Experimental Protocol

1.1 Dataset Overview

The dataset used in this experiment is the UCMerced Land-Use dataset, which consists of 2,100 aerial images, each with a size of 256×256 pixels, categorized into 21 land-use classes.

1.2 Data Management and Loading

```
DataPath = 'UCMerced_LandUse/Images'
try:
  Dataset = datasets.ImageFolder(root = DataPath)
except:
  print("The dataset path is wrong, please check the path!")
  exit()
```

- The dataset is stored in data_path, and ImageFolder is used to load it. A try-except block checks if the dataset path is correct, avoiding runtime errors.

1.3 Hyperparameter Settings

```
Inputsize = 256
```

- Matches the dataset's native resolution, avoiding unnecessary resizing.

```
batch_size = 32
```

- Balances memory usage and training stability.

```
Numworkers = min(4, os.cpu_count())
```

- Enables parallel data loading, improving efficiency. balances memory usage and training stability.

```
NFolds = 5
kf = KFold(n_splits=NFolds, shuffle=True, random_state=RandomSeed)
```

- Sets up 5-fold cross-validation, providing a robust evaluation by using 80% of data for training and 20% for testing per fold. The decision to use 5-fold cross-validation was motivated by its ability to provide a robust and reliable evaluation of the model's performance. Given the dataset's moderate size, a single train-test split could lead to biased results due to variability in the data distribution across splits. By averaging performance across five folds, this method reduces overfitting risks and ensures that the model is evaluated on all samples, maximizing the use of available data. The 80%-20% split per fold strikes a

balance between sufficient training data to learn the 21-class classification task and a substantial test set to assess generalization.

1.4 Data Transform

```
TrainTransform = transforms.Compose([
    transforms.Resize((Inputsize, Inputsize)),
    transforms.Lambda(lambda x: x.convert('RGB') if isinstance(x, Image.Image) else x),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomVerticalFlip(p=0.5),
    transforms.RandomRotation(15),
    transforms.ColorJitter(
        brightness=0.2,
        contrast=0.2,
        saturation=0.2,
        hue=0.1
    ),
    transforms.ToTensor(),
    transforms.Normalize(mean=Mean, std=STD)
])
```

- First, resize all input images to a uniform size.
- Make sure the image is in RGB three-channel format. Because .tif images usually contain multiple bands and can be very large and contain geographic coordinates, they are often stored as floating point numbers or other special formats and cannot be directly input into deep learning models.
- Randomly flip the image horizontally or vertically with 50% probability and randomly rotate the image within ±15 degrees to improve the model's robustness to direction changes and enhance data diversity.
- The brightness change range, contrast change range, saturation change range ±20%, and hue change range ±0.1 are used to simulate different lighting conditions to improve the model's robustness to color changes.
- Convert a PIL.Image or NumPy array to a PyTorch Tensor and automatically normalize pixel values to the [0, 1] range. Normalization scales pixel values to a consistent range, improving model convergence and performance.

```
TestTransform = transforms.Compose([
    transforms.Resize((Inputsize, Inputsize)),
    transforms.Lambda(lambda x: x.convert('RGB') if isinstance(x, Image.Image) else x),
    transforms.ToTensor(),
    transforms.Normalize(mean=Mean, std=STD)
])
```

- Note: The above transformations are only for the training set. The test set only needs deterministic preprocessing (such as Resize, ToTensor, Normalize) and no random enhancement is required to ensure stable evaluation results.

1.5 Data Loader and Visualization

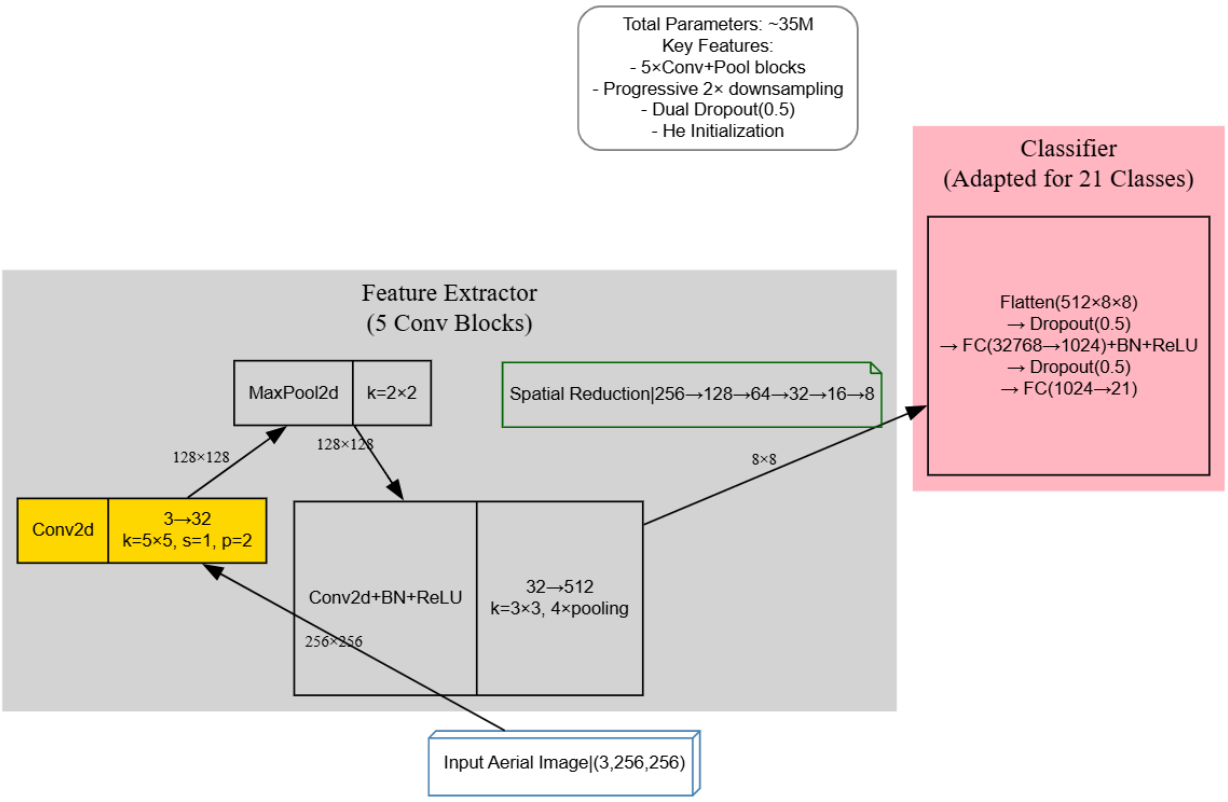
```
Foldloaders = []
for fold, (TrainIdx, TestIdx) in enumerate(kf.split(Indices)):
    print(f"=== Preparing Fold {fold+1}/{NFolds} ===")

    TrainDataset = datasets.ImageFolder(root=DataPath, transform=TrainTransform)
    TestDataset = datasets.ImageFolder(root=DataPath, transform=TestTransform)

    TrainSubset = Subset(TrainDataset, TrainIdx)
    TestSubset = Subset(TestDataset, TestIdx)

    TrainLoader = DataLoader(TrainSubset, batch_size=batch_size, shuffle=True, num_workers=Numworkers)
    TestLoader = DataLoader(TestSubset, batch_size=batch_size, shuffle=False, num_workers=Numworkers)

    Foldloaders.append({
        'Train': TrainLoader,
        'Test': TestLoader,
        'TrainIdx': TrainIdx,
        'TestIdx': TestIdx
    })
```

- First, split the indices into NFolds partitions. For each fold, it provides a set of training indices (TrainIdx) and testing indices (TestIdx).
- TrainDataset and TestDataset are instantiated using datasets.ImageFolder, which loads images from a specified directory (DataPath). These datasets apply distinct transformations (TrainTransform and TestTransform) to preprocess the images.
- TrainSubset and TestSubset create subsets of the dataset based on the indices provided by the K-fold split. This ensures that each fold uses a unique combination of training and testing data.
- TrainLoader and TestLoader are created using DataLoader, which batches the data, shuffles the training set, and parallelizes data loading with num_workers. The testing loader does not shuffle to maintain consistency during evaluation.The batch_size parameter controls the number of samples per batch, optimizing memory usage and training efficiency.
- For each fold, a dictionary is created containing: 'Train', 'Test', 'TrainIdx', 'TestIdx'. This dictionary is appended to the Foldloaders list, resulting in a collection of all fold-specific data loaders and indices.

Neural Networks

2.1 Custom CNN architecture

- LandCNN class inherits from nn.Module and consists of three main components:
    - Feature Extractor
    - Classifier
    - Weight Initialization

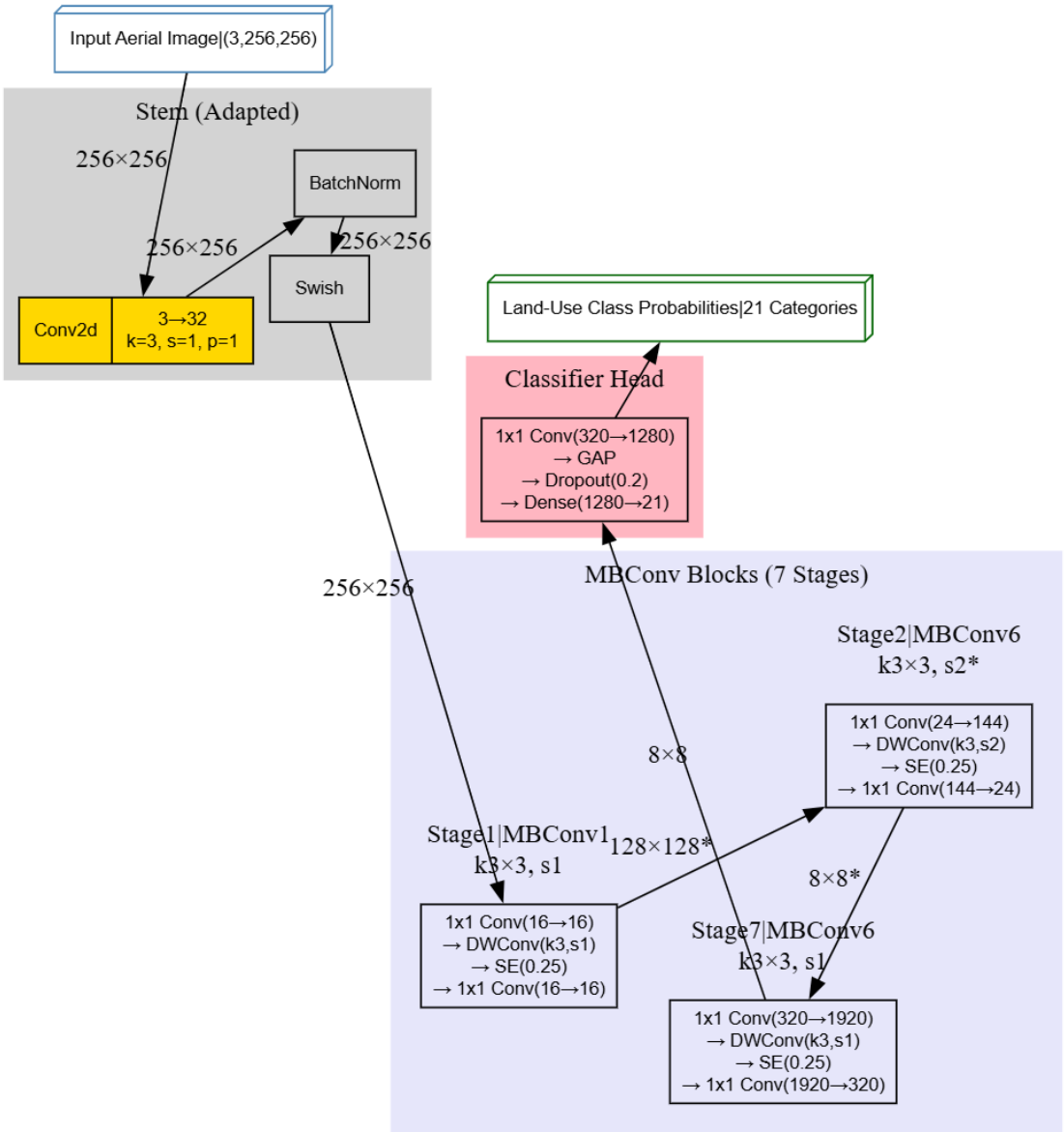


- Working principle and module overview

Component	Details
Feature Extraction Backbone	5-layer convolutional block with increasing channels: **32 → 64 → 128 → 256 → 512**
	Each convolutional layer includes:
	- **Kernel Sizes**: 5×3 (first layer), 3×3 (subsequent layers) with **padding** to maintain spatial dimensions
	- **Batch Normalization**: Stabilizes training
	- **ReLU Activation**: Introduces non-linearity
	- **2×2 Max Pooling**: Reduces dimensionality
	- Overall **spatial size reduction**: **256×256 → 8×8** (**32× reduction**)
Classification Head	Two **fully connected layers** with **1024** hidden units
	Includes:
	- **Dropout (p=0.5)**: Prevents overfitting
	- **Batch Normalization**: Speeds up convergence
	- **Final Output Layer**: Produces logits for **21** land use categories
Weight Initialization	- **Conv Layers**: Kaiming Normal (ReLU-optimized)
	- **BatchNorm Layers**: γ=1, β=0 initialization
	- **Linear Layers**: Small Normal Initialization (σ=0.01)
Key Design Choices	- Expanding **channel dimensions** to increase feature complexity while reducing spatial size
	- **Smaller 3×3 kernels** in deeper layers to preserve receptive field while reducing parameters
	- **Final 8×8 feature map size** balances information retention and efficiency
	- **Aggressive dropout (50%)** to reduce overfitting in the fully connected layers
Forward Propagation	**FeatureExtractor → Flatten → Classifier**

2.2 Custom CNN architecture

- For this experiment, EfficientNet-B0 was chosen as the existing convolutional neural network.
- The working principle of EfficientNet-B0 is briefly described as follows:The workflow begins with a 256×256 aerial image passed through an adapted Stem block to preserve fine-grained spatial details. Seven MBConv blocks then process the features hierarchically: early stages maintain high resolution for local texture, while later stages progressively downsample using depthwise separable convolutions and Squeeze-Excitation attention to capture global context. The classifier head employs 1×1 channel expansion, global pool averaging, and dropout before outputting 21 land-use classes.



EfficientNet-B0

2.3 Components

```
ClassCounts = np.bincount(Labelarray)
ClassWeights = 1. / torch.tensor(ClassCounts, dtype=torch.float).to(device)
Criterion = nn.CrossEntropyLoss(weight=ClassWeights)
```

- Weighted Cross Entropy Loss: The dataset may have imbalanced class distributions, and weighting mitigates bias toward majority classes, improving overall classification fairness. The small constant (1e-6) prevents division by 0.

```
Optimizer = AdamW(params=ParaGps, lr=3e-4, weight_decay=0.05, betas=(0.9, 0.999))
```

- AdawAdamW combines adaptive learning rates with weight decay regularization (weight_decay=0.05, lr=3e-4, betas= (0.9, 0.999)). And lr=3e-4 balances fast convergence and stability, while weight_decay=0.05 prevents overfitting on this moderately sized dataset. Default betas ensure efficient momentum updates.

```
Scheduler = CosineAnnealingWarmRestarts(Optimizer, T_0=20, T_mult=2)
```

- Use The scheduler adjusts the learning rate with a cosine decay, restarting periodically. This allows sufficient epochs for initial learning and enable exploration of loss landscapes, preventing stagnation in local minima.

- Scales gradients enabled only on CUDA to leverage faster computation and

```
Scaler = torch.amp.GradScaler(enabled=(device.type == 'cuda'))
```

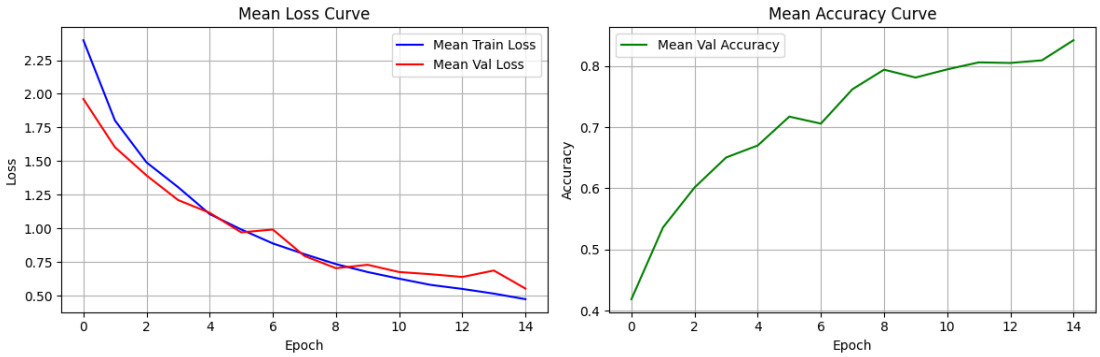
```
class EarlyStopping:
  def __init__(self, Patience=7, Delta=0.001):
    self.Patience = Patience
    self.Delta = Delta
    self.Counter = 0
    self.BestScore = None
    self.EarlyStop = False
  def __call__(self, ValLoss, Model):
    Score = -ValLoss
    if self.BestScore is None:
      self.BestScore = Score
      self.SaveCheckpoint(ValLoss, Model)
    elif Score < self.BestScore + self.Delta:
      self.Counter += 1
      if self.Counter >= self.Patience:
        self.EarlyStop = True
    else:
      self.BestScore = Score
      self.SaveCheckpoint(ValLoss, Model)
      self.Counter = 0
  def SaveCheckpoint(self, ValLoss, Model):
    print(f"Validation loss improved: {self.BestScore:.4f} → {ValLoss:.4f}. Saving model...")
    torch.save(Model.state_dict(), 'Checkpoint.pt')
```

reduced memory usage.

- Stops training if validation loss doesn't improve, Patience=7 allows reasonable recovery time, while Delta=0.001 ensures meaningful progress, preventing overfitting and reduces training time.

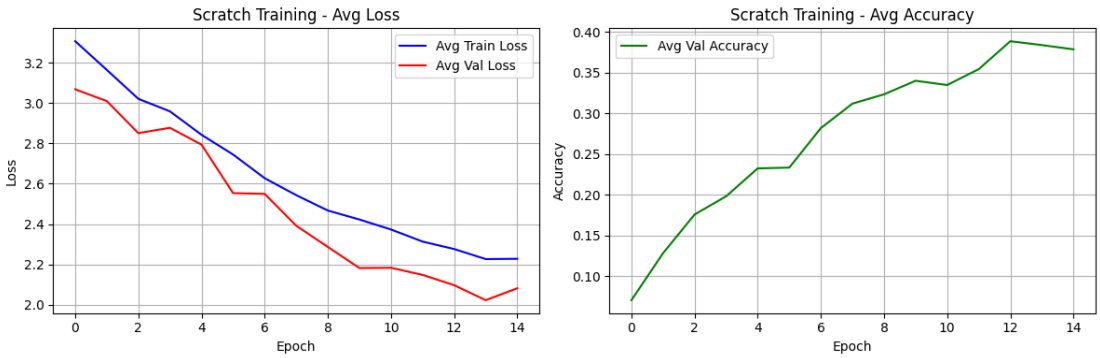
2.4 Results

My Own CNN:



- Achieved steady improvement across folds, with validation accuracy peaking at 85.95%. The model learns effectively, leveraging data augmentation and weighted loss to handle the 21-class task. However, accuracy plateaus around 85%, suggesting limited capacity to capture complex features compared to deeper architectures.
- Built from scratch, this simpler architecture lacks the depth and pre-learned features of EfficientNet. Despite effective training with data augmentation and weighted cross-entropy, its capacity to extract complex patterns from the UCMerced dataset is limited. The absence of pre-trained knowledge means it relies solely on the dataset's 2100 images, insufficient for optimal feature learning within 15 epochs.

Scratch Learning:

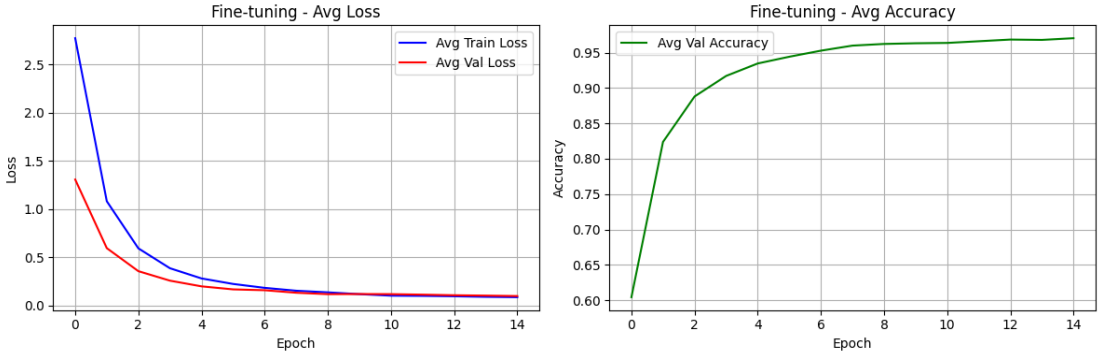


- Progress was slow, training loss was high and validation loss hovered around 2.0.

Starting with random weights, EfficientNet struggles to converge within 15 epochs, likely due to its depth and the dataset's moderate size (2100 images). This indicates insufficient training time or data for full optimization.

- Training EfficientNet-B0 from scratch struggles due to its deep, parameter-heavy design, requiring extensive data and epochs to converge. With a mean accuracy of 38.71%, it fails to leverage the dataset effectively, as random initialization and the moderate dataset size hinder feature development, even with advanced strategies like cosine annealing and AdamW.

Fine-tuning Learning:



- Demonstrated rapid convergence and superior results, leveraging pre-trained weights, fine-tuning adapts EfficientNet's robust feature extraction to the task, achieving near-perfect accuracy. The lower learning rate (1e-4) ensures stable updates, enhancing generalization.
- Fine-tuning starts with pre-trained weights from a large, diverse dataset (e.g., ImageNet), providing robust initial features. This approach, combined with a lower learning rate (1e-4) and targeted parameter updates, adapts these features efficiently to the 21-class task, achieving 97.24% mean accuracy. The pre-trained knowledge accelerates convergence and enhances generalization, outpacing scratch training's data-limited progress.

Fold	Custom CNN	EfficientNet (Scratch)	EfficientNet (Fine-tuned)
1	85.95	38.81	98.57
2	84.52	39.76	96.43
3	85.71	39.52	96.19
4	81.19	35.95	97.38
5	84.76	39.52	97.62
Mean	**84.43**	**38.71**	**97.24**

- In summary, fine-tuning's pre-trained foundation and strategic adaptation outperform the custom CNN's limited capacity and scratch training's data-hungry initialization, making it the most effective strategy for this task.

Evaluation(Based on the best model: Fine-tuned EfficientNet-B0)

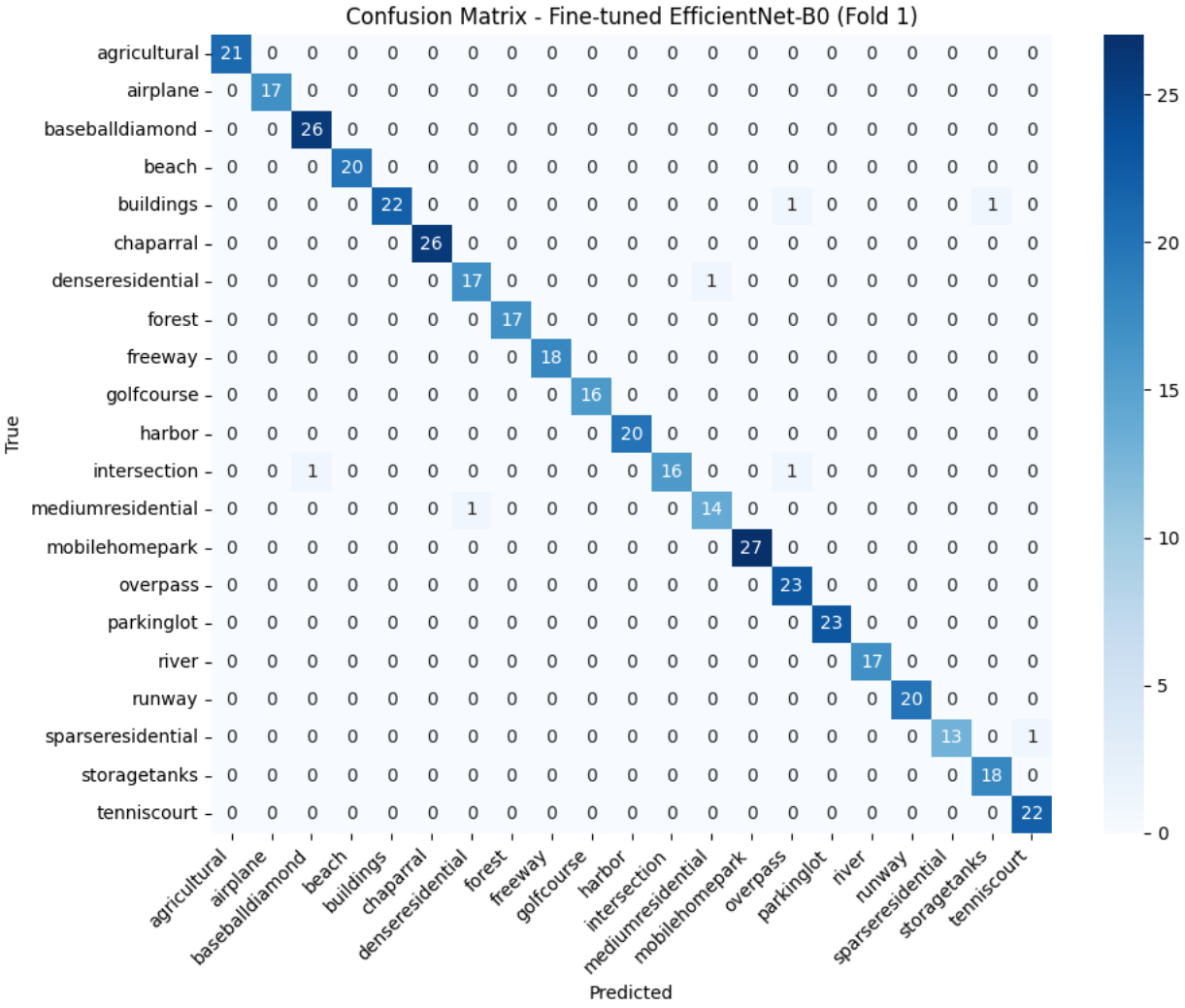
4.1 5 Different Metrics

Test Loss	**0.0781**
Test Accuracy	**98.33%**
Precision	**0.984**
Recall	**0.9833**
F1-Score	**0.9832**

- The fine-tuned EfficientNet-B0, determined as the best model, was evaluated on the test set, yielding exceptional performance across multiple metrics.
- The test loss was 0.0781, indicating minimal prediction error. Test accuracy reached 98.33%, reflecting near-perfect classification of the 21 land-use classes in the UCMerced dataset. Precision (0.9840) and recall (0.9833) demonstrate high reliability and completeness of predictions, respectively, while the F1-score (0.9832) confirms a balanced trade-off between the two.
- These results highlight the model's robustness, leveraging pre-trained features and fine-tuning to achieve superior generalization and accuracy on unseen data.

4.2 Confusion Matrix.

- The confusion matrix for the fine-tuned EfficientNet-B0 (Fold 1) reveals its exceptional performance, with high diagonal values (e.g., 21 for 'agricultural', 27 for 'mobilehomepark') reflecting accurate class-specific feature extraction, consistent with its 98.33% accuracy.
- Misclassifications, such as one 'intersection' as 'buildings' or one 'mediumresidential' as 'buildings', stem from shared low-level features (e.g., edges, textures) that pre-trained convolutional layers may overgeneralize, a common challenge in transfer learning.
- Similarly, one 'sparsresidential' misclassified as 'storagetanks' suggests difficulties in distinguishing sparse layouts from industrial patterns, highlighting potential limitations in high-level semantic differentiation.
- These errors, though rare, underscore the model's reliance on pre-trained feature hierarchies, which excel with abundant data but may require task-specific fine-tuning to fully resolve subtle class boundaries. Maybe enhancing the dataset with diverse examples or adjusting layer weights could mitigate these inherent biases.


Confusion Matrix - Fine-tuned EfficientNet-B0 (Fold 1)
True
Predicted
agricultural
airplane
baseballdiamond
beach
buildings
chaparral
denseresidential
forest
freeway
golfcourse
harbor
intersection
mediumresidential
mobilehomepark
overpass
parkinglot
river
runway
sparseresidential
storagetanks
tenniscourt
0
5
10
15
20
25