Homework: Galaxy Image Classification

Course: Deep Learning for Computer Vision

Objective: Train a deep learning model to classify galaxy images from the Galaxy10 DECals dataset into one of 10 categories.

Dataset: Galaxy10 DECals

- **Source:** Hugging Face Datasets
- **Description:** Contains 17,736 color galaxy images (256x256 pixels) divided into 10 classes. Images originate from DESI Legacy Imaging Surveys, with labels from Galaxy Zoo.
- **Classes:**
  - 0: Disturbed Galaxies
  - 1: Merging Galaxies
  - 2: Round Smooth Galaxies
  - 3: In-between Round Smooth Galaxies
  - 4: Cigar Shaped Smooth Galaxies
  - 5: Barred Spiral Galaxies
  - 6: Unbarred Tight Spiral Galaxies
  - 7: Unbarred Loose Spiral Galaxies
  - 8: Edge-on Galaxies without Bulge
  - 9: Edge-on Galaxies with Bulge

Tasks:

1. Load and explore the dataset.
2. Preprocess the images.
3. Define and train a model.
4. Evaluate the model's performance using standard classification metrics on the test set.

Homework is succesfully completed if you get >0.9 Accuracy on the Test set.

Prerequisites

```
!pip install datasets scikit-learn matplotlib numpy -q >> None

import datasets
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score,
precision_recall_fscore_support, confusion_matrix,
ConfusionMatrixDisplay
```

```
ERROR: pip's dependency resolver does not currently take into account
all the packages that are installed. This behaviour is the source of
the following dependency conflicts.
gcsfs 2024.10.0 requires fsspec==2024.10.0, but you have fsspec
2024.12.0 which is incompatible.
torch 2.5.1+cu124 requires nvidia-cublas-cu12==12.4.5.8;
platform_system == "Linux" and platform_machine == "x86_64", but you
have nvidia-cublas-cu12 12.8.4.1 which is incompatible.
torch 2.5.1+cu124 requires nvidia-cudnn-cu12==9.1.0.70;
platform_system == "Linux" and platform_machine == "x86_64", but you
have nvidia-cudnn-cu12 9.3.0.75 which is incompatible.
torch 2.5.1+cu124 requires nvidia-cufft-cu12==11.2.1.3;
platform_system == "Linux" and platform_machine == "x86_64", but you
have nvidia-cufft-cu12 11.3.3.83 which is incompatible.
torch 2.5.1+cu124 requires nvidia-curand-cu12==10.3.5.147;
platform_system == "Linux" and platform_machine == "x86_64", but you
have nvidia-curand-cu12 10.3.9.90 which is incompatible.
torch 2.5.1+cu124 requires nvidia-cusolver-cu12==11.6.1.9;
platform_system == "Linux" and platform_machine == "x86_64", but you
have nvidia-cusolver-cu12 11.7.3.90 which is incompatible.
torch 2.5.1+cu124 requires nvidia-cusparse-cu12==12.3.1.170;
platform_system == "Linux" and platform_machine == "x86_64", but you
have nvidia-cusparse-cu12 12.5.8.93 which is incompatible.
torch 2.5.1+cu124 requires nvidia-nvjitlink-cu12==12.4.127;
platform_system == "Linux" and platform_machine == "x86_64", but you
have nvidia-nvjitlink-cu12 12.8.93 which is incompatible.
bigframes 1.36.0 requires rich<14,>=12.4.4, but you have rich 14.0.0
which is incompatible.

# Cell 4: Visualize one example from each class
def show_class_examples(dataset, class_names_map, samples_per_row=5,
num_rows=2):
    """Displays one sample image for each class."""
    if not dataset:
        print("Dataset not loaded. Cannot visualize.")
        return

    num_classes_to_show = len(class_names_map)
    if num_classes_to_show > samples_per_row * num_rows:
        print(f"Warning: Not enough space to show all
{num_classes_to_show} classes.")
        num_classes_to_show = samples_per_row * num_rows

    fig, axes = plt.subplots(num_rows, samples_per_row, figsize=(15,
6)) # Adjusted figsize
    axes = axes.ravel() # Flatten the axes array

    split_name = 'train' if 'train' in dataset else
list(dataset.keys())[0]
    data_split = dataset[split_name]
```

```
    images_shown = 0
    processed_labels = set()

    for i in range(len(data_split)):
        if images_shown >= num_classes_to_show:
            break # Stop once we have shown one for each target class

        example = data_split[i]
        label = example['label']

        if label not in processed_labels and label <
num_classes_to_show:
            img = example['image']
            ax_idx = label # Use label directly as index into the
flattened axes
            axes[ax_idx].imshow(img)
            axes[ax_idx].set_title(f"Class {label}:
{class_names_map[label]}", fontsize=9)
            axes[ax_idx].axis('off')
            processed_labels.add(label)
            images_shown += 1

    # Hide any unused subplots
    for i in range(images_shown, len(axes)):
        axes[i].axis('off')

    plt.tight_layout()
    plt.show()

def evaluate_predictions(predicted_labels, true_labels,
class_names_list):
    """
    Calculates and prints classification metrics from predicted labels
and true labels.

    Args:
        predicted_labels (list or np.array): The predicted class
indices for the test set.
        true_labels (list or np.array): The ground truth class indices
for the test set.
        class_names_list (list): A list of strings containing the
names of the classes.
    """
    if len(predicted_labels) != len(true_labels):
        print(f"Error: Number of predictions ({len(predicted_labels)})
does not match number of true labels ({len(true_labels)}).")
        return None # Indicate failure

    print(f"Evaluating {len(predicted_labels)} predictions against
```

```
true labels...")

    # Ensure inputs are numpy arrays for scikit-learn
    predicted_labels = np.array(predicted_labels)
    true_labels = np.array(true_labels)

    # Calculate metrics using scikit-learn
    accuracy = accuracy_score(true_labels, predicted_labels)
    # Calculate precision, recall, f1 per class and average (weighted)
    # Use zero_division=0 to handle cases where a class might not be
predicted or present in labels
    precision, recall, f1, _ = precision_recall_fscore_support(
        true_labels, predicted_labels, average='weighted',
zero_division=0
    )
    # Get per-class metrics as well
    per_class_precision, per_class_recall, per_class_f1,
per_class_support = precision_recall_fscore_support(
        true_labels, predicted_labels, average=None, zero_division=0,
labels=range(len(class_names_list))
    )

    # Generate Confusion Matrix
    cm = confusion_matrix(true_labels, predicted_labels,
labels=range(len(class_names_list)))

    # Print Metrics
    print(f"\n--- Evaluation Metrics ---")
    print(f"Accuracy: {accuracy:.4f}")
    print(f"Weighted Precision: {precision:.4f}")
    print(f"Weighted Recall: {recall:.4f}")
    print(f"Weighted F1-Score: {f1:.4f}")
    print("-" * 25)
    print("Per-Class Metrics:")
    print(f"{'Class':<30} | {'Precision':<10} | {'Recall':<10} | {'F1-
Score':<10} | {'Support':<10}")
    print("-" * 80)
    for i, name in enumerate(class_names_list):
         # Handle cases where support might be 0 for a class in true
labels if dataset is small/filtered
         support = per_class_support[i] if i < len(per_class_support)
else 0
         prec = per_class_precision[i] if i < len(per_class_precision)
else 0
         rec = per_class_recall[i] if i < len(per_class_recall) else 0
         f1s = per_class_f1[i] if i < len(per_class_f1) else 0
         print(f"{f'{i}: {name}':<30} | {prec:<10.4f} | {rec:<10.4f} |
{f1s:<10.4f} | {support:<10}")
    print("-" * 80)
```

```
    # Plot Confusion Matrix
    print("\nPlotting Confusion Matrix...")
    fig, ax = plt.subplots(figsize=(10, 10))
    disp = ConfusionMatrixDisplay(confusion_matrix=cm,
display_labels=class_names_list)
    disp.plot(cmap=plt.cm.Blues, ax=ax, xticks_rotation='vertical')
    plt.title('Confusion Matrix')
    plt.tight_layout() # Adjust layout to prevent overlap
    plt.show()

    metrics = {
        'accuracy': accuracy,
        'precision_weighted': precision,
        'recall_weighted': recall,
        'f1_weighted': f1,
        'confusion_matrix': cm,
        'per_class_metrics': {
            'precision': per_class_precision,
            'recall': per_class_recall,
            'f1': per_class_f1,
            'support': per_class_support
        }
    }
    return metrics

def per_class_precision(predicted_labels, true_labels,
class_names_list):
    # Get per-class metrics as well
    per_class_precision, per_class_recall, per_class_f1,
per_class_support = precision_recall_fscore_support(
        true_labels, predicted_labels, average=None, zero_division=0,
labels=range(len(class_names_list))
    )

    return per_class_precision
```

Data

```
dataset_name = "matthieulel/galaxy10_decals"
galaxy_dataset = datasets.load_dataset(dataset_name)

# Define class names based on the dataset card
class_names = [
    "Disturbed", "Merging", "Round Smooth", "In-between Round Smooth",
    "Cigar Shaped Smooth", "Barred Spiral", "Unbarred Tight Spiral",
    "Unbarred Loose Spiral", "Edge-on without Bulge", "Edge-on with
Bulge"
]
```

```python
# Create a dictionary for easy lookup
label2name = {i: name for i, name in enumerate(class_names)}
name2label = {name: i for i, name in enumerate(class_names)}

num_classes = len(class_names)
print(f"\nNumber of classes: {num_classes}")
print("Class names:", class_names)
```


{"model_id":"0a882f1e62254e51b59c43e7edbfceac","version_major":2,"version_minor":0}

{"model_id":"3dcf42786b8e4bcf97e59503e1b7f10e","version_major":2,"version_minor":0}

{"model_id":"7dc31d0383ce409f840581608379e857","version_major":2,"version_minor":0}

{"model_id":"c1ec13272cbf4059aff0a698d0bf9397","version_major":2,"version_minor":0}

{"model_id":"acb63ed1628c4c8093c16064f6d939f3","version_major":2,"version_minor":0}

{"model_id":"250e1431aa6d471da2af98bf5075a23c","version_major":2,"version_minor":0}

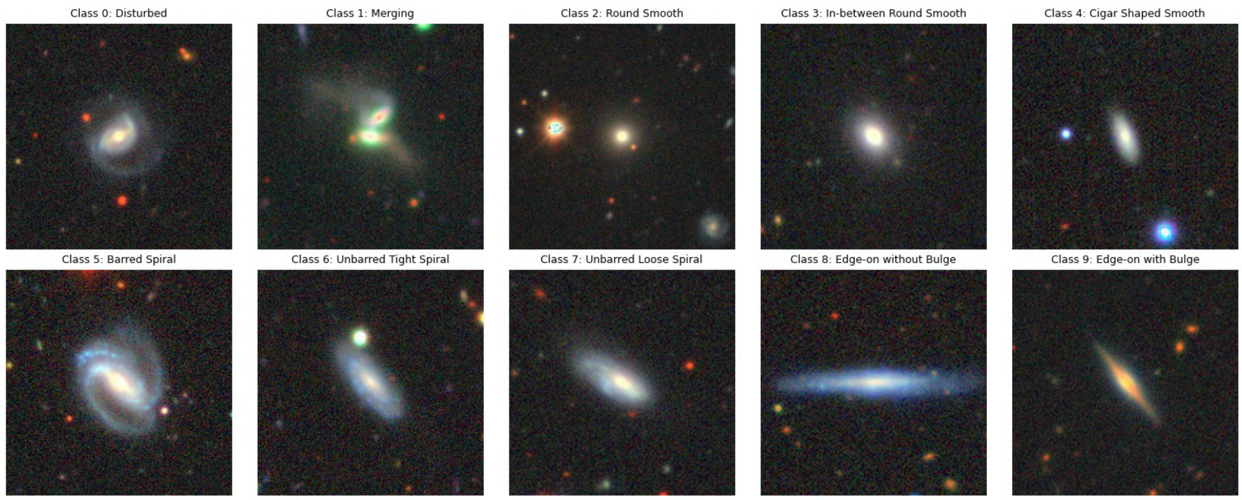
{"model_id":"8f9981c6e8d74cc58dda9d7ee3beebcd","version_major":2,"version_minor":0}

{"model_id":"68239d6dbaff4d62b931a952f9a2bdf4","version_major":2,"version_minor":0}

{"model_id":"7d6220b8aeb8413ab2f5847e9b380927","version_major":2,"version_minor":0}


```
Number of classes: 10
Class names: ['Disturbed', 'Merging', 'Round Smooth', 'In-between
Round Smooth', 'Cigar Shaped Smooth', 'Barred Spiral', 'Unbarred Tight
Spiral', 'Unbarred Loose Spiral', 'Edge-on without Bulge', 'Edge-on
with Bulge']
```

```python
show_class_examples(galaxy_dataset, label2name, samples_per_row=5,
num_rows=2)
```



Your training code here

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torchvision.models import resnet18, ResNet18_Weights
from torch.utils.data import Dataset, DataLoader
import matplotlib.pyplot as plt
from tqdm import tqdm

device = 'cuda'
# device = 'cpu'

elems_per_class = [0] * 10
for el in tqdm(galaxy_dataset['train']):
    elems_per_class[el['label']] += 1
elems_per_class
```

```
100%|██████████| 15962/15962 [00:42<00:00, 375.39it/s]
```

```
[972, 1668, 2395, 1829, 306, 1826, 1650, 2355, 1266, 1695]
```

```
elems_per_class = [972, 1668, 2395, 1829, 306, 1826, 1650, 2355, 1266,
1695]

class_weights = np.sum(elems_per_class) / (elems_per_class)
class_weights = class_weights / class_weights.sum()
class_weights
```

```
array([0.1166609 , 0.06798225, 0.0473463 , 0.06199803, 0.37056991,
       0.06209989, 0.06872387, 0.04815048, 0.08956903, 0.06689935])
```

```
class GalaxyDataset(Dataset):
    def __init__(self, data, transform=None):
        self.data = data
        self.transform = transform

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

    def __getitem__(self, idx):
        sample = self.data[idx]
        image = sample['image']
        label = sample['label']

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

        return image, label

num_classes = 10
batch_size = 64

transform = transforms.Compose([
    transforms.Resize(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225]),
])
train_dataset = GalaxyDataset(galaxy_dataset['train'],
transform=transform)
test_dataset = GalaxyDataset(galaxy_dataset['test'],
transform=transform)

train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False)
```

My first try was tuning pretrained ResNet-50 with few unfrozen layers.

```
import torchvision.models as models
import torch.nn as nn

model = models.resnet50(pretrained=True)
for param in model.parameters():
    param.requires_grad = False

for param in model.layer4.parameters():
```

```
    param.requires_grad = True

model.fc = nn.Sequential(
    nn.BatchNorm1d(model.fc.in_features),
    nn.Linear(num_ftrs, 256),
    nn.ReLU(),
    nn.Dropout(0.4),
    nn.Linear(256, 10)
)

/usr/local/lib/python3.11/dist-packages/torchvision/models/
_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated
since 0.13 and may be removed in the future, please use 'weights'
instead.
  warnings.warn(
/usr/local/lib/python3.11/dist-packages/torchvision/models/_utils.py:2
23: UserWarning: Arguments other than a weight enum or `None` for
'weights' are deprecated since 0.13 and may be removed in the future.
The current behavior is equivalent to passing
`weights=ResNet50_Weights.IMAGENET1K_V1`. You can also use
`weights=ResNet50_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/resnet50-
0676ba61.pth" to /root/.cache/torch/hub/checkpoints/resnet50-
0676ba61.pth
100%|██████████| 97.8M/97.8M [00:00<00:00, 194MB/s]

model = model.to(device)

learning_rate = 0.001
num_epochs = 10
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0

    # Progress bar
    loop = tqdm(train_loader, desc=f'Epoch [{epoch+1}/{num_epochs}]')

    for images, labels in loop:
        images = images.to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
```

```
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        loop.set_postfix(loss=loss.item())

    # Calculate average training loss for the epoch
    epoch_loss = running_loss / len(train_loader)
    print(epoch_loss, ' on epoch ', epoch+1)
```

```
Epoch [1/10]: 100%|██████████| 250/250 [02:27<00:00,  1.69it/s,
loss=1.06]

1.1990398824214936  on epoch  1

Epoch [2/10]: 100%|██████████| 250/250 [02:25<00:00,  1.72it/s,
loss=0.723]

0.8481011377573013  on epoch  2

Epoch [3/10]: 100%|██████████| 250/250 [02:25<00:00,  1.72it/s,
loss=0.682]

0.6880607516765594  on epoch  3

Epoch [4/10]: 100%|██████████| 250/250 [02:25<00:00,  1.72it/s,
loss=0.575]

0.47687066036462783  on epoch  4

Epoch [5/10]: 100%|██████████| 250/250 [02:26<00:00,  1.71it/s,
loss=0.359]

0.31639994484186174  on epoch  5

Epoch [6/10]: 100%|██████████| 250/250 [02:24<00:00,  1.72it/s,
loss=0.333]

0.20528729942440987  on epoch  6

Epoch [7/10]: 100%|██████████| 250/250 [02:25<00:00,  1.72it/s,
loss=0.312]

0.13291373752057553  on epoch  7

Epoch [8/10]: 100%|██████████| 250/250 [02:24<00:00,  1.73it/s,
loss=0.035]

0.2232086379826069  on epoch  8

Epoch [9/10]: 100%|██████████| 250/250 [02:24<00:00,  1.73it/s,
loss=0.366]
```

```
0.12928173715993763  on epoch  9

Epoch [10/10]: 100%|██████████| 250/250 [02:24<00:00,  1.73it/s,
loss=1.15]

0.09974944451637566  on epoch  10
```

It achieved decent results on train, however on test it was not that good. test accuracy -- 0.72

Next try was using supposedly task-specific model WaveMix

At first I calculated image statistics on train, not relying on standard values.

```python
import torch
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import numpy as np

simple_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor()
])

train_dataset = GalaxyDataset(galaxy_dataset['train'],
simple_transform)

loader = DataLoader(train_dataset, batch_size=64, shuffle=False,
num_workers=2)

mean = 0.0
std = 0.0
nb_samples = 0.0

for images, _ in loader:
    batch_samples = images.size(0)
    images = images.view(batch_samples, images.size(1), -1)

    mean += images.mean(2).sum(0)
    std += images.std(2).sum(0)
    nb_samples += batch_samples

mean /= nb_samples
std /= nb_samples

print(f'Mean: {mean}')
print(f'Std: {std}')
```

here are the mean/std

mean=[0.1678, 0.1629, 0.1592], # Our means

std=[0.1164, 0.1065, 0.0985] # Our stds

```
! pip install wavemix
```

```
Collecting wavemix
  Downloading wavemix-0.2.4-py3-none-any.whl.metadata (10 kB)
Requirement already satisfied: einops in
/usr/local/lib/python3.11/dist-packages (from wavemix) (0.8.1)
Requirement already satisfied: torch in
/usr/local/lib/python3.11/dist-packages (from wavemix) (2.5.1+cu124)
Requirement already satisfied: torchvision in
/usr/local/lib/python3.11/dist-packages (from wavemix) (0.20.1+cu124)
Requirement already satisfied: pywavelets in
/usr/local/lib/python3.11/dist-packages (from wavemix) (1.8.0)
Requirement already satisfied: numpy in
/usr/local/lib/python3.11/dist-packages (from wavemix) (1.26.4)
Requirement already satisfied: mkl_fft in
/usr/local/lib/python3.11/dist-packages (from numpy->wavemix) (1.3.8)
Requirement already satisfied: mkl_random in
/usr/local/lib/python3.11/dist-packages (from numpy->wavemix) (1.2.4)
Requirement already satisfied: mkl_umath in
/usr/local/lib/python3.11/dist-packages (from numpy->wavemix) (0.1.1)
Requirement already satisfied: mkl in /usr/local/lib/python3.11/dist-
packages (from numpy->wavemix) (2025.1.0)
Requirement already satisfied: tbb4py in
/usr/local/lib/python3.11/dist-packages (from numpy->wavemix)
(2022.1.0)
Requirement already satisfied: mkl-service in
/usr/local/lib/python3.11/dist-packages (from numpy->wavemix) (2.4.1)
Requirement already satisfied: filelock in
/usr/local/lib/python3.11/dist-packages (from torch->wavemix) (3.18.0)
Requirement already satisfied: typing-extensions>=4.8.0 in
/usr/local/lib/python3.11/dist-packages (from torch->wavemix) (4.13.1)
Requirement already satisfied: networkx in
/usr/local/lib/python3.11/dist-packages (from torch->wavemix) (3.4.2)
Requirement already satisfied: jinja2 in
/usr/local/lib/python3.11/dist-packages (from torch->wavemix) (3.1.6)
Requirement already satisfied: fsspec in
/usr/local/lib/python3.11/dist-packages (from torch->wavemix)
(2024.12.0)
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.4.127 in
/usr/local/lib/python3.11/dist-packages (from torch->wavemix)
(12.4.127)
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.4.127
in /usr/local/lib/python3.11/dist-packages (from torch->wavemix)
(12.4.127)
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.4.127 in
/usr/local/lib/python3.11/dist-packages (from torch->wavemix)
```

```
(12.4.127)
Collecting nvidia-cudnn-cu12==9.1.0.70 (from torch->wavemix)
  Downloading nvidia_cudnn_cu12-9.1.0.70-py3-none-
manylinux2014_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cublas-cu12==12.4.5.8 (from torch->wavemix)
  Downloading nvidia_cublas_cu12-12.4.5.8-py3-none-
manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cufft-cu12==11.2.1.3 (from torch->wavemix)
  Downloading nvidia_cufft_cu12-11.2.1.3-py3-none-
manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-curand-cu12==10.3.5.147 (from torch->wavemix)
  Downloading nvidia_curand_cu12-10.3.5.147-py3-none-
manylinux2014_x86_64.whl.metadata (1.5 kB)
Collecting nvidia-cusolver-cu12==11.6.1.9 (from torch->wavemix)
  Downloading nvidia_cusolver_cu12-11.6.1.9-py3-none-
manylinux2014_x86_64.whl.metadata (1.6 kB)
Collecting nvidia-cusparse-cu12==12.3.1.170 (from torch->wavemix)
  Downloading nvidia_cusparse_cu12-12.3.1.170-py3-none-
manylinux2014_x86_64.whl.metadata (1.6 kB)
Requirement already satisfied: nvidia-nccl-cu12==2.21.5 in
/usr/local/lib/python3.11/dist-packages (from torch->wavemix) (2.21.5)
Requirement already satisfied: nvidia-nvtx-cu12==12.4.127 in
/usr/local/lib/python3.11/dist-packages (from torch->wavemix)
(12.4.127)
Collecting nvidia-nvjitlink-cu12==12.4.127 (from torch->wavemix)
  Downloading nvidia_nvjitlink_cu12-12.4.127-py3-none-
manylinux2014_x86_64.whl.metadata (1.5 kB)
Requirement already satisfied: triton==3.1.0 in
/usr/local/lib/python3.11/dist-packages (from torch->wavemix) (3.1.0)
Requirement already satisfied: sympy==1.13.1 in
/usr/local/lib/python3.11/dist-packages (from torch->wavemix) (1.13.1)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in
/usr/local/lib/python3.11/dist-packages (from sympy==1.13.1->torch-
>wavemix) (1.3.0)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in
/usr/local/lib/python3.11/dist-packages (from torchvision->wavemix)
(11.1.0)
Requirement already satisfied: MarkupSafe>=2.0 in
/usr/local/lib/python3.11/dist-packages (from jinja2->torch->wavemix)
(3.0.2)
Requirement already satisfied: intel-openmp<2026,>=2024 in
/usr/local/lib/python3.11/dist-packages (from mkl->numpy->wavemix)
(2024.2.0)
Requirement already satisfied: tbb==2022.* in
/usr/local/lib/python3.11/dist-packages (from mkl->numpy->wavemix)
(2022.1.0)
Requirement already satisfied: tcmlib==1.* in
/usr/local/lib/python3.11/dist-packages (from tbb==2022.*->mkl->numpy-
>wavemix) (1.2.0)
```

```
Requirement already satisfied: intel-cmplr-lib-rt in
/usr/local/lib/python3.11/dist-packages (from mkl_umath->numpy-
>wavemix) (2024.2.0)
Requirement already satisfied: intel-cmplr-lib-ur==2024.2.0 in
/usr/local/lib/python3.11/dist-packages (from intel-
openmp<2026,>=2024->mkl->numpy->wavemix) (2024.2.0)
Downloading wavemix-0.2.4-py3-none-any.whl (11 kB)
Downloading nvidia_cublas_cu12-12.4.5.8-py3-none-
manylinux2014_x86_64.whl (363.4 MB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 363.4/363.4 MB 4.7 MB/s eta
0:00:000:00:0100:01
anylinux2014_x86_64.whl (664.8 MB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 664.8/664.8 MB 1.9 MB/s eta
0:00:000:00:0100:01
anylinux2014_x86_64.whl (211.5 MB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 211.5/211.5 MB 6.9 MB/s eta
0:00:000:00:0100:01
anylinux2014_x86_64.whl (56.3 MB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 56.3/56.3 MB 30.5 MB/s eta
0:00:00:00:0100:01
anylinux2014_x86_64.whl (127.9 MB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 127.9/127.9 MB 13.6 MB/s eta
0:00:00:00:0100:01
anylinux2014_x86_64.whl (207.5 MB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 207.5/207.5 MB 2.9 MB/s eta
0:00:000:00:0100:01
anylinux2014_x86_64.whl (21.1 MB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 21.1/21.1 MB 80.7 MB/s eta
0:00:00:00:0100:01
ix
  Attempting uninstall: nvidia-nvjitlink-cu12
    Found existing installation: nvidia-nvjitlink-cu12 12.8.93
    Uninstalling nvidia-nvjitlink-cu12-12.8.93:
      Successfully uninstalled nvidia-nvjitlink-cu12-12.8.93
  Attempting uninstall: nvidia-curand-cu12
    Found existing installation: nvidia-curand-cu12 10.3.9.90
    Uninstalling nvidia-curand-cu12-10.3.9.90:
      Successfully uninstalled nvidia-curand-cu12-10.3.9.90
  Attempting uninstall: nvidia-cufft-cu12
    Found existing installation: nvidia-cufft-cu12 11.3.3.83
    Uninstalling nvidia-cufft-cu12-11.3.3.83:
      Successfully uninstalled nvidia-cufft-cu12-11.3.3.83
  Attempting uninstall: nvidia-cublas-cu12
    Found existing installation: nvidia-cublas-cu12 12.8.4.1
    Uninstalling nvidia-cublas-cu12-12.8.4.1:
      Successfully uninstalled nvidia-cublas-cu12-12.8.4.1
  Attempting uninstall: nvidia-cusparse-cu12
    Found existing installation: nvidia-cusparse-cu12 12.5.8.93
    Uninstalling nvidia-cusparse-cu12-12.5.8.93:
```

```
      Successfully uninstalled nvidia-cusparse-cu12-12.5.8.93
  Attempting uninstall: nvidia-cudnn-cu12
    Found existing installation: nvidia-cudnn-cu12 9.3.0.75
    Uninstalling nvidia-cudnn-cu12-9.3.0.75:
      Successfully uninstalled nvidia-cudnn-cu12-9.3.0.75
  Attempting uninstall: nvidia-cusolver-cu12
    Found existing installation: nvidia-cusolver-cu12 11.7.3.90
    Uninstalling nvidia-cusolver-cu12-11.7.3.90:
      Successfully uninstalled nvidia-cusolver-cu12-11.7.3.90
ERROR: pip's dependency resolver does not currently take into account
all the packages that are installed. This behaviour is the source of
the following dependency conflicts.
pylibcugraph-cu12 24.12.0 requires pylibraft-cu12==24.12.*, but you
have pylibraft-cu12 25.2.0 which is incompatible.
pylibcugraph-cu12 24.12.0 requires rmm-cu12==24.12.*, but you have
rmm-cu12 25.2.0 which is incompatible.
Successfully installed nvidia-cublas-cu12-12.4.5.8 nvidia-cudnn-cu12-
9.1.0.70 nvidia-cufft-cu12-11.2.1.3 nvidia-curand-cu12-10.3.5.147
nvidia-cusolver-cu12-11.6.1.9 nvidia-cusparse-cu12-12.3.1.170 nvidia-
nvjitlink-cu12-12.4.127 wavemix-0.2.4
```

```python
import torch, wavemix
from wavemix.classification import WaveMix

model = WaveMix(
    num_classes= 10,
    depth= 6,
    mult= 2,
    ff_channel= 192,
    final_dim= 192,
    dropout= 0.2,
    level=4,
    patch_size=4,
).to(device)
img = torch.randn(1, 3, 256, 256).to(device)

preds = model(img)
preds.shape
```

```
torch.Size([1, 10])
```

```python
model_parameters = filter(lambda p: p.requires_grad,
model.parameters())
params = sum([np.prod(p.size()) for p in model_parameters])
params
```

```
13468234
```

Model is quite big, however we could try even deeper architecture of it if needed. As we'll see in a couple of cells, I was not satisfied with this model and didn't try to make it work at all costs

```python
train_transform = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomVerticalFlip(p=0.5),
    transforms.RandomRotation(degrees=20),
    transforms.RandomGrayscale(p=0.2),
    transforms.ToTensor(),
    transforms.Normalize(
        mean=[0.1678, 0.1629, 0.1592],  # Our means
        std=[0.1164, 0.1065, 0.0985]    # Our stds
    )
])

val_transform = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.ToTensor(),
    transforms.Normalize(
        mean=[0.1678, 0.1629, 0.1592],  # Our means
        std=[0.1164, 0.1065, 0.0985]    # Our stds
    )
])

train_ds = GalaxyDataset(galaxy_dataset['train'])
all_labels = []
for idx in range(len(train_ds)):
    _, label = train_ds[idx]
    all_labels.append(label)

from torch.utils.data import Subset
from sklearn.model_selection import train_test_split

train_idx, val_idx = train_test_split(
    list(range(len(train_ds))),
    test_size=0.2,
    random_state=42,
    stratify=all_labels
)

class TransformSubset(torch.utils.data.Dataset):
    def __init__(self, subset, transform=None):
        self.subset = subset
        self.transform = transform

    def __getitem__(self, index):
        x, y = self.subset[index]
        if self.transform:
            x = self.transform(x)
        return x, y

    def __len__(self):
```

```
        return len(self.subset)

train_dataset = Subset(train_ds, train_idx)
val_dataset = Subset(train_ds, val_idx)
# Example
train_dataset = TransformSubset(train_dataset,
transform=train_transform)
val_dataset = TransformSubset(val_dataset, transform=val_transform)

print(f"Train samples: {len(train_dataset)}")
print(f"Val samples: {len(val_dataset)}")
```

```
Train samples: 12769
Val samples: 3193
```

```
test_dataset = GalaxyDataset(galaxy_dataset['test'],
transform=val_transform)
```

Here I also added validation subset, LR_scheduler and reweighted loss w.r.t. reverse class frequency. Throughout this whole task I was experimenting with different augmentations as well. One of the main approaches is shown above -- besides basic ones I've added probabilistic greyscale. hoping that classification depends on geometry only. It's a bit controversial, since afaik color corresponds with bounds of the space objects etc, but I still gave it a try. Unfortunately, no straightforward conclusion if it works :)

```
import torch
from torch.optim import AdamW
from torch.optim.lr_scheduler import CosineAnnealingLR, LinearLR

optimizer = AdamW(model.parameters(), lr=1e-4)
criterion =
nn.CrossEntropyLoss(weight=torch.tensor(class_weights).to(device).floa
t())

warmup_epochs = 5
total_epochs = 100

scheduler_warmup = LinearLR(
    optimizer,
    start_factor=0.01,
    end_factor=1.0,
    total_iters=warmup_epochs
)

scheduler_cosine = CosineAnnealingLR(
    optimizer,
    T_max=total_epochs - warmup_epochs,
    eta_min=1e-6
)
```

```
from torch.optim.lr_scheduler import SequentialLR
scheduler = SequentialLR(
    optimizer,
    schedulers=[scheduler_warmup, scheduler_cosine],
    milestones=[warmup_epochs]
)

batch_size = 64
train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size,
shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False)

for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0

    loop = tqdm(train_loader, desc=f'Epoch [{epoch+1}/{num_epochs}]')

    for images, labels in loop:
        images = images.to(device)
        labels = labels.to(device)

        outputs = model(images)
        loss = criterion(outputs, labels)


        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        loop.set_postfix(loss=loss.item())
    scheduler.step()

    epoch_loss = running_loss / len(train_loader)
    if epoch % 1 == 0:
        preds = []
        true_val_labels = []

        with torch.no_grad():
            for images, labels in val_loader:
                images = images.to(device)
                outputs = model(images)
                _, predicted = torch.max(outputs.data, 1)

                preds.extend(predicted.cpu().numpy())
                true_val_labels.extend(labels.cpu().numpy())
```

```python
        preds = np.array(preds)
        true_val_labels = np.array(true_val_labels)
        acc = (preds == true_val_labels).sum() / len(true_val_labels)
        print(epoch_loss, acc, 'loss/val acc on epoch ', epoch+1)


    model_path = f'wave_mix.pth'
    torch.save({
        'epoch': epoch+1,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'loss': epoch_loss,
    }, model_path)
```

```
Epoch [1/105]: 100%|██████████| 200/200 [06:48<00:00,  2.04s/it, loss=2]

1.9412593859434129 0.3617287817099906 loss/val acc on epoch  1

Epoch [2/105]: 100%|██████████| 200/200 [06:48<00:00,  2.04s/it, loss=1.43]

1.759851895570755 0.4293767616661447 loss/val acc on epoch  2

Epoch [3/105]:  13%|█         | 26/200 [00:53<05:55,  2.04s/it, loss=1.84]
```

I ran it several times with total of around 30 epochs, results were alright(~77 accuracy on validation). In the end, it was not close enough to 0.9, so we move further.

NEXT TRY is convnext.

```python
from torchvision.models import convnext_base, ConvNeXt_Base_Weights

# weights = ConvNeXt_Base_Weights.IMAGENET1K_V1
model = convnext_base()

num_classes = 10

# for param in model.parameters():
#     param.requires_grad = False

# # Unfreeze last layer (classifier)
# for param in model.classifier.parameters():
#     param.requires_grad = True

# Replace final classification layer
model.classifier = nn.Sequential(
        nn.AdaptiveAvgPool2d((1, 1)),
```

```
        nn.Flatten(),
        nn.LayerNorm(1024),
        nn.Linear(1024, num_classes)
    )
```

Here I decided to train the whole model already.

```
import torch
from torch.optim import AdamW
from torch.optim.lr_scheduler import CosineAnnealingLR, LinearLR

optimizer = AdamW(model.parameters(), lr=1e-3, weight_decay=1e-4)
criterion =
nn.CrossEntropyLoss(weight=torch.tensor(class_weights).to(device).floa
t())
warmup_epochs = 5
total_epochs = 100

scheduler_warmup = LinearLR(
    optimizer,
    start_factor=0.01,
    end_factor=1.0,
    total_iters=warmup_epochs
)

scheduler_cosine = CosineAnnealingLR(
    optimizer,
    T_max=total_epochs - warmup_epochs,
    eta_min=1e-6
)

from torch.optim.lr_scheduler import SequentialLR
scheduler = SequentialLR(
    optimizer,
    schedulers=[scheduler_warmup, scheduler_cosine],
    milestones=[warmup_epochs]
)

train_ds = GalaxyDataset(galaxy_dataset['train'])

train_ds = GalaxyDataset(galaxy_dataset['train'])
all_labels = []
for idx in range(len(train_ds)):
    _, label = train_ds[idx]
    all_labels.append(label)

train_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomVerticalFlip(p=0.5),
```

```python
    transforms.RandomRotation(degrees=20),
    transforms.RandomGrayscale(p=0.2),
    transforms.ToTensor(),
    transforms.Normalize(
        mean=[0.1678, 0.1629, 0.1592],  # Our means
        std=[0.1164, 0.1065, 0.0985]    # Our stds
    )
])

val_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(
        mean=[0.1678, 0.1629, 0.1592],  # Our means
        std=[0.1164, 0.1065, 0.0985]    # Our stds
    )
])

from torch.utils.data import Subset
from sklearn.model_selection import train_test_split

train_idx, val_idx = train_test_split(
    list(range(len(train_ds))),
    test_size=0.2,
    random_state=42,
    stratify=all_labels
)

class TransformSubset(torch.utils.data.Dataset):
    def __init__(self, subset, transform=None):
        self.subset = subset
        self.transform = transform

    def __getitem__(self, index):
        x, y = self.subset[index]
        if self.transform:
            x = self.transform(x)
        return x, y

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

train_dataset = Subset(train_ds, train_idx)
val_dataset = Subset(train_ds, val_idx)
train_dataset = TransformSubset(train_dataset,
transform=train_transform)
val_dataset = TransformSubset(val_dataset, transform=val_transform)

print(f"Train samples: {len(train_dataset)}")
print(f"Val samples: {len(val_dataset)}")
```

```
Train samples: 12769
Val samples: 3193
```

```python
# train_dataset = GalaxyDataset(galaxy_dataset['train'],
transform=train_transform)
test_dataset = GalaxyDataset(galaxy_dataset['test'],
transform=val_transform)

batch_size = 64
train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size,
shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False)

from google.colab import drive
import os
drive.mount('/content/drive')

save_dir = '/content/drive/MyDrive/galaxy_models'
```

```
Mounted at /content/drive
```

```python
model_path = os.path.join(save_dir, f'convnext_galaxy_epoch_{6}.pth')
res = torch.load(model_path, map_location=device)

model.load_state_dict(res['model_state_dict'])
```

```
<All keys matched successfully>
```

```python
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0

    loop = tqdm(train_loader, desc=f'Epoch [{epoch+1}/{num_epochs}]')

    for images, labels in loop:
        images = images.to(device)
        labels = labels.to(device)

        outputs = model(images)
        loss = criterion(outputs, labels)


        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        loop.set_postfix(loss=loss.item())
```

```python
    scheduler.step()

    epoch_loss = running_loss / len(train_loader)
    if epoch % 1 == 0:
        preds = []
        true_val_labels = []

        with torch.no_grad():
            for images, labels in val_loader:
                images = images.to(device)
                outputs = model(images)
                _, predicted = torch.max(outputs.data, 1)

                preds.extend(predicted.cpu().numpy())
                true_val_labels.extend(labels.cpu().numpy())
        preds = np.array(preds)
        true_val_labels = np.array(true_val_labels)
        acc = (preds == true_val_labels).sum() / len(true_val_labels)
        print(epoch_loss, acc, 'loss/val acc on epoch ', epoch+1)


    model_path = f'{save_dir}_epoch_{epoch+1}.pth'
    torch.save({
        'epoch': epoch+1,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'loss': epoch_loss,
    }, model_path)
```

Training of this model was done in another notebook(using exactly the same code). OK-ish results(0.82 val accuracy), but our final candidate was EfficientNet.

```python
import timm
model = timm.create_model('tf_efficientnet_b0_ns', pretrained=True,
num_classes=10).to(device)
```

```
/usr/local/lib/python3.11/dist-packages/timm/models/_factory.py:126:
UserWarning: Mapping deprecated model name tf_efficientnet_b0_ns to
current tf_efficientnet_b0.ns_jft_in1k.
  model = create_fn(
```

{"model_id":"a96da981b80043189c4d3b839bb4104e","version_major":2,"vers
ion_minor":0}

Let's use a WeightedLoader, which would more often yield samples from some classes(rare or worse performing). I've tried two strategies -- assigning weights as inverse of frequency(as below and most used) and also assigning weights as inverse of class precision on validation. Can't really say, which one is better or helped more. It seems that with weighted loss it is sort of double penalty for mistakes on rare classes, but still not obvious whether it's a good or bad thing.

```python
import torch
from torch.utils.data import DataLoader, Dataset
from collections import Counter
import math
from typing import Dict, Optional, Callable
import random

class WeightedDataLoader(DataLoader):
    def __init__(self,
                 dataset: Dataset,
                 label_weights: Optional[Dict[int, float]] = None,
                 batch_size: int = 1,
                 shuffle: bool = True,
                 replacement: bool = True,
                 num_samples: Optional[int] = None,
                 **kwargs):

        self.dataset = dataset
        self.shuffle = shuffle
        self.replacement = replacement
        self.epoch = 0
        self.current_weights = label_weights

        self.labels = self._extract_labels(dataset)
        self.label_counts = Counter(self.labels)
        self.num_classes = len(self.label_counts)
        self.num_samples = len(dataset) if num_samples is None else
num_samples

        if self.current_weights is None:
            self.current_weights = self._inverse_frequency_weights()

        self.indices = self._generate_indices()

        super().__init__(
            dataset,
            batch_size=batch_size,
            sampler=None,
            shuffle=False,
            **kwargs
        )

    def _extract_labels(self, dataset: Dataset) -> list:
        return [label for _, label in dataset]


    def _inverse_frequency_weights(self, smooth_factor: float = 1e-2)
-> Dict[int, float]:
        total = sum(self.label_counts.values())
        return {label: (total + smooth_factor) / (count +
```

```
smooth_factor)
                for label, count in self.label_counts.items()}

    def _normalize_weights(self, weights: Dict[int, float]) ->
Dict[int, float]:
        weight_sum = sum(weights.values())
        return {label: w/weight_sum for label, w in weights.items()}

    def _generate_indices(self) -> list:
        normalized_weights =
self._normalize_weights(self.current_weights)
        sample_weights = [normalized_weights[label] for label in
self.labels]

        indices = random.choices(
            range(len(self.dataset)),
            weights=sample_weights,
            k=self.num_samples
        )

        if self.shuffle:
            random.Random(self.epoch).shuffle(indices)

        return indices

    def update_weights(self, new_weights: Dict[int, float]):
        self.current_weights = new_weights
        self.indices = self._generate_indices()

    def __iter__(self):
        img_batch = []
        label_batch = []
        for idx in self.indices:
            img_batch.append(self.dataset[idx][0])
            label_batch.append(self.dataset[idx][1])
            if len(img_batch) == self.batch_size:
                yield torch.tensor(np.array(img_batch)),
torch.tensor(np.array(label_batch))
                img_batch = []
                label_batch = []
        if img_batch:  # Yield remaining samples
            yield torch.tensor(np.array(img_batch)),
torch.tensor(np.array(label_batch))

    def __len__(self):
        return math.ceil(self.num_samples / self.batch_size)

batch_size = 64
# train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
```

```
train_loader = WeightedDataLoader(train_dataset,
batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size,
shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False)

num_epochs = 100

import gc
torch.cuda.empty_cache()
gc.collect()

292

from torch.optim.lr_scheduler import CosineAnnealingLR, LinearLR
from torch.optim import AdamW

import torch
from torch.optim import AdamW
from torch.optim.lr_scheduler import CosineAnnealingLR, LinearLR

optimizer = AdamW(model.parameters(), lr=1e-3)
criterion =
nn.CrossEntropyLoss(weight=1/torch.tensor(class_weights).to(device).fl
oat())
warmup_epochs = 2
total_epochs = 30

scheduler_warmup = LinearLR(
    optimizer,
    start_factor=0.01,
    end_factor=1.0,
    total_iters=warmup_epochs
)

scheduler_cosine = CosineAnnealingLR(
    optimizer,
    T_max=total_epochs - warmup_epochs,
    eta_min=1e-6
)

from torch.optim.lr_scheduler import SequentialLR
scheduler = SequentialLR(
    optimizer,
    schedulers=[scheduler_warmup, scheduler_cosine],
    milestones=[warmup_epochs]
)

num_epochs = total_epochs
```

```python
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0

    loop = tqdm(train_loader, desc=f'Epoch [{epoch+1}/{num_epochs}]')

    for images, labels in loop:
        images = images.to(device)
        labels = labels.to(device)

        outputs = model(images)
        loss = criterion(outputs, labels)


        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        loop.set_postfix(loss=loss.item())
    scheduler.step()

    epoch_loss = running_loss / len(train_loader)
    if epoch % 1 == 0:
        preds = []
        true_val_labels = []
        loss_val = 0.0

        with torch.no_grad():
            for images, labels in val_loader:
                images = images.to(device)
                outputs = model(images)
                _, predicted = torch.max(outputs.data, 1)

                preds.extend(predicted.cpu().numpy())
                true_val_labels.extend(labels.cpu().numpy())
                loss = criterion(outputs, labels.to(device))
                loss_val += loss.item()
        preds = np.array(preds)
        true_val_labels = np.array(true_val_labels)
        acc = (preds == true_val_labels).sum() / len(true_val_labels)
        print(epoch_loss, loss_val / len(val_loader), acc, 'loss/val
loss/val acc on epoch ', epoch+1)
        new_weights = per_class_precision(preds, true_val_labels,
class_names)
        train_loader.update_weights({i : new_weights[i] for i in
range(10)})



    model_path = f'efficient_net.pth'
```

```
    torch.save({
        'epoch': epoch+1,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'loss': epoch_loss,
    }, model_path)
```

```
Epoch [1/100]: 100%|██████████| 200/200 [03:02<00:00,  1.10it/s,
loss=0.094]

0.17837897611781955 0.21407871440052986 0.9198246163482618 loss/val
loss/val acc on epoch  1

Epoch [2/100]:  57%|█████▋    | 115/200 [01:46<01:18,  1.08it/s,
loss=0.0568]

----------------------------------------------------------------------
-----
KeyboardInterrupt                         Traceback (most recent call
last)
/tmp/ipykernel_31/3422429321.py in <cell line: 0>()
     17         optimizer.step()
     18 
---> 19         running_loss += loss.item()
     20         loop.set_postfix(loss=loss.item())
     21     scheduler.step()

KeyboardInterrupt:
```

```
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0

    loop = tqdm(train_loader, desc=f'Epoch [{epoch+1}/{num_epochs}]')

    for images, labels in loop:
        images = images.to(device)
        labels = labels.to(device)

        outputs = model(images)
        loss = criterion(outputs, labels)


        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        loop.set_postfix(loss=loss.item())
    scheduler.step()
```

```
    epoch_loss = running_loss / len(train_loader)
    if epoch % 1 == 0:
        preds = []
        true_val_labels = []
        loss_val = 0.0

        with torch.no_grad():
            for images, labels in val_loader:
                images = images.to(device)
                outputs = model(images)
                _, predicted = torch.max(outputs.data, 1)

                preds.extend(predicted.cpu().numpy())
                true_val_labels.extend(labels.cpu().numpy())
                loss = criterion(outputs, labels.to(device))
                loss_val += loss.item()
        preds = np.array(preds)
        true_val_labels = np.array(true_val_labels)
        acc = (preds == true_val_labels).sum() / len(true_val_labels)
        print(epoch_loss, loss_val / len(val_loader), acc, 'loss/val
loss/val acc on epoch ', epoch+1)

    model_path = f'efficient_net.pth'
    torch.save({
        'epoch': epoch+1,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'loss': epoch_loss,
    }, model_path)
```

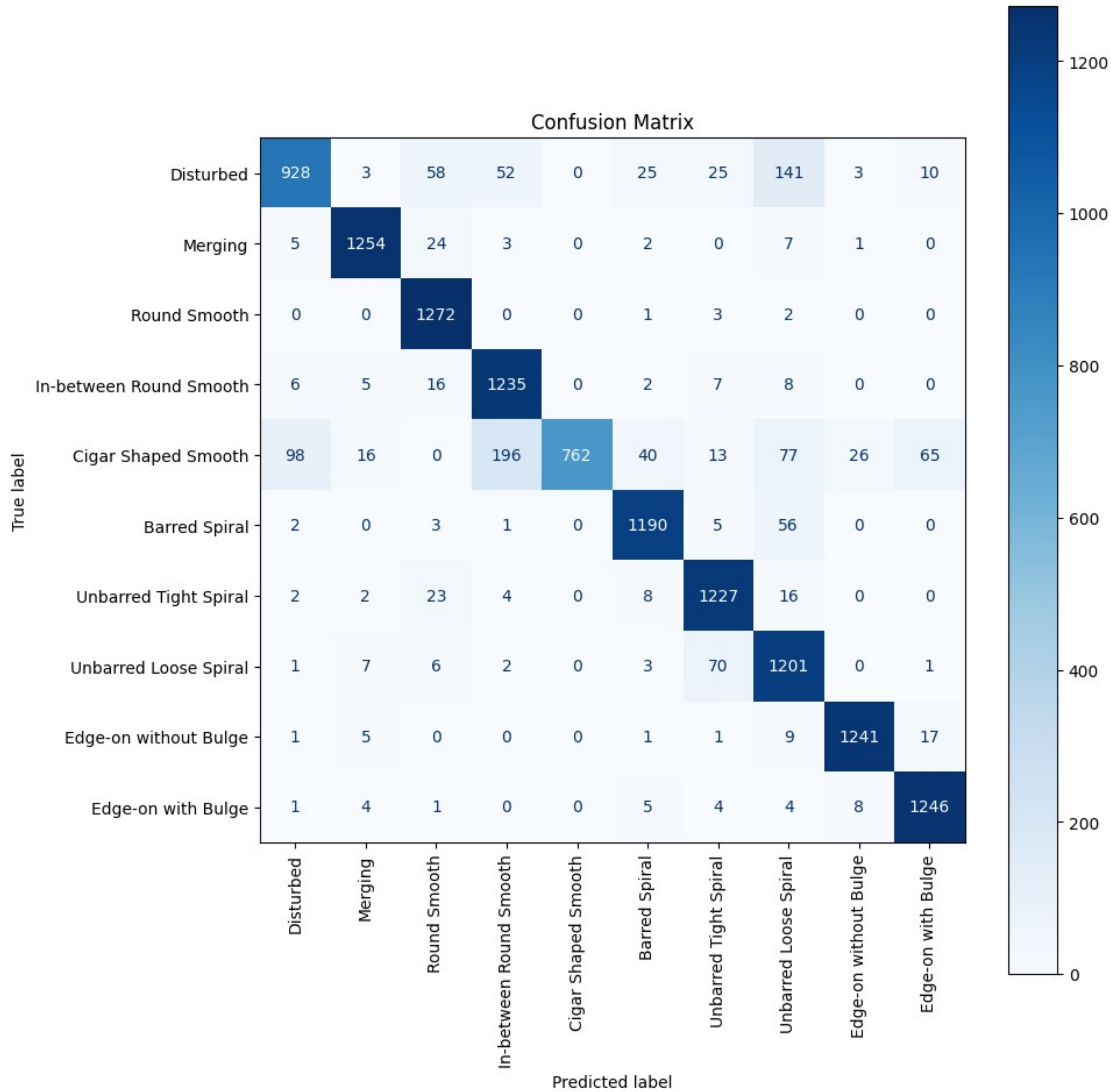
```
Epoch [1/100]: 100%|██████████| 200/200 [02:22<00:00,  1.40it/s,
loss=0.41]

0.18703504202887417 0.34307483792304994 0.8750391481365487 loss/val
loss/val acc on epoch  1

Epoch [2/100]: 100%|██████████| 200/200 [02:22<00:00,  1.41it/s,
loss=0.0761]

0.13494462547823788 0.4086768752336502 0.8578139680551206 loss/val
loss/val acc on epoch  2

Epoch [3/100]:  26%|██▌       | 51/200 [00:37<01:48,  1.37it/s,
loss=0.0652]

---------------------------------------------------------------------
-----
KeyboardInterrupt                         Traceback (most recent call
last)
/tmp/ipykernel_31/3075447770.py in <cell line: 0>()
     19         optimizer.step()
```

```
     20
---> 21         running_loss += loss.item()
     22         loop.set_postfix(loss=loss.item())
     23     scheduler.step()

KeyboardInterrupt:
```

This saved version was trained for god knows how many epochs(I believe around 30-40 overall, not that much, but I hooped between weighted loader and a standard one, slightly changed the lr scheduling strategies). In the end, I don't think that lr needed to be altered, and the main difference came from loader.

```
model_path = 
'/kaggle/input/efficient_net/pytorch/default/1/efficient_net.pth'
model.load_state_dict(torch.load(model_path)['model_state_dict'])

/tmp/ipykernel_31/607918565.py:2: FutureWarning: You are using 
`torch.load` with `weights_only=False` (the current default value), 
which uses the default pickle module implicitly. It is possible to 
construct malicious pickle data which will execute arbitrary code 
during unpickling (See 
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-
models for more details). In a future release, the default value for 
`weights_only` will be flipped to `True`. This limits the functions 
that could be executed during unpickling. Arbitrary objects will no 
longer be allowed to be loaded via this mode unless they are 
explicitly allowlisted by the user via 
`torch.serialization.add_safe_globals`. We recommend you start setting 
`weights_only=True` for any use case where you don't have full control 
of the loaded file. Please open an issue on GitHub for any issues 
related to this experimental feature.
  model.load_state_dict(torch.load(model_path)['model_state_dict'])

<All keys matched successfully>

preds = []
true_test_labels = []

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

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

        preds.extend(predicted.cpu().numpy())
        true_test_labels.extend(labels.cpu().numpy())

test_metrics = evaluate_predictions(preds, true_test_labels, 
class_names)
```

```
Evaluating 12769 predictions against true labels...

--- Evaluation Metrics ---
Accuracy: 0.9050
Weighted Precision: 0.9119
Weighted Recall: 0.9050
Weighted F1-Score: 0.9010
-------------------------
Per-Class Metrics:
Class                          | Precision  | Recall     | F1-Score
| Support
----------------------------------------------------------------------
----------
0: Disturbed                   | 0.8889     | 0.7454     | 0.8108
| 1245
1: Merging                     | 0.9676     | 0.9676     | 0.9676
| 1296
2: Round Smooth                | 0.9066     | 0.9953     | 0.9489
| 1278
3: In-between Round Smooth     | 0.8272     | 0.9656     | 0.8911
| 1279
4: Cigar Shaped Smooth         | 1.0000     | 0.5893     | 0.7416
| 1293
5: Barred Spiral               | 0.9319     | 0.9467     | 0.9392
| 1257
6: Unbarred Tight Spiral       | 0.9055     | 0.9571     | 0.9306
| 1282
7: Unbarred Loose Spiral       | 0.7896     | 0.9303     | 0.8542
| 1291
8: Edge-on without Bulge       | 0.9703     | 0.9733     | 0.9718
| 1275
9: Edge-on with Bulge          | 0.9305     | 0.9788     | 0.9541
| 1273
----------------------------------------------------------------------
----------

Plotting Confusion Matrix...
```



These are the train metrics, and we see that it is close to 0.9 both on train and validation. So, it almost works finally, yet it is to be fine-tuned a bit.

```
train_dataset = Subset(train_ds, train_idx)
train_dataset = TransformSubset(train_dataset,
transform=val_transform)
```

Here I tried to train it a bit without any augmentation. Surprisingly, it got me closer to 0.9 on test(0.93 on validation, as shown below), however personally I don't think this is a good general approach and here I just got lucky(which happens when working with a single dataset only for so long)

```
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0

    loop = tqdm(train_loader, desc=f'Epoch [{epoch+1}/{num_epochs}]')

    for images, labels in loop:
        images = images.to(device)
        labels = labels.to(device)

        outputs = model(images)
        loss = criterion(outputs, labels)


        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        loop.set_postfix(loss=loss.item())
    scheduler.step()

    epoch_loss = running_loss / len(train_loader)
    if epoch % 1 == 0:
        preds = []
        true_val_labels = []
        loss_val = 0.0

        with torch.no_grad():
            for images, labels in val_loader:
                images = images.to(device)
                outputs = model(images)
                _, predicted = torch.max(outputs.data, 1)

                preds.extend(predicted.cpu().numpy())
                true_val_labels.extend(labels.cpu().numpy())
                loss = criterion(outputs, labels.to(device))
                loss_val += loss.item()
        preds = np.array(preds)
        true_val_labels = np.array(true_val_labels)
        acc = (preds == true_val_labels).sum() / len(true_val_labels)
        print(epoch_loss, loss_val / len(val_loader), acc, 'loss/val
loss/val acc on epoch ', epoch+1)


    model_path = f'efficient_net_w.pth'
    torch.save({
        'epoch': epoch+1,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
```

```
        'loss': epoch_loss,
    }, model_path)
```

```
Epoch [1/100]: 100%|██████████| 200/200 [02:17<00:00,  1.45it/s,
loss=0.319]

0.23097235282883047 0.15949416145682335 0.9304729094895083 loss/val
loss/val acc on epoch  1

Epoch [2/100]: 100%|██████████| 200/200 [02:17<00:00,  1.46it/s,
loss=0.191]

0.07155322439502924 0.14627576299011708 0.9398684622611964 loss/val
loss/val acc on epoch  2

Epoch [3/100]:  76%|███████▋  | 153/200 [01:44<00:32,  1.46it/s,
loss=0.0238]
```

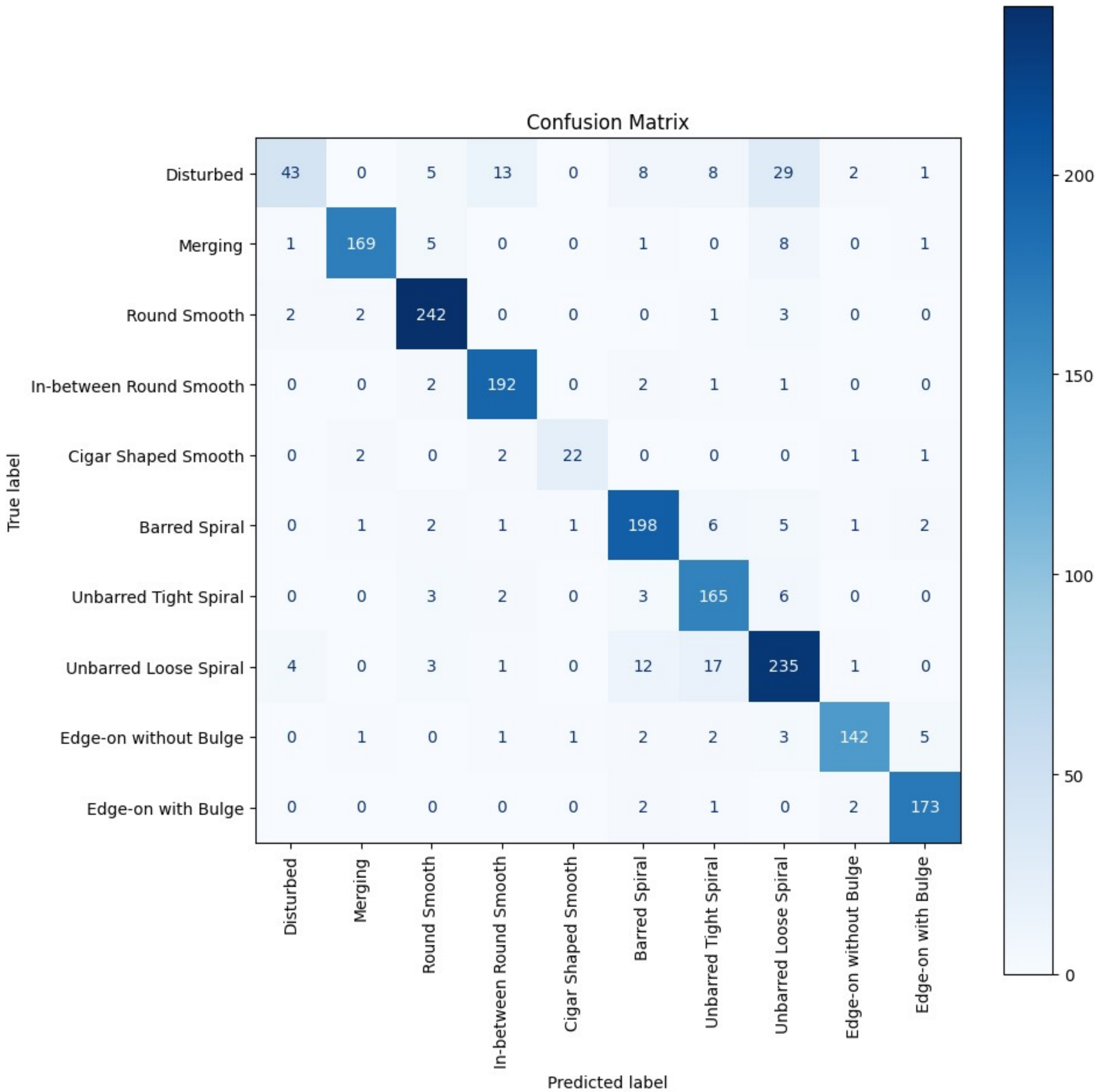
Evaluation

```
model_path =
'/kaggle/input/net_final/pytorch/default/1/efficient_net_w.pth'
model.load_state_dict(torch.load(model_path)['model_state_dict'])
```

```
/tmp/ipykernel_31/2922732412.py:2: FutureWarning: You are using
`torch.load` with `weights_only=False` (the current default value),
which uses the default pickle module implicitly. It is possible to
construct malicious pickle data which will execute arbitrary code
during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-
models for more details). In a future release, the default value for
`weights_only` will be flipped to `True`. This limits the functions
that could be executed during unpickling. Arbitrary objects will no
longer be allowed to be loaded via this mode unless they are
explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control
of the loaded file. Please open an issue on GitHub for any issues
related to this experimental feature.
  model.load_state_dict(torch.load(model_path)['model_state_dict'])

<All keys matched successfully>
```

```
preds = []
true_test_labels = []

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

```
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)

        preds.extend(predicted.cpu().numpy())
        true_test_labels.extend(labels.cpu().numpy())

# true_test_labels = galaxy_dataset['test']['label']
test_metrics = evaluate_predictions(preds, true_test_labels,
class_names)

Evaluating 1774 predictions against true labels...

--- Evaluation Metrics ---
Accuracy: 0.8912
Weighted Precision: 0.8922
Weighted Recall: 0.8912
Weighted F1-Score: 0.8855
-------------------------
Per-Class Metrics:
Class                          | Precision  | Recall     | F1-Score
| Support
----------------------------------------------------------------------
----------
0: Disturbed                   | 0.8600     | 0.3945     | 0.5409
| 109
1: Merging                     | 0.9657     | 0.9135     | 0.9389
| 185
2: Round Smooth                | 0.9237     | 0.9680     | 0.9453
| 250
3: In-between Round Smooth     | 0.9057     | 0.9697     | 0.9366
| 198
4: Cigar Shaped Smooth         | 0.9167     | 0.7857     | 0.8462
| 28
5: Barred Spiral               | 0.8684     | 0.9124     | 0.8899
| 217
6: Unbarred Tight Spiral       | 0.8209     | 0.9218     | 0.8684
| 179
7: Unbarred Loose Spiral       | 0.8103     | 0.8608     | 0.8348
| 273
8: Edge-on without Bulge       | 0.9530     | 0.9045     | 0.9281
| 157
9: Edge-on with Bulge          | 0.9454     | 0.9719     | 0.9584
| 178
----------------------------------------------------------------------
----------

Plotting Confusion Matrix...
```



We are almost there ((((((I didn't get the desired 0.9, but close enough. What concerns me -- test accuracy is 0.89 < 0.93 = validation accuracy, which is not ideal of course.

To summarize, the final model is: EfficientNet, trained for ~35 epochs on train with augmentations and fine-tuned without them for ~5 epochs in total. Augmentations helped a lot, without them I got ~0.97 train accuracy and ~0.72 test accuracy. with ResNet-50. WeightedLoader also made an impact, however it's combination with weighted loss remains questionable. Sampling weights w.r.t inverse class precision is interesting, but showed almost the same results as sampling w.r.t inverse class frequency. I am still interested in how did the authors get results like 0.95 test acc with WaveMix or ConvNext, it seems really challenging from my perspective :)