# **Poultry Disease Classification using ResNet-18 and PyTorch**



## **Overview**

This notebook implements a poultry disease classification system using the ResNet-18 architecture and PyTorch. The goal is to accurately classify poultry diseases based on a curated dataset of fecal images, aiding in early disease detection and management for poultry farmers.

### **Dataset Description**

The Poultry Pathology Visual Dataset contains a diverse collection of poultry fecal images categorized into four classes:

* Coccidiosis
* Healthy
* Newcastle Disease
* Salmonella

### **Model Architecture**

The ResNet-18 model, a deep convolutional neural network, is utilized for this classification task. This architecture is well-suited for image classification tasks, offering a balance between accuracy and computational efficiency.

### **Key Steps**

1. **Data Loading and Preprocessing:** The dataset is loaded, preprocessed
2. **Model Definition:** The ResNet-18 architecture is defined and configured for poultry disease classification.
3. **Model Training:** The model is trained on the training set..
4. **Model Evaluation:** The trained model's performance is assessed on test set.

### **Why PyTorch?**

PyTorch, a popular deep learning framework, provides a flexible and efficient platform for building and training neural networks. Its dynamic computation graph and ease of use make it a preferred choice for many machine learning practitioners.

This notebook serves as a practical example of leveraging PyTorch and powerful neural network architectures like ResNet-18 for real-world applications in poultry disease detection and management.

In [1]:

import os

from random import shuffle

import pandas as pd

import numpy as np

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, random\_split

from torch.optim import lr\_scheduler

from torchvision.models import resnet18, ResNet18\_Weights

import torchvision.datasets as datasets

import time

import csv

import matplotlib.pyplot as plt

from sklearn.metrics import confusion\_matrix, classification\_report

import seaborn as sns

/opt/conda/lib/python3.10/site-packages/scipy/\_\_init\_\_.py:146: UserWarning: A NumPy version >=1.16.5 and <1.23.0 is required for this version of SciPy (detected version 1.24.3

warnings.warn(f"A NumPy version >={np\_minversion} and <{np\_maxversion}"

In [2]:

data\_transforms = transforms.Compose([

transforms.Resize((224, 224)),

transforms.ToTensor(),

transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

])

### **Data Loading and Preparation**

The following code snippet demonstrates the data loading and preparation process for the poultry disease classification task using PyTorch DataLoader and ImageFolder from the datasets module.

#### **Parameters:**

* batch\_size: Set to 256 for batch processing during training and evaluation.
* num\_workers: Configured with 6 workers to parallelize data loading for efficiency.
* pin\_memory: Enabled to facilitate faster data transfer if using a GPU.

This setup ensures the efficient handling of data, enabling the training, validation, and evaluation of the poultry disease classification model using the specified datasets.

In [3]:

batch\_size = 256

num\_workers = 6

pin\_memory = True *# Enable for faster data transfer if using GPU*

dataset = datasets.ImageFolder(root='/kaggle/input/poultry-diseases/data/data/train', transform=data\_transforms)

train\_loader = DataLoader(dataset, batch\_size=batch\_size, shuffle=True, num\_workers=num\_workers, pin\_memory=pin\_memory)

dataset\_val = datasets.ImageFolder(root='/kaggle/input/poultry-diseases/data/data/val', transform=data\_transforms)

val\_loader = DataLoader(dataset\_val, batch\_size=batch\_size, shuffle=False, num\_workers=num\_workers, pin\_memory=pin\_memory)

dataset\_test = datasets.ImageFolder(root='/kaggle/input/poultry-diseases/data/data/test', transform=data\_transforms)

test\_loader = DataLoader(dataset\_test, batch\_size=batch\_size, shuffle=False, num\_workers=num\_workers, pin\_memory=pin\_memory)

### **Viewing Samples from the Test Dataset**

To visualize and inspect the images along with their corresponding labels from the train dataset, the following Python code utilizes Matplotlib.

The iteration through the train\_loader retrieves a batch of images (images) and their associated labels (labels). The function imshow() is defined to display images with their corresponding labels.

In [4]:

class\_names = ["Coccidiosis", "Healthy", "New Castle Disease", "Salmonella"]

*# Assuming 'images' contains the image data and 'labels' contains the corresponding labels*

for images, labels **in** train\_loader:

*# Function to display images with labels*

def imshow(img, title):

img = np.transpose(img, (1, 2, 0))

plt.imshow(img)

plt.title(title)

plt.axis('off')

*# Displaying 16 images in a 4x4 grid*

plt.figure(figsize=(10, 10))

for idx **in** range(16):

plt.subplot(4, 4, idx + 1)

*# Get the class name corresponding to the label*

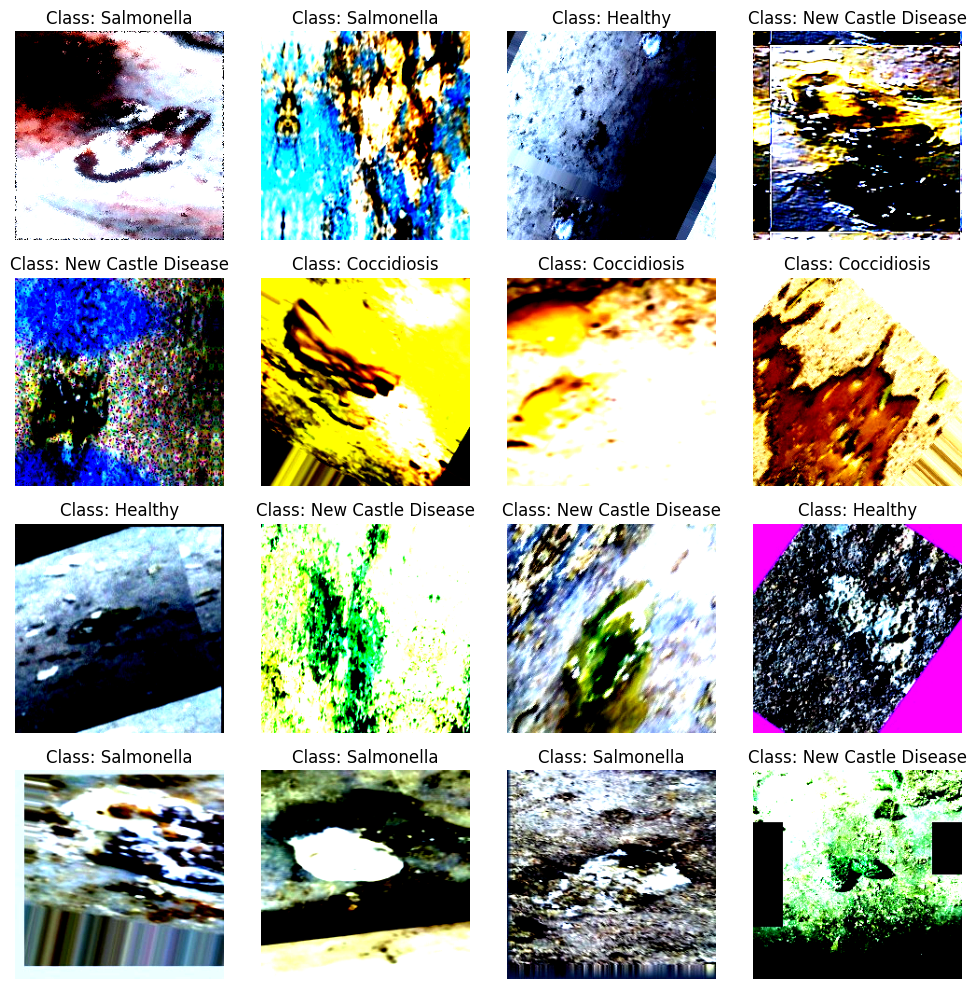
class\_label = class\_names[labels[idx]]

imshow(images[idx], f"Class: **{**class\_label**}**") *# Display class name as title*

plt.tight\_layout()

plt.show()

break *# Only display one batch of images for brevity*



In [5]:

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

*# Check for the availability of GPUs*

if torch.cuda.is\_available():

num\_devices = torch.cuda.device\_count()

for i **in** range(num\_devices):

print(f"GPU **{**i**}**: **{**torch.cuda.get\_device\_name(i)**}**")

else:

print("No GPU available, training on CPU")

GPU 0: Tesla P100-PCIE-16GB

### **ResNet-18 Model Configuration**

This snippet initializes a ResNet-18 model for poultry disease classification. Key steps include:

* Using resnet18 with default weights to create the model.
* Allowing parameter fine-tuning during training.
* Modifying the output layer for 4 classes.
* Configuring multi-GPU training for enhanced performance.
* Moving the model to the specified device (e.g., GPU) for computation.

This setup readies the ResNet-18 architecture for efficient training and classification of poultry diseases.

In [6]:

if **not** os.path.exists("resnet18"):

os.mkdir("resnet18")

*# Define ResNet-18 model*

model = resnet18(weights=ResNet18\_Weights.DEFAULT)

for param **in** model.parameters():

param.requires\_grad = True

num\_classes = 4

model.fc = nn.Linear(model.fc.in\_features, num\_classes)

model.to(device)

Downloading: "https://download.pytorch.org/models/resnet18-f37072fd.pth" to /root/.cache/torch/hub/checkpoints/resnet18-f37072fd.pth

100%|██████████| 44.7M/44.7M [00:00<00:00, 187MB/s]

Out[6]:

ResNet(

(conv1): Conv2d(3, 64, kernel\_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)

(bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

(relu): ReLU(inplace=True)

(maxpool): MaxPool2d(kernel\_size=3, stride=2, padding=1, dilation=1, ceil\_mode=False)

(layer1): Sequential(

(0): BasicBlock(

(conv1): Conv2d(64, 64, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)

(bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

(relu): ReLU(inplace=True)

(conv2): Conv2d(64, 64, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)

(bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

)

(1): BasicBlock(

(conv1): Conv2d(64, 64, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)

(bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

(relu): ReLU(inplace=True)

(conv2): Conv2d(64, 64, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)

(bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

)

)

(layer2): Sequential(

(0): BasicBlock(

(conv1): Conv2d(64, 128, kernel\_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)

(bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

(relu): ReLU(inplace=True)

(conv2): Conv2d(128, 128, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)

(bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

(downsample): Sequential(

(0): Conv2d(64, 128, kernel\_size=(1, 1), stride=(2, 2), bias=False)

(1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

)

)

(1): BasicBlock(

(conv1): Conv2d(128, 128, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)

(bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

(relu): ReLU(inplace=True)

(conv2): Conv2d(128, 128, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)

(bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

)

)

(layer3): Sequential(

(0): BasicBlock(

(conv1): Conv2d(128, 256, kernel\_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)

(bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

(relu): ReLU(inplace=True)

(conv2): Conv2d(256, 256, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)

(bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

(downsample): Sequential(

(0): Conv2d(128, 256, kernel\_size=(1, 1), stride=(2, 2), bias=False)

(1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

)

)

(1): BasicBlock(

(conv1): Conv2d(256, 256, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)

(bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

(relu): ReLU(inplace=True)

(conv2): Conv2d(256, 256, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)

(bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

)

)

(layer4): Sequential(

(0): BasicBlock(

(conv1): Conv2d(256, 512, kernel\_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)

(bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

(relu): ReLU(inplace=True)

(conv2): Conv2d(512, 512, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)

(bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

(downsample): Sequential(

(0): Conv2d(256, 512, kernel\_size=(1, 1), stride=(2, 2), bias=False)

(1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

)

)

(1): BasicBlock(

(conv1): Conv2d(512, 512, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)

(bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

(relu): ReLU(inplace=True)

(conv2): Conv2d(512, 512, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)

(bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

)

)

(avgpool): AdaptiveAvgPool2d(output\_size=(1, 1))

(fc): Linear(in\_features=512, out\_features=4, bias=True)

)

### **Training Setup: Loss, Optimizer, and Scheduler**

This code block defines:

* **Loss Function:** Cross-Entropy Loss suitable for multi-class classification tasks.
* **Optimizer:** AdamW optimizer with a learning rate of 0.001 and weight decay for model updates.
* **Scheduler:** Reduces the learning rate on plateaus to optimize training efficiency.

These settings configure the training process by specifying the loss calculation, optimizer for updating model weights, and a scheduler to dynamically adjust the learning rate.

In [8]:

*# Define loss function and optimizer*

criterion = nn.CrossEntropyLoss()

optimizer = optim.AdamW(model.parameters(), lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight\_decay=0.01)

scheduler = lr\_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=3, verbose=True, factor=0.2, min\_lr=1e-5)

### **Training Loop with Validation and Logging**

In [9]:

linkcode

num\_epochs = 30

best\_val\_accuracy = 0.0

csv\_filename = "resnet18/training\_log.csv"

with open(csv\_filename, mode='w', newline='') as file:

writer = csv.writer(file)

writer.writerow(["Epoch", "Train Loss", "Val Loss", "Train Accuracy", "Val Accuracy", "Time (s)"])

*# Initialize variables for best model tracking*

best\_val\_loss = float('inf')

for epoch **in** range(num\_epochs):

model.train()

running\_loss = 0.0

correct\_train = 0

total\_train = 0

start\_time = time.time()

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()

running\_loss += loss.item()

\_, predicted = torch.max(outputs.data, 1)

total\_train += labels.size(0)

correct\_train += (predicted == labels).sum().item()

*# Calculate training accuracy for this epoch*

train\_accuracy = 100 \* correct\_train / total\_train

*# Validation*

model.eval()

correct\_val = 0

total\_val = 0

val\_loss = 0.0

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)

val\_loss += loss.item()

\_, predicted = torch.max(outputs.data, 1)

total\_val += labels.size(0)

correct\_val += (predicted == labels).sum().item()

*# Calculate validation accuracy for this epoch*

val\_accuracy = 100 \* correct\_val / total\_val

scheduler.step(val\_loss)

end\_time = time.time()

epoch\_time = end\_time - start\_time

*# Print and log epoch results*

print(f"Epoch **{**epoch+1**}**/**{**num\_epochs**}**, Training Loss: **{**running\_loss/len(train\_loader)**:**.4f**}**, "

f"Val Loss: **{**val\_loss/len(val\_loader)**:**.4f**}**, Train Accuracy: **{**train\_accuracy**:**.2f**}**%, "

f"Val Accuracy: **{**val\_accuracy**:**.2f**}**%, Time: **{**epoch\_time**:**.2f**}** seconds")

*# Log results to CSV file*

with open(csv\_filename, mode='a', newline='') as file:

writer = csv.writer(file)

writer.writerow([epoch+1, running\_loss/len(train\_loader), val\_loss/len(val\_loader), train\_accuracy, val\_accuracy, epoch\_time])

*# Check if this is the best model so far based on validation loss*

if val\_loss < best\_val\_loss:

best\_val\_loss = val\_loss

best\_model\_epoch = epoch + 1

*# Save the best model checkpoint*

best\_model\_path = f"resnet18/best.pt"

torch.save({

'epoch': epoch + 1,

'model\_state\_dict': model.state\_dict(),

'optimizer\_state\_dict': optimizer.state\_dict(),

'loss': val\_loss,

}, best\_model\_path)

torch.save(model.state\_dict(), "resnet18/last.pt")

print("Training complete")

Epoch 1/30, Training Loss: 0.2573, Val Loss: 0.2193, Train Accuracy: 90.22%, Val Accuracy: 91.82%, Time: 781.15 seconds

Epoch 2/30, Training Loss: 0.1683, Val Loss: 0.1699, Train Accuracy: 93.72%, Val Accuracy: 93.80%, Time: 666.69 seconds

Epoch 3/30, Training Loss: 0.1346, Val Loss: 0.1612, Train Accuracy: 94.94%, Val Accuracy: 94.05%, Time: 664.79 seconds

Epoch 4/30, Training Loss: 0.1101, Val Loss: 0.1549, Train Accuracy: 95.84%, Val Accuracy: 94.43%, Time: 666.15 seconds

Epoch 5/30, Training Loss: 0.0899, Val Loss: 0.1570, Train Accuracy: 96.59%, Val Accuracy: 94.42%, Time: 667.25 seconds

Epoch 6/30, Training Loss: 0.0743, Val Loss: 0.1613, Train Accuracy: 97.21%, Val Accuracy: 94.66%, Time: 664.69 seconds

Epoch 7/30, Training Loss: 0.0624, Val Loss: 0.1626, Train Accuracy: 97.67%, Val Accuracy: 94.62%, Time: 669.41 seconds

Epoch 00008: reducing learning rate of group 0 to 2.0000e-04.

Epoch 8/30, Training Loss: 0.0525, Val Loss: 0.1622, Train Accuracy: 98.05%, Val Accuracy: 94.83%, Time: 664.81 seconds

Epoch 9/30, Training Loss: 0.0148, Val Loss: 0.1727, Train Accuracy: 99.47%, Val Accuracy: 95.86%, Time: 666.98 seconds

Epoch 10/30, Training Loss: 0.0049, Val Loss: 0.2097, Train Accuracy: 99.84%, Val Accuracy: 95.79%, Time: 665.57 seconds

Epoch 11/30, Training Loss: 0.0040, Val Loss: 0.2411, Train Accuracy: 99.86%, Val Accuracy: 95.62%, Time: 667.72 seconds

Epoch 00012: reducing learning rate of group 0 to 4.0000e-05.

Epoch 12/30, Training Loss: 0.0037, Val Loss: 0.2588, Train Accuracy: 99.87%, Val Accuracy: 95.79%, Time: 665.34 seconds

Epoch 13/30, Training Loss: 0.0013, Val Loss: 0.2576, Train Accuracy: 99.96%, Val Accuracy: 95.89%, Time: 667.41 seconds

Epoch 14/30, Training Loss: 0.0007, Val Loss: 0.2665, Train Accuracy: 99.97%, Val Accuracy: 95.92%, Time: 665.68 seconds

Epoch 15/30, Training Loss: 0.0006, Val Loss: 0.2800, Train Accuracy: 99.98%, Val Accuracy: 95.88%, Time: 669.89 seconds

Epoch 00016: reducing learning rate of group 0 to 1.0000e-05.

Epoch 16/30, Training Loss: 0.0005, Val Loss: 0.2862, Train Accuracy: 99.98%, Val Accuracy: 95.93%, Time: 668.75 seconds

Epoch 17/30, Training Loss: 0.0004, Val Loss: 0.2902, Train Accuracy: 99.98%, Val Accuracy: 96.00%, Time: 671.54 seconds

Epoch 18/30, Training Loss: 0.0004, Val Loss: 0.2919, Train Accuracy: 99.99%, Val Accuracy: 95.92%, Time: 673.00 seconds

Epoch 19/30, Training Loss: 0.0003, Val Loss: 0.2968, Train Accuracy: 99.99%, Val Accuracy: 95.97%, Time: 672.41 seconds

Epoch 20/30, Training Loss: 0.0003, Val Loss: 0.3026, Train Accuracy: 99.99%, Val Accuracy: 96.00%, Time: 669.69 seconds

Epoch 21/30, Training Loss: 0.0003, Val Loss: 0.3037, Train Accuracy: 99.98%, Val Accuracy: 95.94%, Time: 671.16 seconds

Epoch 22/30, Training Loss: 0.0003, Val Loss: 0.3076, Train Accuracy: 99.99%, Val Accuracy: 96.00%, Time: 669.31 seconds

Epoch 23/30, Training Loss: 0.0003, Val Loss: 0.3090, Train Accuracy: 99.98%, Val Accuracy: 95.92%, Time: 672.75 seconds

Epoch 24/30, Training Loss: 0.0003, Val Loss: 0.3118, Train Accuracy: 99.99%, Val Accuracy: 95.95%, Time: 670.08 seconds

Epoch 25/30, Training Loss: 0.0003, Val Loss: 0.3123, Train Accuracy: 99.98%, Val Accuracy: 95.94%, Time: 666.56 seconds

Epoch 26/30, Training Loss: 0.0003, Val Loss: 0.3131, Train Accuracy: 99.99%, Val Accuracy: 95.96%, Time: 670.84 seconds

Epoch 27/30, Training Loss: 0.0003, Val Loss: 0.3208, Train Accuracy: 99.99%, Val Accuracy: 95.93%, Time: 673.24 seconds

Epoch 28/30, Training Loss: 0.0003, Val Loss: 0.3193, Train Accuracy: 99.99%, Val Accuracy: 95.90%, Time: 675.13 seconds

Epoch 29/30, Training Loss: 0.0002, Val Loss: 0.3217, Train Accuracy: 99.99%, Val Accuracy: 95.95%, Time: 674.42 seconds

Epoch 30/30, Training Loss: 0.0002, Val Loss: 0.3222, Train Accuracy: 99.99%, Val Accuracy: 95.95%, Time: 669.91 seconds

Training complete

### **Visualizing Model Training Metrics**

This code snippet processes a CSV file storing model training metrics such as epoch, train loss, validation loss, train accuracy, validation accuracy, and time taken per epoch.

The Python code reads the CSV using Pandas, organizing the data into a DataFrame. It then employs Matplotlib to create two informative plots:

* **Train and Validation Loss**: Depicts how the training and validation losses evolve with each epoch, providing insights into the model's learning progress and potential overfitting.
* **Train and Validation Accuracy**: Illustrates the changes in training and validation accuracies across epochs, offering an understanding of the model's learning performance.

In [10]:

linkcode

data = pd.read\_csv(csv\_filename)

*# Extract data columns*

epochs = data['Epoch']

train\_loss = data['Train Loss']

val\_loss = data['Val Loss']

train\_accuracy = data['Train Accuracy']

val\_accuracy = data['Val Accuracy']

time\_seconds = data['Time (s)']

*# Plotting*

plt.figure(figsize=(10, 6))

*# Train and Validation Loss Plot*

plt.subplot(2, 1, 1)

plt.plot(epochs, train\_loss, label='Train Loss')

plt.plot(epochs, val\_loss, label='Val Loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.title('Train and Validation Loss')

plt.legend()

*# Train and Validation Accuracy Plot*

plt.subplot(2, 1, 2)

plt.plot(epochs, train\_accuracy, label='Train Accuracy')

plt.plot(epochs, val\_accuracy, label='Val Accuracy')

plt.xlabel('Epochs')

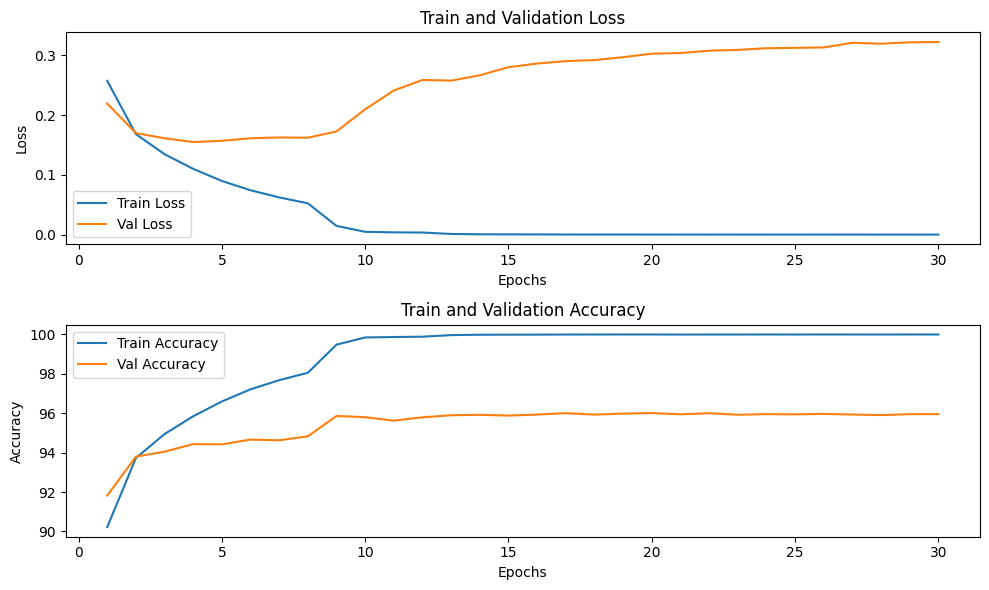
plt.ylabel('Accuracy')

plt.title('Train and Validation Accuracy')

plt.legend()

plt.tight\_layout()

plt.show()



Model Evaluation on Test Data¶

In this section, the trained ResNet-18 model is evaluated using the test dataset to assess its performance on unseen data.

In(11)

# Testing

Model.eval()

Class\_labels = dataset.classes

Correct\_test = 0

Total\_test = 0

Test\_loss = 0.0

True\_labels = []

Predicted\_labels = []

With torch.no\_grad():

For inputs, labels in test\_loader:

Inputs, labels = inputs.to(device), labels.to(device)

Outputs = model(inputs)

Loss = criterion(outputs, labels)

Test\_loss += loss.item()

\_, predicted = torch.max(outputs.data, 1)

Total\_test += labels.size(0)

Correct\_test += (predicted == labels).sum().item()

True\_labels.extend(labels.cpu().numpy())

Predicted\_labels.extend(predicted.cpu().numpy())

Test\_accuracy = 100 \* correct\_test / total\_test

Print(f”Test Loss: {test\_loss/len(test\_loader):.4f}, Test Accuracy: {test\_accuracy:.2f}%”}

Output:

Test Loss: 0.3087, Test Accuracy: 96.03%

In(12)

# Compute and plot the confusion matrix for the final test set

Conf\_matrix = confusion\_matrix(true\_labels, predicted\_labels)

Plt.figure(figsize=(len(class\_labels), len(class\_labels)))

Sns.heatmap(conf\_matrix, annot=True, fmt=”d”, cmap=”Blues”, xticklabels=class\_labels, yticklabels=class\_labels)

Plt.xlabel(“Predicted Labels”)

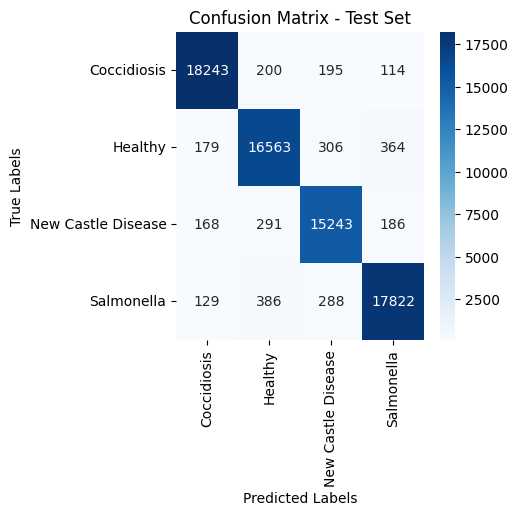
Plt.ylabel(“True Labels”)

Plt.title(“Confusion Matrix – Test Set”)

Plt.savefig(“resnet18/confusion\_matrix.png”, dpi=300, bbox\_inches=”tight”)

Plt.show()

Print(“Confusion matrix saved as confusion\_matrix.png”)



Confusion matrix saved as confusion\_matrix.png

In(13)

Print(classification\_report(true\_labels, predicted\_labels, target\_names=class\_labels))

Output

Precision recall f1-score support

Coccidiosis 0.97 0.97 0.97 18752

Healthy 0.95 0.95 0.95 17412

New Castle Disease 0.95 0.96 0.96 15888

Salmonella 0.96 0.96 0.96 18625

Accuracy 0.96 70677

Macro avg 0.96 0.96 0.96 70677

Weighted avg 0.96 0.96 0.96 70677

Conclusion¶

This implementation demonstrates a sample approach to poultry disease classification using the ResNet-18 model. Achieving a peak test accuracy of 96.03% on the test dataset, this model showcases robust performance in accurately identifying poultry diseases based on fecal images.

This methodology serves as a sample framework that can be similarly employed with other powerful models such as DenseNet, VGG, or custom architectures. By exploring alternative architectures and fine-tuning hyperparameters, further advancements and improvements in disease classification accuracy can be pursued within the domain of poultry pathology.