Butterfly Dataset

By Hatem Elgenedy

November 28, 2025

BUTTERFLY IMAGE CLASSIFICATION USING PYTORCH

[135]:
```
#  IMPORT LIBRARIES
# os: for file path handling
# numpy: numerical operations (arrays, etc.)
# matplotlib: for plotting
# pandas: reading and working with CSV files (train/test metadata)
# PIL.Image: to open and handle image files
import os
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from PIL import Image
# PyTorch core imports:
# Dataset, DataLoader: building input pipelines
# transforms: image preprocessing (resize, normalize, etc.)
# models: pretrained architectures like ResNet
# nn: neural network layers and loss functions
# optim: optimization algorithms such as Adam

import torch
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms, models
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from tqdm.auto import tqdm
```

[136]:
```
# DEFINE BASE PATHS
# BASE_DIR: the root folder of the butterfly dataset on disk.
# TRAIN_DIR / TEST_DIR: subfolders for train and test images.
# Note: "Train " and "Test " actually contain trailing spaces in their names,
# so we must use them exactly as they appear in the filesystem.

BASE_DIR = "/Users/hatemelgenedy/Desktop/AI and Data Science Microsoft course/
 ↪Butterfly dataset"
```

```
# TRAIN_IMAGES_DIR: the folder that stores the actual training images.
# TEST_IMAGES_DIR: the folder that stores the actual test images.
# Note the leading space in " butterfly_train" because the folder name has it.
TRAIN_DIR = os.path.join(BASE_DIR, "Train ")
TEST_DIR  = os.path.join(BASE_DIR, "Test ")

#Sanity check: print whether these image directories actually exist on disk.
TRAIN_IMAGES_DIR = os.path.join(TRAIN_DIR, " butterfly_train")   # note leading␣
 ↪space
TEST_IMAGES_DIR  = os.path.join(TEST_DIR, "butterfly_test")

print("Train images dir exists? ", os.path.exists(TRAIN_IMAGES_DIR))
print("Test images dir exists?  ", os.path.exists(TEST_IMAGES_DIR))
```

```
Train images dir exists?  True
Test images dir exists?   True
```

[137]:
```
# LOAD CSV FILES
# The dataset uses CSV files to map image filenames to labels.
# Training_set.csv: has 'filename' + 'label' columns.
# Testing_set.csv: has only 'filename' (no labels).
TRAIN_CSV_PATH = os.path.join(BASE_DIR, "Training_set.csv")
TEST_CSV_PATH  = os.path.join(BASE_DIR, "Testing_set.csv")
```

[138]:
```
# Read CSVs into pandas DataFrames for easy manipulation.

train_df = pd.read_csv(TRAIN_CSV_PATH)
test_df  = pd.read_csv(TEST_CSV_PATH)
```

[155]:
```
train_df.describe()
```

[155]:
```
          label_idx
count  6499.000000
mean     36.771042
std      21.457386
min       0.000000
25%      18.000000
50%      37.000000
75%      55.000000
max      74.000000
```

[156]:
```
test_df.describe()
```

[156]:
```
       pred_label_idx
count     2786.000000
mean        36.854630
std         21.429339
min          0.000000
```

```
25%         18.000000
50%         37.000000
75%         56.000000
max         74.000000
```

[139]:
```python
# Quick look at the first few rows and shape of each dataframe.

print(train_df.head())
print(test_df.head())
print("Train shape:", train_df.shape)
print("Test shape:", test_df.shape)
```

```
      filename                    label
0  Image_1.jpg          SOUTHERN DOGFACE
1  Image_2.jpg                   ADONIS
2  Image_3.jpg            BROWN SIPROETA
3  Image_4.jpg                  MONARCH
4  Image_5.jpg  GREEN CELLED CATTLEHEART
      filename
0  Image_1.jpg
1  Image_2.jpg
2  Image_3.jpg
3  Image_4.jpg
4  Image_5.jpg
Train shape: (6499, 2)
Test shape: (2786, 1)
```

[ ]:
```python
# ENCODE LABELS (STRING → INTEGER)
# Neural networks work with numbers, not strings.
# We convert butterfly species names (e.g., "MONARCH") to integer indices.

# Get sorted list of unique class names from the 'label' column.

class_names = sorted(train_df["label"].unique())

# Count how many different butterfly species we have.

num_classes = len(class_names)
print("Num classes:", num_classes)

# Create mapping from label string to integer index.
# Example: "ADONIS" → 0, "BROWN SIPROETA" → 1, etc.
label_to_index = {name: i for i, name in enumerate(class_names)}

# Create reverse mapping from integer index to label string.
# This is useful when we want to decode predictions back to readable names.
index_to_label = {i: name for name, i in label_to_index.items()}  # helper for␣
 ↪decoding
```

```
# Add a new column 'label_idx' to train_df that stores the numeric label.
# .map() applies our dictionary to the 'label' column.
train_df["label_idx"] = train_df["label"].map(label_to_index)

# Peek again to confirm we now have 'label_idx'.
train_df.head()
```

```
Num classes: 75
```

[ ]:

	filename	label	label_idx
0	Image_1.jpg	SOUTHERN DOGFACE	66
1	Image_2.jpg	ADONIS	0
2	Image_3.jpg	BROWN SIPROETA	12
3	Image_4.jpg	MONARCH	44
4	Image_5.jpg	GREEN CELLED CATTLEHEART	33

[ ]:
```
# DEFINE IMAGE TRANSFORMS
# IMG_SIZE: target size (height and width) for all images.
# BATCH_SIZE: how many images to process in one training step.

IMG_SIZE = 224
BATCH_SIZE = 32

# train_transform:
# - Resize: ensure all images are 224x224, as expected by ResNet.
# - RandomHorizontalFlip: simple data augmentation to help generalization.
# - ToTensor: convert from PIL image (H x W x C) to PyTorch tensor (C x H x W).
# - Normalize: standard ImageNet normalization since we're using a pretrained␣
 ↪model.

train_transform = transforms.Compose([
    transforms.Resize((IMG_SIZE, IMG_SIZE)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],  #These values represent␣
 ↪the average pixel intensity (mean) and standard deviation (std) for each RGB␣
 ↪channel in the ImageNet training dataset, computed over 1.2 million images.
                         std=[0.229, 0.224, 0.225]),
])

# test_transform:
# Similar to train, but without random augmentation (we want deterministic␣
 ↪behavior on test data).
test_transform = transforms.Compose([
    transforms.Resize((IMG_SIZE, IMG_SIZE)),
    transforms.ToTensor(),
```

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

[142]:
```
# CUSTOM DATASET CLASSES
# We define our own Dataset classes because our labels come from CSV files,
# not from folder names as in torchvision.datasets.ImageFolder.

class ButterflyTrainDataset(Dataset):
    # __init__ is called once when creating the dataset object.
    # df: DataFrame containing 'filename' and 'label_idx'
    # images_dir: folder where the actual image files live
    # transform: transforms (resize, normalize, etc.)
    def __init__(self, df, images_dir, transform=None):
        # We reset the index so that __getitem__ can use a clean 0..N-1 index.
        # drop=True: we don't keep the old index column, we just reindex.
        self.df = df.reset_index(drop=True)
        self.images_dir = images_dir
        self.transform = transform

 # __len__ tells PyTorch how many samples are in the dataset.
    def __len__(self):
        return len(self.df)

# __getitem__ loads and returns a single sample at the given index.
    def __getitem__(self, idx):
        # Get the row corresponding to this index.
        row = self.df.iloc[idx]
        filename = row["filename"]
        label_idx = int(row["label_idx"])

# Build the full path to the image file.
        img_path = os.path.join(self.images_dir, filename)
        # Open the image with PIL and convert to RGB to ensure 3 channels.
        image = Image.open(img_path).convert("RGB")

        if self.transform:
            image = self.transform(image)

# Convert the numeric label to a tensor of type long (required by␣
 ↪CrossEntropyLoss).
        label_tensor = torch.tensor(label_idx, dtype=torch.long)
        # Return a tuple (image_tensor, label_tensor) for this sample.
        return image, label_tensor


# Similar to ButterflyTrainDataset, but without labels.
```

```
    # This is used for the test set where we only have filenames.
class ButterflyTestDataset(Dataset):
    def __init__(self, df, images_dir, transform=None):
         # Reset index to simplify indexing.
        self.df = df.reset_index(drop=True)
        self.images_dir = images_dir
        self.transform = transform

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

    def __getitem__(self, idx):
        row = self.df.iloc[idx]
        filename = row["filename"]

        img_path = os.path.join(self.images_dir, filename)
        image = Image.open(img_path).convert("RGB")

        if self.transform:
            image = self.transform(image)

# We return (image, filename) so we can later match predictions back to the␣
 ↪original file.
        return image, filename
```

[ ]:
```
# CREATE DATASET OBJECTS
# Here we instantiate our custom Dataset classes with the appropriate:
# DataFrames, directories, and transforms.
#  TRAIN / VALIDATION SPLIT (80% / 20%)

train_df_split, val_df = train_test_split(
    train_df,
    test_size=0.2,                  # 20% validation
    stratify=train_df["label_idx"],# keep class distribution balanced
    random_state=42
)

print("Train split:", train_df_split.shape)
print("Validation split:", val_df.shape)

#  DATASET OBJECTS (TRAIN / VAL / TEST)

train_dataset = ButterflyTrainDataset(
    df=train_df_split,
    images_dir=TRAIN_IMAGES_DIR,
    transform=train_transform
)
```

```
val_dataset = ButterflyTrainDataset(
    df=val_df,
    images_dir=TRAIN_IMAGES_DIR,
    transform=test_transform
)

test_dataset = ButterflyTestDataset(
    df=test_df,
    images_dir=TEST_IMAGES_DIR,
    transform=test_transform
)
```

```
Train split: (5199, 3)
Validation split: (1300, 3)
```

[144]:
```
# CREATE DATALOADERS
# DataLoaders handle batching, shuffling, and parallel loading of data.
# num_workers=0: important for Jupyter/macOS to avoid multiprocessing pickling␣
 ↪errors.

train_loader = DataLoader(
    train_dataset,
    batch_size=BATCH_SIZE,
    shuffle=True,   # shuffle=True: helps break any ordering in the data and␣
 ↪improves training.
    num_workers=0,  # 0 means data loading happens in the main process (safer␣
 ↪in notebooks).
)

test_loader = DataLoader(
    test_dataset,
    batch_size=BATCH_SIZE,
    shuffle=False,  # no need to shuffle test data; we want predictions in the␣
 ↪same order.
    num_workers=0,
)
```

[ ]:
```
#  SELECT DEVICE (GPU / MPS / CPU)
#  We use the best available device:
# - CUDA GPU if available
# - Apple Silicon MPS if available
# - Otherwise, fall back to CPU.

device = (
    torch.device("cuda") if torch.cuda.is_available()
    else torch.device("mps") if torch.backends.mps.is_available()
```

```
    else torch.device("cpu")
)
print("Using device:", device)
```

```
Using device: mps
```

[146]:
```
#  LOAD PRETRAINED RESNET18 MODEL
# We use transfer learning with a pretrained ResNet18 model on ImageNet.
# weights=... loads pretrained weights that already know generic image features.

resnet = models.resnet18(weights=models.ResNet18_Weights.IMAGENET1K_V1)
```

[147]:
```
# The original final fully connected (fc) layer outputs 1000 classes (ImageNet).
# We replace it with a new Linear layer that outputs num_classes (butterfly␣
 ↪species).
in_features = resnet.fc.in_features
resnet.fc = nn.Linear(in_features, num_classes)

# Move the model to the selected device (GPU / MPS / CPU).
resnet = resnet.to(device)


# DEFINE LOSS FUNCTION AND OPTIMIZER
# CrossEntropyLoss is standard for multi-class classification tasks.
# Adam is a commonly used optimizer that adapts the learning rate for each␣
 ↪parameter.

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(resnet.parameters(), lr=1e-4)
```

[ ]:
```
# TRAINING + VALIDATION LOOP

EPOCHS = 5   # number of passes over the whole training dataset

# Lists to store loss and accuracy over epochs for plotting.

# Loop over all epochs.

train_losses = []
val_losses   = []
train_accs   = []
val_accs     = []

for epoch in range(EPOCHS):

    # ---------- TRAIN ----------
```

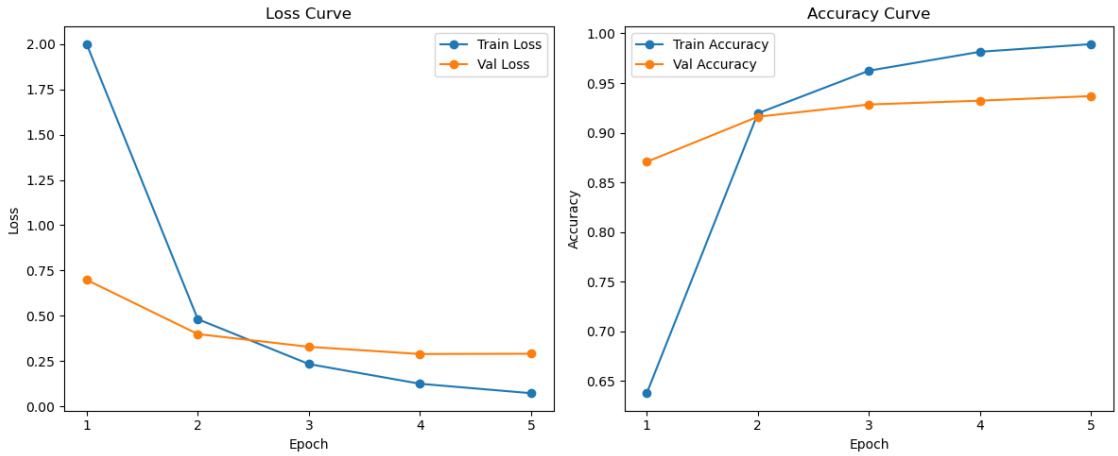
```
    resnet.train()                  # set model to training mode (enables␣
 ↪dropout/BN updates)
    running_loss = 0.0      # sum of training losses this epoch
    correct      = 0     # count of correct predictions
    total        = 0    # count of all samples seen

# Wrap train_loader with tqdm to get a progress bar.
    train_bar = tqdm(train_loader, desc=f"Train {epoch+1}/{EPOCHS}",␣
 ↪leave=False)

    # Iterate over mini-batches of training data.
    for images, labels in train_bar:
         # Move images and labels onto the selected device.
        images = images.to(device)
        labels = labels.to(device)

       # Clear gradients from the previous step.
        optimizer.zero_grad()         # reset gradients
        outputs = resnet(images)      # forward pass
        loss    = criterion(outputs, labels)   # Compute training loss for this␣
 ↪batch.

        loss.backward()               # backprop  compute gradients w.r.t. all␣
 ↪model parameters.
        optimizer.step()             # Optimizer step: update model parameters␣
 ↪using gradients.

 # Accumulate loss, scaled by batch size, for epoch average.
        running_loss += loss.item() * images.size(0)
        # Get predicted class index for each sample in the batch.
        _, preds = torch.max(outputs, 1)
         # Update correct and total counts for accuracy.
        correct += (preds == labels).sum().item()
        total   += labels.size(0)
 # Show current batch loss on the tqdm progress bar.
        train_bar.set_postfix(loss=loss.item())
# Compute average training loss and accuracy for this epoch.
    train_epoch_loss = running_loss / total
    train_epoch_acc  = correct / total

    # ---------- VALIDATION ----------
    resnet.eval()                     # set model to evaluation mode (no␣
 ↪dropout/BN updates)

    val_running_loss = 0.0      # sum of validation losses this epoch
    val_correct      = 0       # number of correct val predictions
```

```
    val_total        = 0      # number of val samples

# Disable gradient calculation for validation (faster, less memory).
    with torch.no_grad():
        # Loop over validation mini-batches.
        for images, labels in val_loader:
            # Move validation batch to the same device.
            images = images.to(device)
            labels = labels.to(device)
# Forward pass only (no backward or optimizer step).
            outputs = resnet(images)
              # Compute validation loss for this batch.
            loss    = criterion(outputs, labels)
 # Accumulate total loss for this epoch.
            val_running_loss += loss.item() * images.size(0)
              # Get predicted labels.
            _, preds = torch.max(outputs, 1)
             # Update counts for accuracy.
            val_correct += (preds == labels).sum().item()
            val_total   += labels.size(0)
 # Compute average validation loss and accuracy for the epoch.
    val_epoch_loss = val_running_loss / val_total
    val_epoch_acc  = val_correct / val_total

    # ---------- STORE + PRINT ----------
     # Save metrics to lists for plotting later.
    train_losses.append(train_epoch_loss)
    train_accs.append(train_epoch_acc)
    val_losses.append(val_epoch_loss)
    val_accs.append(val_epoch_acc)
 # Print a summary line for this epoch.
    print(f"Epoch {epoch+1}/{EPOCHS}")
    print(f"  Train Loss: {train_epoch_loss:.4f} | Train Acc: {train_epoch_acc:.
 ↪4f}")
    print(f"  Val Loss:   {val_epoch_loss:.4f} | Val Acc:   {val_epoch_acc:.
 ↪4f}")
```

```
Train 1/5:   0%|          | 0/163 [00:00<?, ?it/s]

Epoch 1/5
  Train Loss: 1.9991 | Train Acc: 0.6378
  Val Loss:   0.6986 | Val Acc:   0.8708

Train 2/5:   0%|          | 0/163 [00:00<?, ?it/s]

Epoch 2/5
  Train Loss: 0.4820 | Train Acc: 0.9196
  Val Loss:   0.3996 | Val Acc:   0.9162

Train 3/5:   0%|          | 0/163 [00:00<?, ?it/s]
```

```
Epoch 3/5
  Train Loss: 0.2341 | Train Acc: 0.9625
  Val Loss:   0.3294 | Val Acc:   0.9285

Train 4/5:   0%|          | 0/163 [00:00<?, ?it/s]

Epoch 4/5
  Train Loss: 0.1255 | Train Acc: 0.9815
  Val Loss:   0.2897 | Val Acc:   0.9323

Train 5/5:   0%|          | 0/163 [00:00<?, ?it/s]

Epoch 5/5
  Train Loss: 0.0734 | Train Acc: 0.9892
  Val Loss:   0.2911 | Val Acc:   0.9369
```

[149]:
```python
#PLOT TRAIN / VAL CURVES
# X-axis values: 1, 2, ..., number of epochs.
epochs_range = range(1, len(train_losses) + 1)
# Create a figure with two subplots side by side.
plt.figure(figsize=(12, 5))

# Loss curves.
plt.subplot(1, 2, 1)
plt.plot(epochs_range, train_losses, marker='o', label='Train Loss')
plt.plot(epochs_range, val_losses,   marker='o', label='Val Loss')
plt.title("Loss Curve")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.xticks(epochs_range)
plt.legend()

# Accuracy curves.
plt.subplot(1, 2, 2)
plt.plot(epochs_range, train_accs, marker='o', label='Train Accuracy')
plt.plot(epochs_range, val_accs,   marker='o', label='Val Accuracy')
plt.title("Accuracy Curve")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.xticks(epochs_range)
plt.legend()
# Adjust layout so subplots don't overlap.
plt.tight_layout()
plt.show()
```



1 Summary of Training vs Validation Loss Curve

2 The loss curve shows that:

1. Training loss decreases steadily from ~2.0 down to ~0.07 over 5 epochs.
2. The model is learning meaningful patterns from the butterfly images.
3. Validation loss decreases from ~0.70 to ~0.30 and stabilizes. → The model improves its performance on unseen data.
4. Training loss is lower than validation loss, but not by a large margin. → This indicates healthy learning with only mild overfitting, which is normal. # Overall: The loss curve demonstrates a strong learning process with no signs of training collapse or severe overfitting. The model is converging well.

3 Summary of Training vs Validation Accuracy Curve

4 The accuracy curve shows that:

1. Training accuracy increases sharply from ~64% to ~99% by epoch 5. → The model handles the training samples extremely well.
2. Validation accuracy improves from ~87% to ~94% and then stabilizes. → The model generalizes well to unseen validation images.
3. The gap between train (99%) and val (94%) accuracy is small, meaning: → No significant overfitting

→ Good generalization

→ The dataset and augmentations are effective

5 Overall:

The accuracy curve shows consistently strong improvement, with high performance on both training and validation sets. The model is highly accurate and well-generalized.

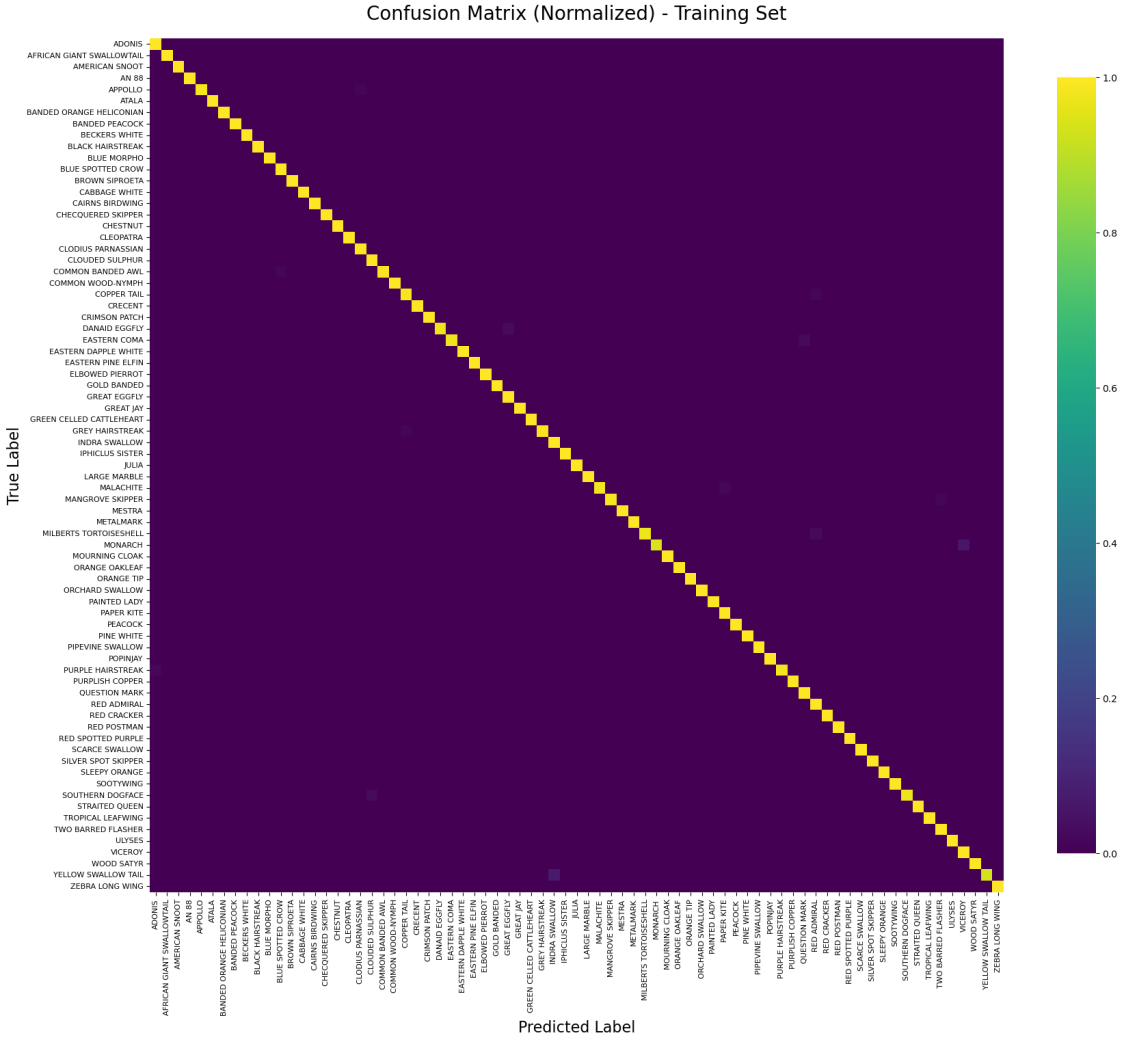
The model learns quickly and effectively, achieving very low loss and high accuracy on both training and validation sets, with only minimal overfitting — indicating a strong, well-generalized classifier.

[152]:
```
# PREDICTIONS ON TEST SET.
# Set model to evaluation mode before inference on test data.
resnet.eval()
all_filenames     = []    # store filenames in order
all_pred_indices  = []   # store predicted label indices

# No gradients needed for inference.
with torch.no_grad():
    # Iterate over all test batches.
    for images, filenames in tqdm(test_loader, desc="Predicting on Test Set"):
        images  = images.to(device)    # move images to device
        outputs = resnet(images)      # forward pass
        _, preds = torch.max(outputs, 1)  # predicted label indices


        preds = preds.cpu().numpy() # move predictions back to CPU as numpy
        all_filenames.extend(filenames) # accumulate filenames
        all_pred_indices.extend(preds)   # accumulate predictions

# Convert label indices back to label names.
all_pred_labels = [index_to_label[int(idx)] for idx in all_pred_indices]

# Add predictions to the test DataFrame.
test_df["pred_label_idx"] = all_pred_indices
test_df["pred_label"]     = all_pred_labels
print(test_df.head())
```

```
Predicting on Test Set:   0%|          | 0/88 [00:00<?, ?it/s]

      filename  pred_label_idx             pred_label
0  Image_1.jpg              52            PINE WHITE
1  Image_2.jpg              24         CRIMSON PATCH
2  Image_3.jpg               0                ADONIS
3  Image_4.jpg              36        IPHICLUS SISTER
4  Image_5.jpg              43  MILBERTS TORTOISESHELL
```

[153]:
```
# CONFUSION MATRIX ON TRAIN SET.
# Evaluate how the model performs on the training set per class.
resnet.eval()
all_true = []              # true labels from train set
all_pred = []            # predicted labels from train set
```

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

        outputs = resnet(images)
        _, preds = torch.max(outputs, 1)

        all_true.extend(labels.cpu().numpy())
        all_pred.extend(preds.cpu().numpy())

# Compute confusion matrix (rows = true label, cols = predicted label).
cm = confusion_matrix(all_true, all_pred)
# Normalize rows to show percentages instead of raw counts.
cm_norm = cm.astype("float") / cm.sum(axis=1, keepdims=True)

import seaborn as sns
plt.figure(figsize=(18, 16))
sns.heatmap(
    cm_norm,
    annot=False,
    cmap="viridis",
    square=True,
    cbar_kws={'shrink': 0.8},
    xticklabels=class_names,
    yticklabels=class_names
)
plt.title("Confusion Matrix (Normalized) - Training Set", fontsize=20, pad=20)
plt.xlabel("Predicted Label", fontsize=16)
plt.ylabel("True Label", fontsize=16)
plt.xticks(rotation=90, fontsize=8)
plt.yticks(rotation=0, fontsize=8)
plt.tight_layout()
plt.show()
```



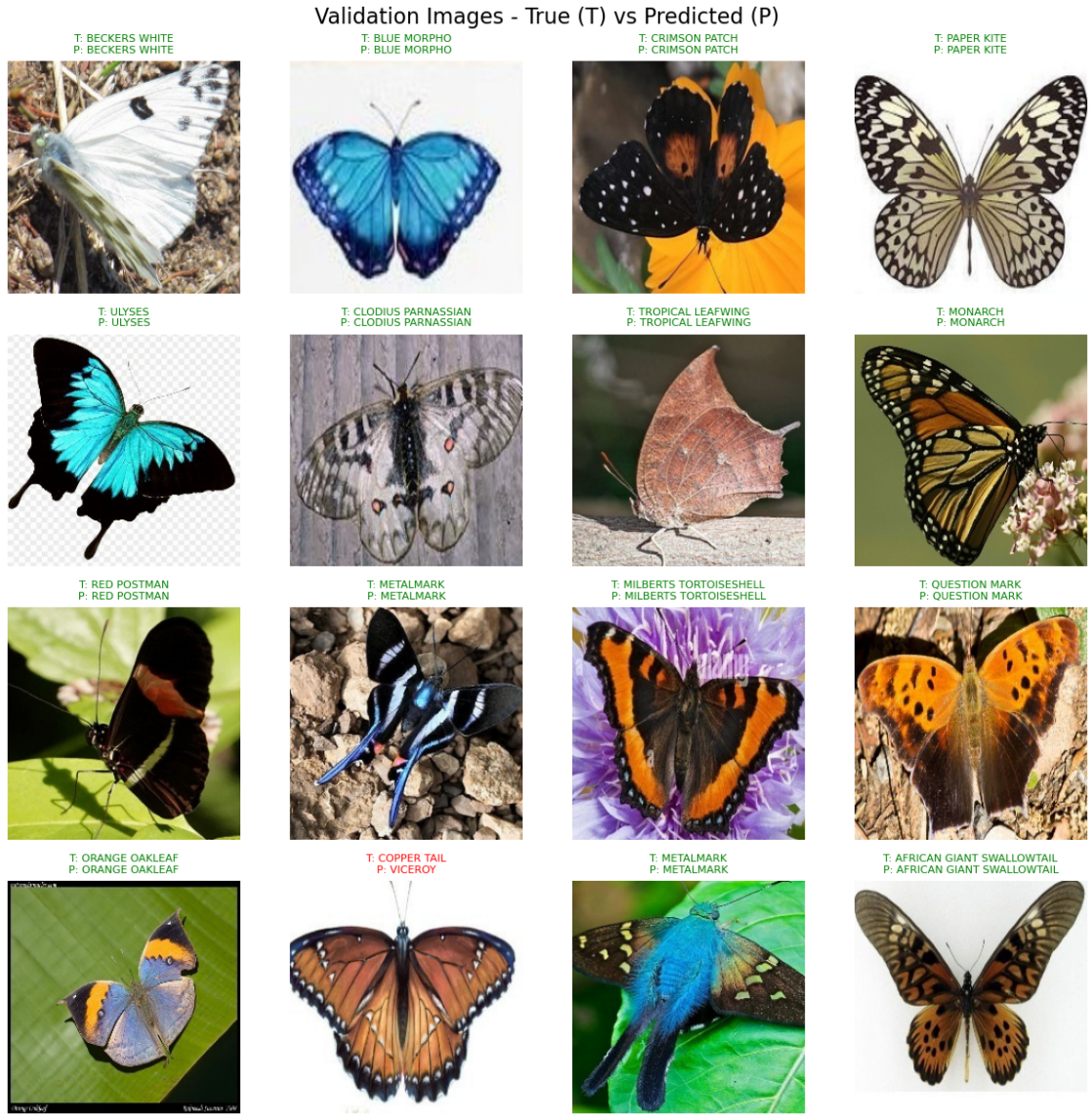
6 Confusion Matrix Summary (Training Set)

The confusion matrix shows near-perfect predictions on the training set: # Clear diagonal line 1. The bright yellow diagonal indicates the model correctly classifies almost every training example for all butterfly species. 2. Each class maps strongly to itself, showing the model has learned the training labels extremely well. # Almost no off-diagonal values 1. The surrounding dark purple areas represent zero or near-zero misclassifications. 2. Only a few tiny light spots appear off the diagonal, meaning very few mistakes on the training data. # Indicates strong model learning 1. The model captures distinctive visual features of each butterfly species. 2. No sign of major confusion between classes in the training set. # Mild overfitting but expected 1. Training accuracy is ~99% 2. Validation accuracy is ~94% 3. The confusion matrix confirms that the high training accuracy is real, not due to label errors or pipeline mistakes.

The confusion matrix reveals almost perfect training performance, with the model correctly identifying all butterfly species and making extremely few mistakes.

[154]:
```
# VALIDATION IMAGE GRID: TRUE (T) VS PREDICTED (P)
# Inverse transform to undo ImageNet normalization for plotting.
# Define a transform to undo ImageNet normalization for visualization.
inv_normalize = transforms.Normalize(
    mean=[-0.485 / 0.229, -0.456 / 0.224, -0.406 / 0.225],
    std =[1 / 0.229, 1 / 0.224, 1 / 0.225]
)

# Take one batch from validation loader.
images, labels = next(iter(val_loader))
images_device  = images.to(device)

# Run the model on this batch to get predictions.
resnet.eval()
with torch.no_grad():
    outputs = resnet(images_device)
    _, preds = torch.max(outputs, 1)

# Move predictions and labels back to CPU for display.
preds  = preds.cpu()
labels = labels.cpu()

# Decide how many images to show (up to 16).
num_to_show = min(16, images.size(0))
plt.figure(figsize=(12, 12))

# Loop over the first `num_to_show` images and plot each.
for i in range(num_to_show):
    plt.subplot(4, 4, i + 1)

    # Undo normalization so images look "normal" again.
    img = inv_normalize(images[i])
    img = img.permute(1, 2, 0).numpy()   # convert from C,H,W → H,W,C
    img = np.clip(img, 0, 1)               # clamp pixel values to [0, 1]

    plt.imshow(img)
    plt.axis("off")
    # readable true and predicted labels.
    true_label = index_to_label[int(labels[i])]
    pred_label = index_to_label[int(preds[i])]
    # Use green title if correct, red if wrong.
    color = "green" if true_label == pred_label else "red"

    plt.title(f"T: {true_label}\nP: {pred_label}", color=color, fontsize=8)
# Add an overall title and tidy up layout.
plt.suptitle("Validation Images - True (T) vs Predicted (P)", fontsize=16)
plt.tight_layout()
```

```
plt.show()
```


Validation Images - True (T) vs Predicted (P)
T: BECKERS WHITE
P: BECKERS WHITE
T: BLUE MORPHO
P: BLUE MORPHO
T: CRIMSON PATCH
P: CRIMSON PATCH
T: PAPER KITE
P: PAPER KITE
T: ULYSES
P: ULYSES
T: CLODIUS PARNASSIAN
P: CLODIUS PARNASSIAN
T: TROPICAL LEAFWING
P: TROPICAL LEAFWING
T: MONARCH
P: MONARCH
T: RED POSTMAN
P: RED POSTMAN
T: METALMARK
P: METALMARK
T: MILBERTS TORTOISESHELL
P: MILBERTS TORTOISESHELL
T: QUESTION MARK
P: QUESTION MARK
T: ORANGE OAKLEAF
P: ORANGE OAKLEAF
T: COPPER TAIL
P: VICEROY
T: METALMARK
P: METALMARK
T: AFRICAN GIANT SWALLOWTAIL
P: AFRICAN GIANT SWALLOWTAIL