In [89]:
```
import os
from PIL import Image

import torch
import torchvision
from sklearn.metrics import classification_report, f1_score
```

In [2]:
```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f'Using device: {device}.')
```

```
Using device: cpu.
```

In [3]:
```
# Transformations
transform_rotation = torchvision.transforms.RandomApply([
    torchvision.transforms.RandomRotation(20)
], p=0.2)
```

In [4]:
```
transform_train = torchvision.transforms.Compose([
    torchvision.transforms.Resize(256),
    torchvision.transforms.CenterCrop(224),
    torchvision.transforms.RandomPerspective(distortion_scale=0.1, p=0.2),
    transform_rotation,
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
])
```

In [5]:
```
transform_valid = torchvision.transforms.Compose([
    torchvision.transforms.Resize(256),
    torchvision.transforms.CenterCrop(224),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
])
```

In [23]:
```
# DataLoaders
TRAIN_DATA_DIR = 'data/train'
VALID_DATA_DIR = 'data/eval'
TEST_DATA_DIR = 'data/test'
```

In [25]:
```
BATCH_SIZE = 32

train_data = torchvision.datasets.ImageFolder(TRAIN_DATA_DIR,
                                              transform=transform_train,
                                              is_valid_file=lambda x: x.endswith('.j

valid_data = torchvision.datasets.ImageFolder(VALID_DATA_DIR,
                                              transform=transform_valid,
                                              is_valid_file=lambda x: x.endswith('.j

test_data = torchvision.datasets.ImageFolder(TEST_DATA_DIR,
                                             transform=transform_valid,
                                             is_valid_file=lambda x: x.endswith('.jp
```

In [26]:
```
train_data_loader = torch.utils.data.DataLoader(
    train_data,
```

```
    batch_size=BATCH_SIZE,
    shuffle=True,
    num_workers=0,
)

valid_data_loader = torch.utils.data.DataLoader(
    valid_data,
    batch_size=BATCH_SIZE,
    shuffle=False,
    num_workers=0,
)

test_data_loader = torch.utils.data.DataLoader(
    test_data,
    batch_size=BATCH_SIZE,
    shuffle=False,
    num_workers=0,
)
```

In [27]:

```
# Model

# initialize model
model = torchvision.models.resnet50(pretrained=True).to(device)

# freeze the backbone
for parameter in model.parameters():
    parameter.requires_grad = False


class ModelHead(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, n_classes):
        super(ModelHead, self).__init__()
        self.fc1 = torch.nn.Linear(input_dim, hidden_dim)
        self.relu1 = torch.nn.ReLU()
        self.fc2 = torch.nn.Linear(hidden_dim, hidden_dim // 2)
        self.relu2 = torch.nn.ReLU()
        self.fc3 = torch.nn.Linear(hidden_dim // 2, n_classes)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu1(x)
        x = self.fc2(x)
        x = self.relu2(x)
        x = self.fc3(x)
        return x


model.fc = ModelHead(2048, 1024, 12)
model.fc.to(device)
```

```
C:\Users\sento\anaconda3\lib\site-packages\torchvision\models\_utils.py:208: UserWarn
ing: The parameter 'pretrained' is deprecated since 0.13 and may be removed in the fu
ture, please use 'weights' instead.
  warnings.warn(
C:\Users\sento\anaconda3\lib\site-packages\torchvision\models\_utils.py:223: UserWarn
ing: 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 C:\Users
\sento/.cache\torch\hub\checkpoints\resnet50-0676ba61.pth
100%|██████████| 97.8M/97.8M [00:01<00:00, 73.5MB/s]
```

Out[27]:
```
ModelHead(
  (fc1): Linear(in_features=2048, out_features=1024, bias=True)
  (relu1): ReLU()
  (fc2): Linear(in_features=1024, out_features=512, bias=True)
  (relu2): ReLU()
  (fc3): Linear(in_features=512, out_features=12, bias=True)
)
```

In [28]:
```
# Training
MODEL_SAVE_PATH = 'checkpoints'

LEARNING_RATE = 1e-3
N_EPOCHS = 2

criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING_RATE)


def train(model, n_epochs, criterion, optimizer, train_data_loader, valid_data_loade
          device, model_save_path, logging_interval: int = 50):
    best_valid_f1_score = 0.0
    os.makedirs(model_save_path, exist_ok=True)

    for epoch in range(n_epochs):
        # training step
        model.train()

        for batch_idx, (batch_data, batch_labels) in enumerate(train_data_loader):
            inputs = batch_data.to(device)
            y_true = batch_labels.to(device)

            # zero the parameter gradients
            optimizer.zero_grad()

            # forward + backward + optimizer step
            y_pred = model(inputs)
            loss = criterion(y_pred, y_true)
            loss.backward()
            optimizer.step()

            if (batch_idx + 1) % logging_interval == 0:
                print(f'Epoch: {epoch + 1}\t| Batch: {batch_idx + 1}\t| Loss: {loss}

        # validation step
        model.eval()
        y_true = []
        y_pred = []
        for valid_data, valid_labels in valid_data_loader:
            valid_data = valid_data.to(device)
            valid_labels = valid_labels.to(device)
            with torch.no_grad():
                valid_preds = model(valid_data)
            valid_pred_labels = torch.argmax(valid_preds, dim=1)
            y_true.extend(valid_labels.detach().cpu().numpy())
            y_pred.extend(valid_pred_labels.detach().cpu().numpy())
        valid_f1_score = f1_score(y_true, y_pred, average='macro')

        if valid_f1_score > best_valid_f1_score:
            best_valid_f1_score = valid_f1_score
            torch.save(model.state_dict(),
                       os.path.join(model_save_path, 'best_checkpoint.pth'))
        print(f'Epoch {epoch + 1} F1-score: {valid_f1_score}\t| Best F1-score: {best
        torch.save(model.state_dict(),
                   os.path.join(model_save_path, f'epoch_{epoch + 1}_checkpoint.pth'
```

```
train(model, N_EPOCHS, criterion, optimizer,
      train_data_loader, valid_data_loader,
      device, MODEL_SAVE_PATH)
```

```
Epoch: 1        | Batch: 50     | Loss: 0.6958112716674805
Epoch 1 F1-score: 0.801086213364942     | Best F1-score: 0.801086213364942
Epoch: 2        | Batch: 50     | Loss: 0.7798282504081726
Epoch 2 F1-score: 0.8374614298178288    | Best F1-score: 0.8374614298178288
```

In [29]:

```
# Testing
model.load_state_dict(torch.load(os.path.join(MODEL_SAVE_PATH, 'best_checkpoint.pth'
model.eval()

y_true = []
y_pred = []
for test_data, test_labels in test_data_loader:
    test_data = test_data.to(device)
    test_labels = test_labels.to(device)
    with torch.no_grad():
        test_preds = model(test_data)
    test_pred_labels = torch.argmax(test_preds, dim=1)
    y_true.extend(test_labels.detach().cpu().numpy())
    y_pred.extend(test_pred_labels.detach().cpu().numpy())

print(classification_report(y_true, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.97      0.78      0.86        36
           1       0.66      0.90      0.76        30
           2       0.88      0.85      0.87        34
           3       0.95      1.00      0.98        20
           4       0.86      0.60      0.71        30
           5       1.00      0.73      0.84        37
           6       1.00      0.85      0.92        34
           7       0.93      1.00      0.96        26
           8       0.72      0.88      0.79        24
           9       0.64      0.90      0.75        30
          10       0.95      0.95      0.95        22
          11       0.95      0.97      0.96        37

    accuracy                           0.86       360
   macro avg       0.88      0.87      0.86       360
weighted avg       0.88      0.86      0.86       360
```

In [91]:

```
image_path = 'data/cat_12.jpg'
image = Image.open(image_path)
transform = torchvision.transforms.Compose([
    torchvision.transforms.Resize(256),
    torchvision.transforms.CenterCrop(224),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
])
preprocessed_image = transform(image).unsqueeze(0).to(device)
```

In [81]:

```
with torch.no_grad():
    predictions = model(preprocessed_image)

predicted_class = torch.argmax(predictions).item()
```

In [82]:

```
class_labels = ['Abyssinian', 'Bengal', 'Birman', 'Bengal', 'Bombay', 'British Short
predicted_label = class_labels[predicted_class]
print(f'Predicted class: {predicted_label}')
```

```
Predicted class: Sphynx
```