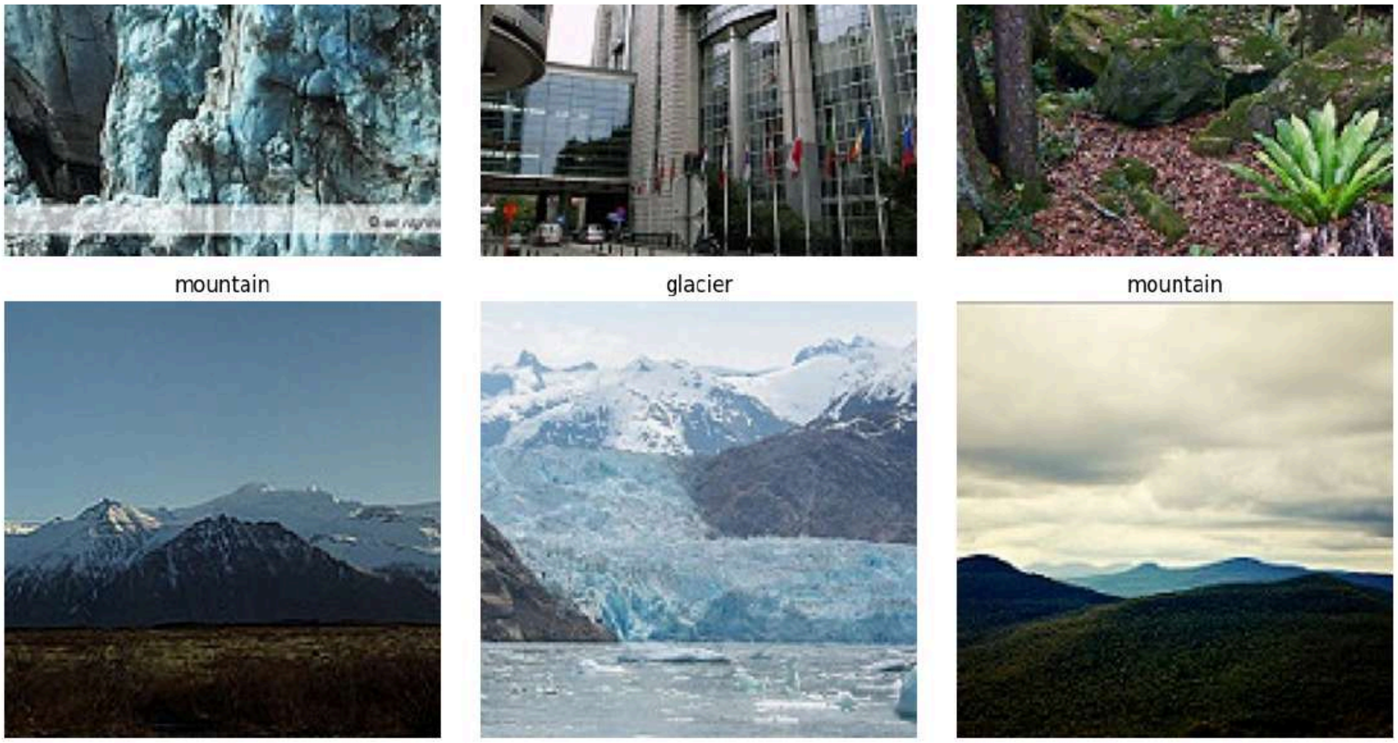
street

glacier

buildings

forest





Bar Chart interpretation

CNN Predictions on `seg_pred`

- CNN predictions are fairly balanced, though there is a **slight bias toward the `mountain` class** (most frequently predicted).
- The prediction for other classes appear to be more balanced.

ResNet Predictions on `seg_pred`

- ResNet shows a more **pronounced class imbalance**:
  - Overpredicts `glacier` and `sea`
  - Underpredicts `mountain`
- This may indicate that ResNet is more sensitive to certain low-level features in `glacier` and `sea`, and tends to **confuse `mountain` with visually similar classes** like `glacier` or `forest`.

EfficientNet Predictions on `seg_pred`

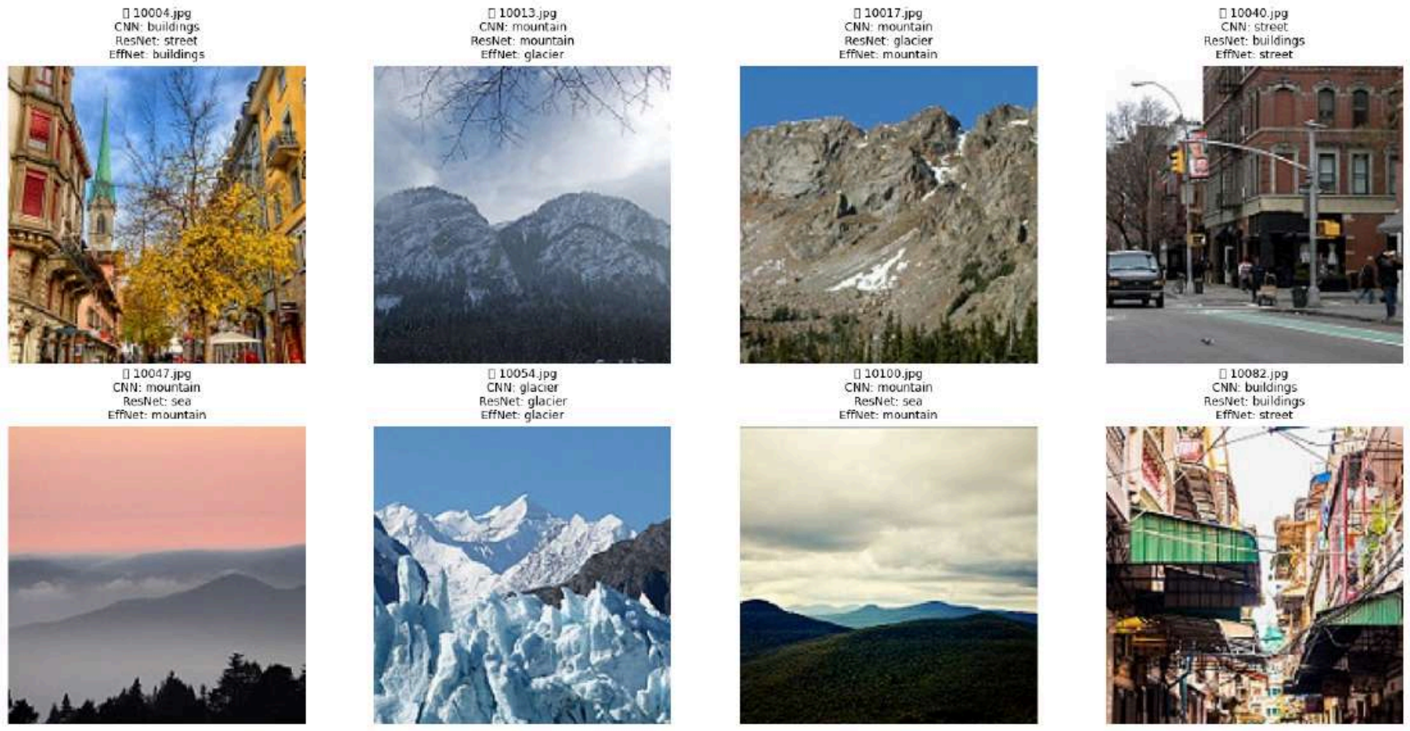
- EfficientNet demonstrates the **most balanced class distribution** among all three models.
- Slightly favors `mountain` and `buildings`, but the differences across classes are minimal.
- This reflects **stronger generalization capabilities**, likely due to EfficientNet's compound scaling and deeper architecture.

Model	Most Predicted Class	Least Predicted Class	Notes
CNN	Mountain	Glacier	Slight preference toward natural scenes
ResNet	Glacier, Sea	Mountain	Confuses mountain with glacier/forest
EfficientNet	Mountain, Buildings	Street	Most balanced; good generalization

These prediction patterns reveal each model's tendencies, biases, and generalization strengths. Even though all were trained on a balanced dataset, differences in **architecture depth**, **feature extraction**, and **inductive biases** shape how each model performs on unseen data like `seg_pred`.

In [94]:
```
# List of 1-based indices to visualize (not in order)
selected_indices = [1, 4, 5, 10, 13, 16, 30, 24]
selected_indices = [i - 1 for i in selected_indices]  # Convert to 0-based indexing

# Plotting setup
n_samples = len(selected_indices)
n_cols = 4
n_rows = math.ceil(n_samples / n_cols)

fig, axs = plt.subplots(n_rows, n_cols, figsize=(16, 4 * n_rows))
axs = axs.flatten()

# Loop through each selected image index
for plot_idx, sample_idx in enumerate(selected_indices):
    row = sample_results[sample_idx]
    img_path = os.path.join(pred_dir, row['Filename'])
    img = Image.open(img_path)

    # Get predicted labels from each model
    filename = row['Filename']
    pred_cnn = next((r['Predicted_Class'] for r in all_results['CNN'] if r['Filename'] == filena
    pred_resnet = next((r['Predicted_Class'] for r in all_results['ResNet'] if r['Filename'] ==
    pred_efficientnet = next((r['Predicted_Class'] for r in all_results['EfficientNet'] if r['Fi

    # Display image and predictions
    axs[plot_idx].imshow(img)
    axs[plot_idx].axis('off')
    axs[plot_idx].set_title(
        f"📷 {filename}\n"
        f"CNN: {pred_cnn}\n"
        f"ResNet: {pred_resnet}\n"
        f"EffNet: {pred_efficientnet}",
        fontsize=9
    )

# Hide any remaining empty subplots
for i in range(len(selected_indices), len(axs)):
    axs[i].axis('off')

plt.tight_layout()
plt.show()
```

```
C:\Users\user\AppData\Local\Temp\ipykernel_38124\2591256299.py:40: UserWarning: Glyph 128247 (\N
{CAMERA}) missing from font(s) DejaVu Sans.
  plt.tight_layout()
C:\Users\user\anaconda3\envs\rl-assignment\lib\site-packages\IPython\core\pylabtools.py:170: User
Warning: Glyph 128247 (\N{CAMERA}) missing from font(s) DejaVu Sans.
  fig.canvas.print_figure(bytes_io, **kw)
```



Based on the first 30 predicted image samples, CNN demonstrated perfect prediction accuracy, correctly classifying all images. In contrast, ResNet50 and EfficientNet showed occasional misclassifications, indicating they struggled with some of the samples.

The 8 images displayed above were selected to highlight instances where ResNet and EfficientNet misclassified the input images.

Visual Analysis of Selected Image Predictions

The visual analysis of the selected image predictions (from **CNN**, **ResNet**, and **EfficientNet**) reveals a few key patterns and model behaviors:

General Observations

1. **Class Confusion in Urban Scenes**:

   - `10004.jpg`, `10040.jpg`, and `10082.jpg` are clearly **urban scenes**.
     - `10004.jpg` is more accurately classified as a building, given the absence of visual cues typically associated with a **street**, such as vehicles, pedestrians, or road markings. Among the three models, only the CNN correctly identified this image.
     - ResNet and EfficientNet correctly predicted **street** for `10040.jpg`, but ResNet misclassified it as **buildings**.
     - Similar to `10004.jpg`, `10082.jpg` lacks key street indicators such as vehicles, pedestrians, or road markings, and is therefore more appropriately classified as a building. However, EfficientNet failed to correctly identify this image.
   - These inconsistencies indicate that the model struggles to distinguish between **street** and **buildings**, particularly when both elements may be present but only partially visible, such as when the street portion appears to be cropped out in images primarily showing buildings.
2. **Mountain vs Glacier Confusion**:

   - `10013.jpg` and `10017.jpg` are mountainous snowy landscapes.

        - CNN and ResNet predicted **mountain**, but EfficientNet predicted **glacier** for `10013.jpg`.
        - ResNet predicted **glacier** for `10017.jpg`, although the terrain looks more **mountainous**.
    - This confusion is likely due to **snowy textures** being present in both glacier and mountain classes, making them visually similar.
3. **Scene Misinterpretation by ResNet**:
    - `10047.jpg` shows a mountainous scene, with trees.
        - CNN and EfficientNet both predict **mountain**, as **trees and background hills** can clearly be seen.
        - ResNet surprisingly predicts **sea**, which seems inaccurate.
    - `10100.jpg`, another hilly forest scene, is interpreted by ResNet as **sea**, which again suggests a potential misreading of **hazy backgrounds or low-contrast areas**.
4. **High Consensus; Glacier**:
    - `10054.jpg` has **strong agreement**: all models predicted **glacier**, likely due to the clear presence of ice structures.
    - This indicates that **glacier** is a well-learned class for all three models when the ice formations are distinct, despite the presence of a mountain in the background.

Key Takeaways

Observation Category	Image(s) Involved	Misclassification Details	Likely Cause
Class Confusion in Urban Scenes	`10004.jpg`, `10040.jpg`, `10082.jpg`	- `10004.jpg`: CNN predicted **buildings** correctly; ResNet & EfficientNet misclassified as **street**. - `10040.jpg`: ResNet misclassified as **buildings**; others predicted **street** correctly. - `10082.jpg`: Only CNN predicted **buildings** correctly; EfficientNet misclassified as **street**.	Lack of street indicators (cars, road, pedestrians) and cropped street views in building scenes cause confusion between **buildings** and **street**.
Mountain vs Glacier Confusion	`10013.jpg`, `10017.jpg`	- `10013.jpg`: CNN & ResNet predicted **mountain**; EfficientNet predicted **glacier**. - `10017.jpg`: ResNet misclassified as **glacier**; others predicted **mountain**.	Snowy textures present in both **mountain** and **glacier** classes contribute to visual similarity.
Scene Misinterpretation by ResNet	`10047.jpg`, `10100.jpg`	- `10047.jpg`: CNN & EfficientNet correctly predicted **mountain**; ResNet misclassified as **sea**. - `10100.jpg`: Forested hills misclassified as **sea** by ResNet.	Possibly due to **hazy backgrounds** or **low-contrast areas** being misread as water bodies.
High Consensus – Glacier	`10054.jpg`	- All models correctly predicted **glacier**.	Distinct ice formations make this an **easily recognizable** glacier scene

Observation Category	Image(s) Involved	Misclassification Details	Likely Cause
			for all models.

6.0 Future Work

While this project has successfully demonstrated the effectiveness of CNN, ResNet50, and EfficientNet-B0 in classifying natural scene images from the Intel dataset, there are several areas for potential improvement and future exploration:

6.1 Fine-Tuning Pretrained Models

In this study, both ResNet50 and EfficientNet-B0 had their feature extraction layers frozen during training, with only the classifier layers updated. While this approach speeds up training and reduces overfitting, it may limit the models' ability to adapt to dataset-specific patterns. Future work should explore:

- **Unfreezing deeper layers gradually** using a discriminative learning rate schedule.
- **Full fine-tuning** for larger training epochs to exploit the full representational power of pretrained weights.

6.2 Expand Evaluation Metrics

Although accuracy,macro F1-score, precision, confusion matrices and recall provided meaningful comparisons, future experiments could benefit from a more comprehensive evaluation by incorporating:

- **Class-wise ROC-AUC**, particularly for binary or grouped class tasks
- **Model robustness checks** using adversarial examples or image augmentations

6.3 Data Augmentation and Regularization

More advanced data augmentation techniques could improve model generalization, particularly for underrepresented classes such as **glacier** or **street**:

- **AutoAugment**, **CutMix**, or **MixUp**
- Additional regularization such as **label smoothing**, **dropblock**, or **stochastic depth**

6.4 Addressing Class Confusion

Error analysis revealed frequent misclassification between:

- **Buildings vs Street**
- **Mountain vs Glacier**

Future strategies may include:

- **Class-specific data balancing**
- **Multi-task learning** to learn shared and distinct features across related classes
- **Attention mechanisms** to help models focus on relevant image regions

6.5 Larger-Scale Evaluation on Unlabelled Data

The inference task on 7,301 unlabelled `seg_pred` images provided further insight into real-world generalization. Future efforts should include:

- **Manual annotation or semi-supervised labelling** of unlabelled samples to validate predictions.
- **Model ensembling** to reduce individual model biases and increase prediction confidence.

6.6 Deployment Considerations

For real-world use:

- **Model compression** (pruning, quantization) could reduce inference time.
- **On-device deployment testing** for edge or mobile scenarios can be explored, especially for the lightweight CNN model.

Overall, this project sets a strong baseline for natural scene classification using both custom and pretrained deep learning models, while highlighting multiple paths for enhancing accuracy, generalization, and deployment readiness.

7.0 Conclusion

Based on the **evaluation using the labeled `seg_test` dataset**, **ResNet50** achieved the highest performance, with both **Accuracy** and **F1-Score** around **0.88**, indicating strong and consistent classification across all six scene classes. **CNN** followed closely with scores around **0.86**, while **EfficientNet-B0** lagged slightly behind at **0.85**, likely due to underfitting caused by freezing too many layers or insufficient fine-tuning.

However, when analyzing the **predictions on the 7,301 unlabeled images from the `seg_pred` folder**, the performance dynamics appeared different:

- **CNN** produced the most accurate predictions among the **first 30 visually inspected samples**, often correctly classifying difficult cases (distinguishing buildings from streets, and mountains from glaciers). This suggests strong instance-level performance and generalization despite being a simpler model.

- **EfficientNet-B0**, although less accurate on those 30 samples, demonstrated the **most balanced class distribution** across the entire `seg_pred` dataset. This indicates that while it may not always predict individual images correctly, it **avoids class bias**, potentially making it better suited for tasks requiring fairness across all categories.

- **ResNet50**, despite its top testing performance, showed **some inconsistencies in generalization** during `seg_pred` predictions. While it maintained good accuracy in several cases, it occasionally misclassified scenes (labeling mountainous forests as "sea"), suggesting **sensitivity to ambiguous or low-contrast features** in real-world images.

- **ResNet50** is the most **accurate and stable model** when evaluated on labeled test data (`seg_test`).
- **CNN** demonstrates **strong generalization** on unseen examples (`seg_pred`) and performs well in practical scenarios.
- **EfficientNet-B0** provides the **most balanced class predictions** on a large volume of unlabeled data, though it may require deeper fine-tuning to improve accuracy on specific instances.

Summary Table: Model Comparison

Model	Strengths	Weaknesses	Scalability	Suitability
CNN	- Fast training and inference time - High accuracy on small sample tests - Simple to implement and tune	- Lower generalization on full dataset - May not capture deep spatial patterns as well as deeper models	Lightweight, deployable on edge devices	Suitable for mobile or resource-constrained environments where model size and speed matter
ResNet50	- Highest accuracy & F1 score on test set - Stable loss curves - Pretrained backbone improves learning	- Occasional misclassifications on ambiguous scenes - Slightly larger model size	Moderate (requires GPU for efficient training)	Best for general-purpose, balanced performance use cases, especially in production environments
EfficientNet-B0	- More balanced predictions across all 7,301 unlabelled images - Compact yet deep architecture	- Underperformed in test set - Shows signs of underfitting due to excessive freezing	Moderate (higher training time, needs careful fine-tuning)	Ideal for large-scale deployment where uniform prediction distribution is important, but requires tuning