

SHORT REPORT

DEEP LEARNING



I. Introduction

Remote sensing image classification plays a crucial role in analyzing geographical features using aerial or satellite images. The task involves categorizing different regions or objects such as forests, beaches, or buildings based on visual data. In this project, we employ InceptionNet, a deep learning architecture known for its efficiency in image classification, to classify remote sensing images into distinct categories.

Our aim is to explore the effectiveness of **Inception v3**, a version of InceptionNet, in classifying a dataset of remote sensing images. The project is implemented and trained on Google Colab, using the UC Merced Land Use Dataset, which consists of 500 images across 5 different classes.

II. Topic Description

1. InceptionNet

InceptionNet is a convolutional neural network (CNN) architecture designed to improve both the accuracy and computational efficiency of deep learning models. Introduced in Google's Inception v1 model, it employs a combination of various filter sizes within each layer to capture different levels of feature details simultaneously. This method helps the network better understand complex features within an image.

The **Inception v3** model, used in this project, enhances the initial version by adding auxiliary classifiers, factorized convolutions, and batch normalization, reducing the computational cost and increasing performance for high-resolution image classification.

2. Remote Sensing and Inception v3 for Image Classification

Remote sensing involves capturing images of Earth's surface using aerial or satellite cameras. These images are often complex, making traditional classification models less effective. **Inception v3** can manage this complexity thanks to its multi-scale feature extraction capability. By using different filter sizes, the model can detect both small and large objects, which is highly useful for remote sensing images that have varied object sizes and textures.

In this project, Inception v3 is pretrained on the ImageNet dataset and fine-tuned for our specific dataset of remote sensing images.

III. User Requirements

1. Use of Google Colab

The decision to use Google Colab as our environment for training the model was driven by multiple factors. Most importantly, Colab provides **GPU access** for free, which significantly speeds up the training process compared to using a regular CPU. Moreover, it allows us to work in a **cloud environment**, ensuring that all group members can access the same resources and code without being hindered by the individual hardware configurations of

their personal devices. This collaborative, cloud-based setup ensures that everyone can work efficiently regardless of machine capabilities.

2. Dataset

We use the **UC Merced Land Use Dataset** for our classification task. This dataset contains **500 images**, divided into **5 classes**:

- Beach
- Airplane
- Forest
- Buildings
- Tennis court

The images are of size **256x256 pixels**, which are resized to **299x299 pixels** to match the input requirements of the **Inception v3** model. This resizing is performed during the data preprocessing stage.



Figure 1: Data Collection of Airplane class

IV. Operating Structure and Principle

1. Code Explanation

a. Library Imports

```
!pip install torchvision matplotlib
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision.datasets import ImageFolder
from torchvision import models
import os
import matplotlib.pyplot as plt
```

We install and import essential libraries such as **torchvision** (for working with datasets and models), **torch** (for building neural networks), and **matplotlib** (for visualizing results).

b. Data Preprocessing and Transformations

```
transform = transforms.Compose([
    transforms.Resize((299, 299)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])
```

`Resize((299, 299))`: Resizes the images to 299x299 pixels, the required input size for **Inception v3**.

`ToTensor()`: Converts the images to tensors, which are the basic data structure used by PyTorch.

`Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])`: Normalizes the image's pixel values using the mean and standard deviation of ImageNet. This step ensures that the input images match the distribution the pre-trained **Inception v3** model was trained on.

c. Dataset Splitting

```
train_data, test_data = train_test_split(dataset, test_size=0.15,
random_state=42)
train_data, val_data = train_test_split(train_data, test_size=0.1765,
random_state=42)
```

`train_test_split()`: A function from **sklearn.model_selection** used to split the dataset into training, validation, and test sets.

d. DataLoader Setup

```
train_loader = DataLoader(train_data, batch_size=32, shuffle=True)
val_loader = DataLoader(val_data, batch_size=32, shuffle=False)
test_loader = DataLoader(test_data, batch_size=32, shuffle=False)
```

`DataLoader()`: Efficiently loads data in batches during training, allowing the model to process the dataset in chunks rather than loading the entire dataset into memory.

- `batch_size=32`: The model processes 32 images at a time, which balances memory usage and speed.
- `shuffle=True`: For training data, this ensures the data is shuffled before each epoch to improve model generalization.
- `shuffle=False`: For validation and test sets, shuffling is unnecessary, so we keep the original order.

e. Model Setup and Modifications

```
model = models.inception_v3(pretrained=True)
model.fc = nn.Linear(model.fc.in_features, 5)
```

Inception v3 pretrained model: Loads a version of InceptionNet that has already been trained on the **ImageNet** dataset. This allows us to leverage knowledge the model has gained from general images.

`model.fc`: We modify the fully connected (fc) layer to output 5 classes instead of the 1000 classes of ImageNet.

-
- `nn.Linear(model.fc.in_features, 5)`: The output layer is changed to have 5 outputs, corresponding to the 5 classes in our dataset.

f. Freezing Pretrained Layers

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

Freezing layers: All layers except the final fully connected layer are "frozen," meaning their weights are not updated during training. This speeds up training and prevents the model from forgetting what it learned from ImageNet.

Unfreezing fully connected layer: Only the weights in the fully connected layer (which we modified) are trained to learn our specific task of remote sensing image classification.

g. Model Training Setup

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = model.to(device)
num_epochs = 10
train_loss_list = []
train_acc_list = []
val_loss_list = []
val_acc_list = []
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.fc.parameters(), lr=0.001)
```

num_epochs: The number of times the model will go through the entire training set.

10 epochs is chosen because it is generally sufficient for fine-tuning a pre-trained model like InceptionNet on a smaller dataset.

More epochs can lead to overfitting and excessive training time, while fewer epochs could result in underfitting, preventing the model from learning the data well.

`train_loss_list, train_acc_list, val_loss_list, val_acc_list`: Lists to store training and validation metrics (loss and accuracy) for each epoch, allowing us to plot and monitor the model's progress.

`criterion = nn.CrossEntropyLoss()`: The loss function used to measure how well the model's predictions match the true labels. Since this is a classification task, `CrossEntropyLoss` is appropriate.

`optimizer = optim.Adam()`: Optimizes the model's parameters using the **Adam** optimizer, which adapts the learning rate based on gradient history.

h. Training Loop

```
def train_model(model, train_loader, val_loader, criterion, optimizer,
num_epochs):
    for epoch in range(num_epochs):
        model.train() # Set the model to training mode
        running_loss = 0.0
        correct = 0
        total = 0
        for inputs, labels in train_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad() # Zero the gradients
            outputs = model(inputs) # Forward pass
            if isinstance(outputs, tuple): outputs = outputs[0] # Handle
Inception's auxiliary outputs
            loss = criterion(outputs, labels) # Compute the loss
            loss.backward() # Backpropagate the loss
            optimizer.step() # Update the model's weights
            _, predicted = torch.max(outputs, 1) # Get the class
predictions
            total += labels.size(0)
            correct += (predicted == labels).sum().item() # Count correct
predictions
            running_loss += loss.item() # Sum the loss

        # Save the training results
        train_loss_list.append(running_loss / len(train_loader))
        train_acc_list.append(100 * correct / total)
        print(f"Epoch [{epoch+1}/{num_epochs}], Loss:
{running_loss/len(train_loader)}, Accuracy: {100 * correct / total}%")
```

```

# Validation loop
model.eval() # Set the model to evaluation mode
val_running_loss = 0.0
val_correct = 0
val_total = 0
with torch.no_grad():
    for inputs, labels in val_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        _, predicted = torch.max(outputs, 1)
        val_total += labels.size(0)
        val_correct += (predicted == labels).sum().item()
        val_running_loss += loss.item()

    val_loss_list.append(val_running_loss / len(val_loader))
    val_acc_list.append(100 * val_correct / val_total)
    print(f"Validation Loss: {val_running_loss/len(val_loader)},
Validation Accuracy: {100 * val_correct / val_total}%")

train_model(model, train_loader, val_loader, criterion, optimizer,
num_epochs)

```

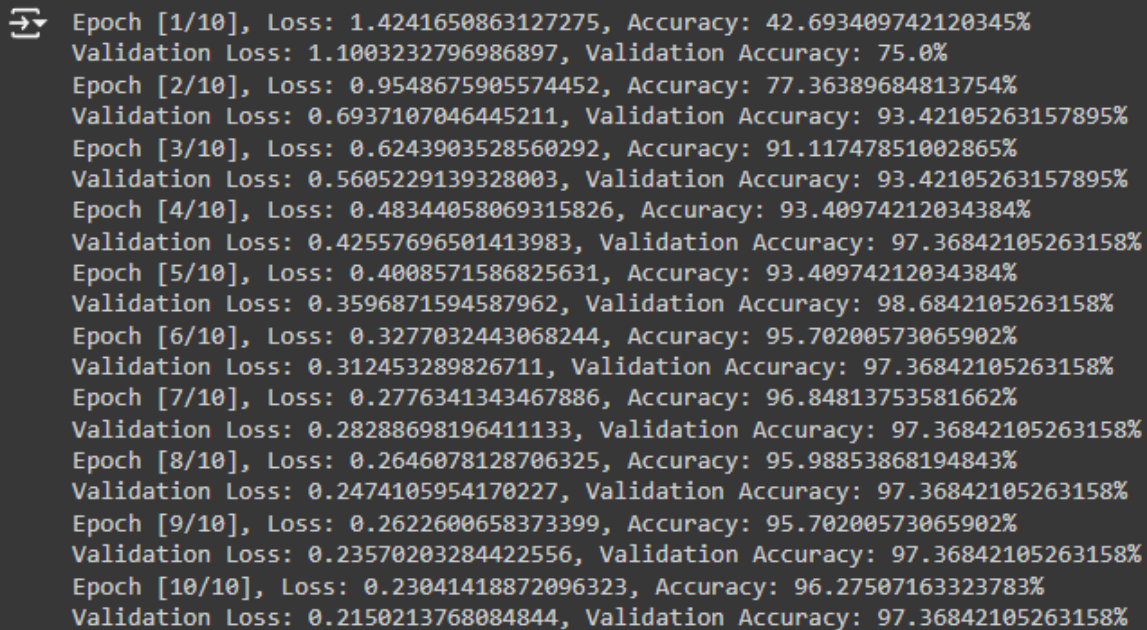
`model.train()`: Switches the model to training mode, allowing it to update weights and compute gradients.

`zero_grad()`: Clears gradients before each iteration, preventing gradient accumulation.

`isinstance(outputs, tuple)`: **Inception v3** has auxiliary outputs. This line ensures that only the primary output is used for loss calculation.

`optimizer.step()`: Updates the weights based on the gradients computed during backpropagation.

`model.eval()`: Puts the model in evaluation mode for validation, ensuring that no gradients are computed.



A terminal window showing the output of a training process over 10 epochs. Each epoch displays the training Loss and Accuracy, followed by the Validation Loss and Accuracy. The training Loss decreases from approximately 1.42 to 0.21, while the training Accuracy increases from 42.69% to 96.27%. The Validation Loss remains relatively stable around 0.21-0.25, and the Validation Accuracy stays around 97.36%.

```
Epoch [1/10], Loss: 1.4241650863127275, Accuracy: 42.693409742120345%  
Validation Loss: 1.1003232796986897, Validation Accuracy: 75.0%  
Epoch [2/10], Loss: 0.9548675905574452, Accuracy: 77.36389684813754%  
Validation Loss: 0.6937107046445211, Validation Accuracy: 93.42105263157895%  
Epoch [3/10], Loss: 0.6243903528560292, Accuracy: 91.11747851002865%  
Validation Loss: 0.5605229139328003, Validation Accuracy: 93.42105263157895%  
Epoch [4/10], Loss: 0.48344058069315826, Accuracy: 93.40974212034384%  
Validation Loss: 0.42557696501413983, Validation Accuracy: 97.36842105263158%  
Epoch [5/10], Loss: 0.4008571586825631, Accuracy: 93.40974212034384%  
Validation Loss: 0.3596871594587962, Validation Accuracy: 98.6842105263158%  
Epoch [6/10], Loss: 0.3277032443068244, Accuracy: 95.70200573065902%  
Validation Loss: 0.312453289826711, Validation Accuracy: 97.36842105263158%  
Epoch [7/10], Loss: 0.2776341343467886, Accuracy: 96.84813753581662%  
Validation Loss: 0.28288698196411133, Validation Accuracy: 97.36842105263158%  
Epoch [8/10], Loss: 0.2646078128706325, Accuracy: 95.98853868194843%  
Validation Loss: 0.2474105954170227, Validation Accuracy: 97.36842105263158%  
Epoch [9/10], Loss: 0.2622600658373399, Accuracy: 95.70200573065902%  
Validation Loss: 0.23570203284422556, Validation Accuracy: 97.36842105263158%  
Epoch [10/10], Loss: 0.23041418872096323, Accuracy: 96.27507163323783%  
Validation Loss: 0.2150213768084844, Validation Accuracy: 97.36842105263158%
```

Figure 2: Loss and Accuracy each epoch

i. Evaluation

```
def evaluate_model(model, test_loader):  
    model.eval()  
    test_correct = 0  
    test_total = 0  
    with torch.no_grad():  
        for inputs, labels in test_loader:  
            inputs, labels = inputs.to(device), labels.to(device)  
            outputs = model(inputs)  
            _, predicted = torch.max(outputs, 1)  
            test_total += labels.size(0)  
            test_correct += (predicted == labels).sum().item()  
    print(f"Test Accuracy: {100 * test_correct / test_total}%")
```

`evaluate_model()`: Tests the trained model on the test set. Since we're evaluating, we use `torch.no_grad()` to avoid calculating gradients (which saves memory and computation time).

```
➡ Accuracy of the model on the test images: 98.6666666666667%
```

Figure 3: Accuracy of the model

j. Plotting Loss and Accuracy

```
def plot_metrics(train_loss, val_loss, train_acc, val_acc):  
    plt.figure(figsize=(12, 5))  
    plt.subplot(1, 2, 1)  
    plt.plot(train_loss, label="Training Loss")  
    plt.plot(val_loss, label="Validation Loss")  
    plt.title("Loss")  
    plt.legend()  
    plt.subplot(1, 2, 2)  
    plt.plot(train_acc, label="Training Accuracy")  
    plt.plot(val_acc, label="Validation Accuracy")  
    plt.title("Accuracy")  
    plt.legend()  
    plt.show()
```

`plot_metrics()`: Plots the loss and accuracy over epochs for both training and validation. This visualization helps to monitor whether the model is overfitting or underfitting.

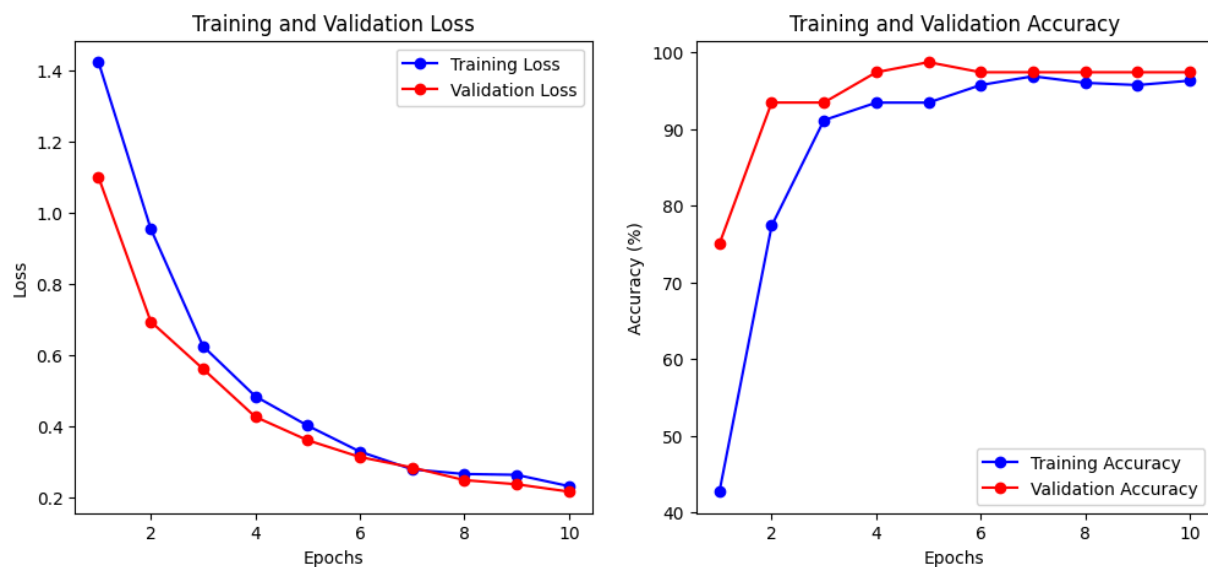


Figure 4: Plots of Loss and Accuracy

IV. Conclusion

1. Scope Within Terms of This Project

This project demonstrated the effectiveness of the **Inception v3** model in classifying remote sensing images into five categories. The model, pretrained on ImageNet, achieved competitive accuracy after fine-tuning on our dataset. By leveraging **Google Colab**, we could train the model efficiently using GPU resources.

2. Future Development and Advanced Applications

For future improvements, we could:

- **Fine-tune additional layers** to capture more domain-specific features from the remote sensing data.
- **Increase dataset size** by incorporating data augmentation techniques to improve the model's robustness.
- Explore the application of **other architectures**, such as ResNet or EfficientNet, to compare performance.

References:

1. UC Merced Land Use Dataset

<http://weegee.vision.ucmerced.edu/datasets/landuse.html>

2. Inception Explained

<https://www.youtube.com/watch?v=STTrebkhnIk>

3. Inception v3

<https://paperswithcode.com/method/inception-v3>

4. How to use Google Colab

<https://www.youtube.com/watch?v=RLYoEyIHL6A>