Image classification with Convolutional Neural Networks

Image classification is one of the applications that first showed the potential of deep learning.

Convolutional Neural Networks (CNN) have been found to be especially suitable for this task, and many advancements in deep learning (batch normalization, residual connections, ...) were first proposed in this domain.

Pet classification

In this exercise, we will address a simple binary classification problem in which we want to distinguish between images of cats and dogs.

We will use the Oxford Pet Dataset, which includes 7349 images of different breeds of cats and dogs.

Setup and Data

We will use a preprocessed version of the dataset to speed up the exercise. All images have been resized to 160×160 pixels. The training set includes 6349 images (2047 cats, 4302 dogs) and 1000 test images (324 cats, 676 dogs). Files have been organized in folders as follows:

- `/train`
  - `train/cat`
  - `train/dog`
- `/test`
  - `test/cat`
  - `test/dog`

We will use the `torchvision.dataset.ImageFolder` class which is perfect for dealing with datasets of that structure.

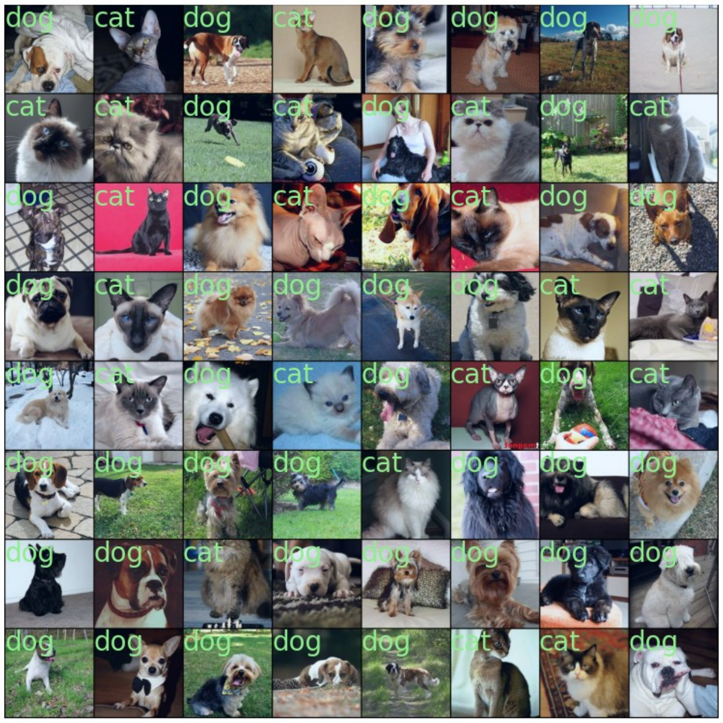
```
import torch
import torchvision
import matplotlib.pyplot as plt
import google.colab
import zipfile

google.colab.drive.mount('/content/drive')
PROJECT_DIR = "/content/drive/MyDrive/"
zip_ref = zipfile.ZipFile(PROJECT_DIR + "pet-classification.zip", "r")
zip_ref.extractall(".")
zip_ref.close()
print("OK")
```

```
Drive already mounted at /content/drive; to attempt to forcibly
remount, call drive.mount("/content/drive", force_remount=True).
OK
```

Let's create two `ImageFolder` datasets (train and test) with the corresponding data loaders.

```
MEAN_RGB = [0.4849, 0.4495, 0.3966]
STDDEV_RGB = [0.2631, 0.2584, 0.2652]


tr = torchvision.transforms.Compose([
    torchvision.transforms.RandomHorizontalFlip(),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize(MEAN_RGB, STDDEV_RGB)
])

train_set = torchvision.datasets.ImageFolder(
    "pet-classification/train", transform=tr)

train_loader = torch.utils.data.DataLoader(
    train_set, batch_size=64, shuffle=True, num_workers=2,
    drop_last=True)


tr = torchvision.transforms.Compose([
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize(MEAN_RGB, STDDEV_RGB)
])

test_set = torchvision.datasets.ImageFolder(
    "pet-classification/test", transform=tr)

test_loader = torch.utils.data.DataLoader(
    test_set, batch_size=64, shuffle=True, num_workers=2,
    drop_last=False)


def show_batch(images, labels, predictions):
    plt.rcParams["figure.dpi"] = 200
    grid = torchvision.utils.make_grid(images, normalize=True)
    plt.imshow(grid.permute(1, 2, 0))
    for i in range(len(labels)):
       row = i // 8
       col = i % 8
       x = col * 162
       y = 40 + row * 162
       color = "lightgreen" if labels[i] == predictions[i] else "red"
       txt = train_set.classes[labels[i]]
       plt.text(x, y, txt, color=color)
    plt.axis("off")
    plt.show()

images, labels = next(iter(train_loader))
print(images.shape, images.dtype)
print(labels.shape, labels.dtype)
show_batch(images, labels, labels)
```

```
torch.Size([64, 3, 160, 160]) torch.float32
torch.Size([64]) torch.int64
```



dog cat dog cat dog dog dog dog
cat cat dog cat dog cat dog cat
dog cat dog cat dog cat dog dog
dog cat dog dog dog dog cat cat
dog cat dog cat dog cat dog cat
dog dog dog dog cat dog dog dog
dog dog cat dog dog dog dog dog
dog dog dog dog dog cat cat dog

Convolutional Networks for image classification

Convolutional Neural Networks are a very effective model for image classification. The network is divided in two parts: a feature extractor and a classifier.

The feature extractor is just a sequence of *convolutional blocks* including two convolutions, normalization layers, and ReLUs. The second convolution in each block is strided, to reduce the spatial dimension of the resulting feature map.

The classifier computes the average feature vector and applies a linear layer to compute the score, and a sigmoid to compute the probability estimate. Normally, you would need one output score for each class. For binary classification one value is enough.

```
class CNN(torch.nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.feature = torch.nn.Sequential(
            self.make_block(3, 16),
            self.make_block(16, 32),
            self.make_block(32, 64),
            self.make_block(64, 128),
            self.make_block(128, 256)
        )
        self.classifier = torch.nn.Sequential(
            torch.nn.AdaptiveAvgPool2d(1),
            torch.nn.Flatten(),
            torch.nn.Linear(256, 1),
            torch.nn.Sigmoid()
        )

    def forward(self, x):
        x = self.feature(x)
        x = self.classifier(x)
        return x

    def make_block(self, in_channels, out_channels):
        block = torch.nn.Sequential(
            torch.nn.Conv2d(in_channels, out_channels, 3, padding=1),
            torch.nn.BatchNorm2d(out_channels),
            torch.nn.ReLU(),
            torch.nn.Conv2d(out_channels, out_channels, 3, stride=2, 
padding=1),
            torch.nn.BatchNorm2d(out_channels),
            torch.nn.ReLU(),
        )
        return block


net = CNN()
x = torch.rand(1, 3, 160, 160)
y = net(x)
print(x.shape, y.shape)

params = sum(p.numel() for p in net.parameters())
print(params, "parameters")
```

```
torch.Size([1, 3, 160, 160]) torch.Size([1, 1])
1181009 parameters
```

Training loop

A loss function suitable for binary classification is the Binary Cross Entropy, defined as the average over a batch of the following expression:

$$L_{\mathrm{BCE}} = -y \log p - (1-y) \log (1-p),$$

where $p \in (0,1)$ is the probability estimate computed by the CNN, and $y \in \{0,1\}$ is the actual class label.

We will use the Adam optimization method to minimize the BCE loss. This method is similar to Stochastic Gradient Descent but less sensitive to the choice of the hyperparameters.

```python
EPOCHS = 10
LR = 0.001
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
print("Device:", DEVICE)

net.to(DEVICE)
optimizer = torch.optim.Adam(net.parameters(), LR)
loss_fun = torch.nn.BCELoss().to(DEVICE)

net.train()
steps = 0
for epoch in range(EPOCHS):
    for images, labels in train_loader:
        images = images.to(DEVICE)
        labels = labels.to(DEVICE)
        optimizer.zero_grad()
        probs = net(images)
        loss = loss_fun(probs.squeeze(1), labels.float())
        loss.backward()
        optimizer.step()
        steps += 1
        if steps % 50 == 0:
            predictions = (probs > 0.5).long()
            accuracy = 100 * (predictions.squeeze(1) ==
labels).float().mean()
            print(f"{steps} [{epoch}]  Loss: {loss.item():.4f}
Accuracy: {accuracy:.1f}%")
    data = {
        "classifier": net.state_dict(),
        "optimizer": optimizer.state_dict()
    }
    torch.save(data, PROJECT_DIR + "classifier.pt")
```

```
Device: cuda
50 [0]  Loss: 0.6062  Accuracy: 68.8%
100 [1]  Loss: 0.5654  Accuracy: 71.9%
150 [1]  Loss: 0.5488  Accuracy: 73.4%
200 [2]  Loss: 0.5428  Accuracy: 75.0%
250 [2]  Loss: 0.5393  Accuracy: 70.3%
300 [3]  Loss: 0.5847  Accuracy: 68.8%
350 [3]  Loss: 0.6552  Accuracy: 71.9%
400 [4]  Loss: 0.5289  Accuracy: 68.8%
450 [4]  Loss: 0.5992  Accuracy: 71.9%
500 [5]  Loss: 0.4508  Accuracy: 78.1%
550 [5]  Loss: 0.4602  Accuracy: 76.6%
600 [6]  Loss: 0.3225  Accuracy: 84.4%
650 [6]  Loss: 0.3260  Accuracy: 82.8%
```

```
700 [7]  Loss: 0.4271  Accuracy: 85.9%
750 [7]  Loss: 0.4286  Accuracy: 82.8%
800 [8]  Loss: 0.3260  Accuracy: 84.4%
850 [8]  Loss: 0.3814  Accuracy: 81.2%
900 [9]  Loss: 0.2678  Accuracy: 87.5%
950 [9]  Loss: 0.1655  Accuracy: 96.9%
```

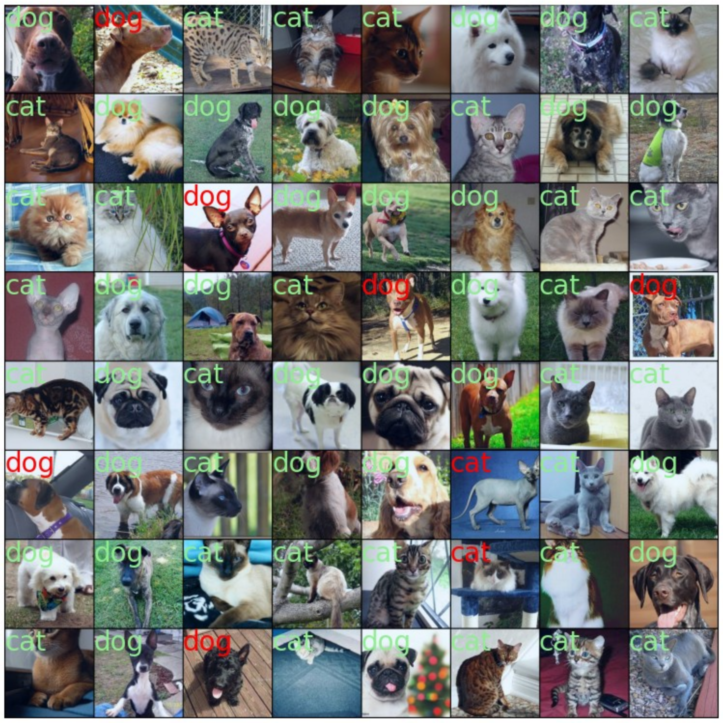
Evaluation

For this problem, accuracy (i.e., fraction of correct predictions) is a reasonable performance measure.

```python
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
print("Device:", DEVICE)
data = torch.load(PROJECT_DIR + "classifier.pt", map_location=DEVICE)
net.to(DEVICE)
net.load_state_dict(data["classifier"])
# optimizer.load_state_dict(data["optimizer"])
```

```
Device: cuda

<All keys matched successfully>
```

```python
correct = 0
tot = 0
net.eval()
first = True
for images, labels in test_loader:
    images = images.to(DEVICE)
    labels = labels.to(DEVICE)
    with torch.no_grad():
        probs = net(images)
    predictions = (probs > 0.5).long()
    correct += (predictions.squeeze(1) == labels).long().sum().item()
    tot += labels.shape[0]
    if first:
        first = False
        show_batch(images.cpu(), labels.cpu(), 
predictions.squeeze(1).cpu())

accuracy = 100 * correct / tot
print(f"Accuracy: {accuracy:.1f}%")
```


dog dog cat cat cat dog dog cat
cat dog dog dog dog cat dog dog
cat cat dog dog dog dog cat cat
cat dog dog cat dog dog cat dog
cat dog cat dog dog dog cat cat
dog dog cat dog dog cat cat dog
dog dog cat cat cat cat cat dog
cat dog dog cat dog cat cat cat

```
Accuracy: 85.2%
```

Questions

In the report, answer to the following questions:

- Identify badly classified test images and group them according to the most likely cause for the errors.

- Pick a handful of images of other similar animals (lions, wolves, foxes, jaguars...) and comment the predictions made by the model (it is easier if you manually scale the images to 160×160 pixels).

Extensions

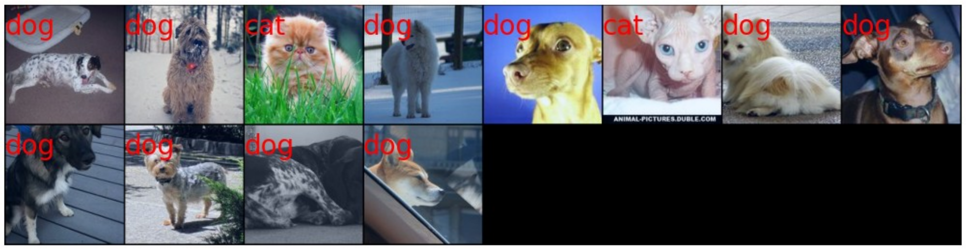
The model presented here are by no means optimal. Try to identify the best combination of parameters, also by adjusting the number of features, blocks etc.

Other possible investigations:

- Identify how frequently each class is misclassified as the other (i.e., build a confusion matrix.
- Find the worst predictions made by the model: can you guess why these mistakes were made?
- Try data augmentation, such as random crop, described here.
- Try introducing dropout layers to improve generalization.

1. identify badly classified test images and group them according to the most likely cause for the errors.

```
import numpy as np

net.eval()
misclassified_examples = []
for images, labels in test_loader:
    images = images.to(DEVICE)
    labels = labels.to(DEVICE)
    with torch.no_grad():
        probs = net(images)
    predictions = (probs > 0.5).long()
    correct += (predictions.squeeze(1) == labels).long().sum().item()
    tot += labels.shape[0]

    misclassified_mask = predictions.squeeze(1) != labels
    misclassified_images = images[misclassified_mask].cpu()
    misclassified_labels = labels[misclassified_mask].cpu()
    misclassified_predictions = predictions.squeeze(1)
[misclassified_mask].cpu()

    misclassified_examples.append((misclassified_images,
misclassified_labels, misclassified_predictions))

accuracy = 100 * correct / tot
print(f"Accuracy: {accuracy:.1f}%")

# Stampa solo gli esempi classificati erroneamente
for misclassified_batch in misclassified_examples:
    show_batch(*misclassified_batch)


Accuracy: 85.2%
```



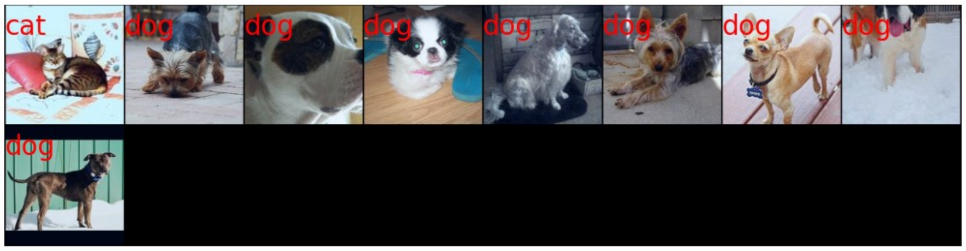
cat
dog
dog
dog
dog
dog
dog
dog
dog

dog
dog
dog
cat
dog
cat

cat
dog
dog
dog
dog
cat
dog
cat
dog

dog
dog
cat
cat
cat
dog
cat
dog
dog
dog
dog

dog
dog
dog
dog
dog
cat
dog
cat
dog
dog


cat
dog
dog
dog
dog
dog
dog
dog
dog

dog
dog
cat
dog
dog
cat
dog

dog
dog
dog
cat
cat
dog
dog

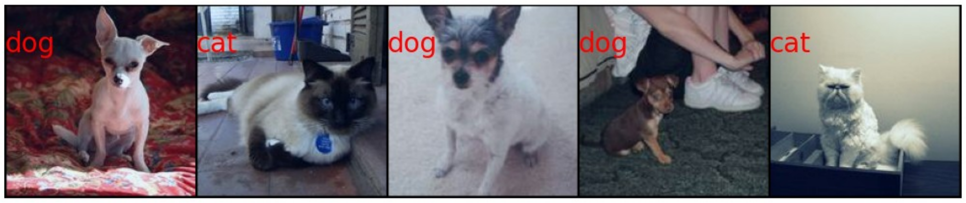
cat
cat
cat
dog
dog
dog
dog
dog

dog
cat
dog
cat
cat
dog
dog
dog
cat
cat
cat
cat

dog
dog
dog
dog
dog
cat
cat
cat
dog
dog
dog

dog
cat
dog
cat
dog
dog
dog

cat
dog
dog
dog
dog
dog
dog
dog
cat
dog
dog
dog
dog
dog



```python
from sklearn.metrics import confusion_matrix

net.eval()
misclassified_examples = []
true_labels_all = []
predicted_labels_all = []

for images, labels in test_loader:
    images = images.to(DEVICE)
    labels = labels.to(DEVICE)
    with torch.no_grad():
        probs = net(images)
    predictions = (probs > 0.5).long()
    correct += (predictions.squeeze(1) == labels).long().sum().item()
    tot += labels.shape[0]

    true_labels_all.extend(labels.cpu().numpy())
    predicted_labels_all.extend(predictions.squeeze(1).cpu().numpy())

accuracy = 100 * correct / tot
print(f"Accuracy: {accuracy:.1f}%")

# Calcola la matrice di confusione
conf_matrix = confusion_matrix(true_labels_all, predicted_labels_all)

# Identifica le categorie di errore più comuni
most_common_errors = []
for i in range(conf_matrix.shape[0]):
    for j in range(conf_matrix.shape[1]):
        if i != j and conf_matrix[i][j] > 0:
            most_common_errors.append((i, j, conf_matrix[i][j]))
```

```
# Stampa le categorie di errore più comuni
# Stampa le categorie di errore più comuni
for true_label, predicted_label, count in most_common_errors:
    print(f"True label: {true_label}, Predicted label: 
{predicted_label}, Count: {count}")


Accuracy: 85.2%
True label: 0, Predicted label: 1, Count: 42
True label: 1, Predicted label: 0, Count: 106
```

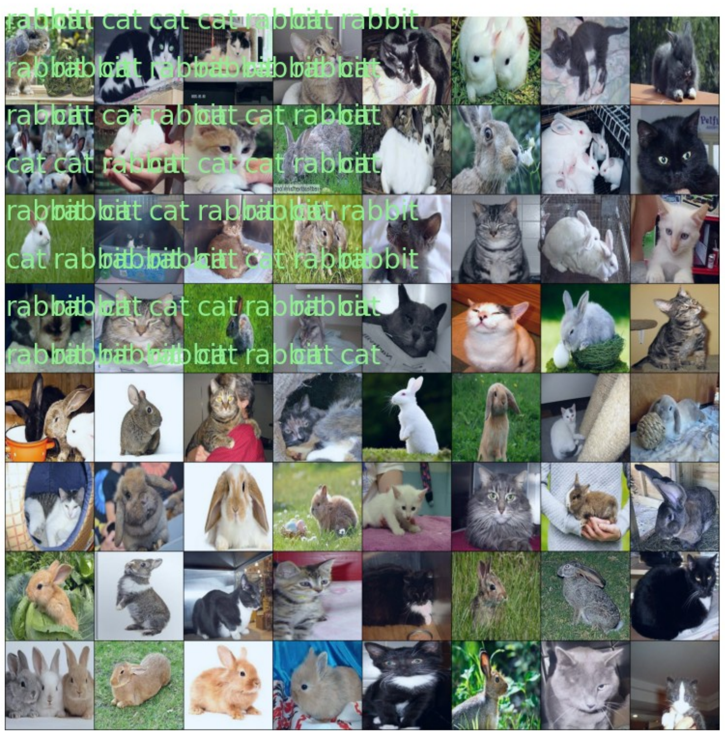
1. Pick a handful of images of other similar animals (lions, wolves, foxes, jaguars...) and comment the predictions made by the model (it is easier if you manually scale the images to 160×160 pixels).

```
import torch
import torchvision
import matplotlib.pyplot as plt
import google.colab
import zipfile

google.colab.drive.mount('/content/drive')
PROJECT_DIR = "/content/drive/MyDrive/"
zip_ref = zipfile.ZipFile(PROJECT_DIR + "cat-rabbit.zip", "r")
zip_ref.extractall(".")
zip_ref.close()
print("OK")

Drive already mounted at /content/drive; to attempt to forcibly 
remount, call drive.mount("/content/drive", force_remount=True).
OK

from PIL import Image
import os
import zipfile
import io
import shutil

import os
import zipfile
from PIL import Image
import io
import shutil

def resize_images_in_zip(input_zip, new_size):
    with zipfile.ZipFile(input_zip, 'r') as zip_ref:
        temp_dir = "temp_extracted_images"
        zip_ref.extractall(temp_dir)

        for filename in os.listdir(temp_dir):
            if filename.endswith(('.jpg', '.jpeg', '.png', '.gif')):
                img_path = os.path.join(temp_dir, filename)
                img = Image.open(img_path)
                resized_img = img.resize(new_size, Image.ANTIALIAS)
                resized_img.save(img_path)

        new_zip_data = io.BytesIO()
        with zipfile.ZipFile(new_zip_data, 'w') as new_zipfile:
            for filename in os.listdir(temp_dir):
                img_path = os.path.join(temp_dir, filename)
                new_zipfile.write(img_path, arcname=filename)

        with open(input_zip, 'wb') as original_zipfile:
            original_zipfile.write(new_zip_data.getvalue())

        shutil.rmtree(temp_dir)

# Example usage:
input_zipfile = "/content/drive/MyDrive/cat-rabbit.zip"
new_size = (160, 160)  # Set the new width and height here

resize_images_in_zip(input_zipfile, new_size)

MEAN_RGB = [0.4849, 0.4495, 0.3966]
STDDEV_RGB = [0.2631, 0.2584, 0.2652]


tr = torchvision.transforms.Compose([
    torchvision.transforms.RandomHorizontalFlip(),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize(MEAN_RGB, STDDEV_RGB)
])

train_set = torchvision.datasets.ImageFolder(
    "train-cat-rabbit", transform=tr)

train_loader = torch.utils.data.DataLoader(
    train_set, batch_size=64, shuffle=True, num_workers=2,
    drop_last=True)


tr = torchvision.transforms.Compose([
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize(MEAN_RGB, STDDEV_RGB)
])

test_set = torchvision.datasets.ImageFolder(
    "test-images", transform=tr)

test_loader = torch.utils.data.DataLoader(
    test_set, batch_size=64, shuffle=True, num_workers=2,
    drop_last=False)
```

```
def show_batch(images, labels, predictions):
    plt.rcParams["figure.dpi"] = 200
    grid = torchvision.utils.make_grid(images, normalize=True)
    plt.imshow(grid.permute(1, 2, 0))
    for i in range(len(labels)):
       row = i // 8
       col = i % 8
       x = col * 162
       y = 40 + row * 162
       color = "lightgreen" if labels[i] == predictions[i] else "red"
       txt = train_set.classes[labels[i]]
       plt.text(x, y, txt, color=color)
    plt.axis("off")
    plt.show()
```

```
images, labels = next(iter(train_loader))
print(images.shape, images.dtype)
print(labels.shape, labels.dtype)
show_batch(images, labels, labels)
```

```
torch.Size([64, 3, 300, 300]) torch.float32
torch.Size([64]) torch.int64
```



```
class CNN(torch.nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.feature = torch.nn.Sequential(
            self.make_block(3, 16),
            self.make_block(16, 32),
            self.make_block(32, 64),
            self.make_block(64, 128),
            self.make_block(128, 256)
        )
        self.classifier = torch.nn.Sequential(
            torch.nn.AdaptiveAvgPool2d(1),
            torch.nn.Flatten(),
            torch.nn.Linear(256, 1),
            torch.nn.Sigmoid()
        )

    def forward(self, x):
        x = self.feature(x)
        x = self.classifier(x)
        return x

    def make_block(self, in_channels, out_channels):
        block = torch.nn.Sequential(
            torch.nn.Conv2d(in_channels, out_channels, 3, padding=1),
            torch.nn.BatchNorm2d(out_channels),
            torch.nn.ReLU(),
            torch.nn.Conv2d(out_channels, out_channels, 3, stride=2,
padding=1),
            torch.nn.BatchNorm2d(out_channels),
            torch.nn.ReLU(),
        )
        return block


net = CNN()
x = torch.rand(1, 3, 160, 160)
y = net(x)
print(x.shape, y.shape)

params = sum(p.numel() for p in net.parameters())
print(params, "parameters")
```

```
torch.Size([1, 3, 160, 160]) torch.Size([1, 1])
1181009 parameters
```

```
EPOCHS = 2
LR = 0.001
#DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
DEVICE = "cpu"
```

```
print("Device:", DEVICE)

net.to(DEVICE)
optimizer = torch.optim.Adam(net.parameters(), LR)
loss_fun = torch.nn.BCELoss().to(DEVICE)

net.train()
steps = 0
for epoch in range(EPOCHS):
    for images, labels in train_loader:
        images = images.to(DEVICE)
        labels = labels.to(DEVICE)
        optimizer.zero_grad()
        probs = net(images)
        loss = loss_fun(probs.squeeze(1), labels.float())
        loss.backward()
        optimizer.step()
        steps += 1
        if steps % 50 == 0:
            predictions = (probs > 0.5).long()
            accuracy = 100 * (predictions.squeeze(1) ==
labels).float().mean()
            print(f"{steps} [{epoch}]  Loss: {loss.item():.4f}
Accuracy: {accuracy:.1f}%")
    data = {
        "classifier": net.state_dict(),
        "optimizer": optimizer.state_dict()
    }
    torch.save(data, PROJECT_DIR + "classifier.pt")
```

```
Device: cpu
50 [1]  Loss: 0.4326  Accuracy: 82.8%
```

```
wrong_images = []
wrong_labels = []
correct_labels = []

with torch.no_grad():
    for images, labels in train_loader:
        images = images.to(DEVICE)
        labels = labels.to(DEVICE)
        probs = net(images)
        predictions = (probs > 0.5).long()
        wrong_indices = (predictions.squeeze(1) !=
labels).nonzero().squeeze().tolist()
        for idx in wrong_indices:
            wrong_images.append(images[idx].cpu())  # Move images back
to CPU for visualization
```

```
            wrong_labels.append(predictions[idx].item())
            correct_labels.append(labels[idx].item())


import matplotlib.pyplot as plt
import numpy as np

# Define a function to visualize the wrongly classified images
def visualize_wrong_images(wrong_images, wrong_labels, correct_labels,
num_images=5):
    plt.figure(figsize=(12, 6))
    for i in range(num_images):
        plt.subplot(1, num_images, i+1)
        image = wrong_images[i].permute(1, 2, 0).numpy()
        image = np.clip(image, 0, 1)  # Clip values to [0, 1]
        plt.imshow(image)
        plt.title(f"Pred: {wrong_labels[i]}, Correct:
{correct_labels[i]}")
        plt.axis('off')
    plt.show()

# Visualize the first 5 wrongly classified images
visualize_wrong_images(wrong_images, wrong_labels, correct_labels,
num_images=5)

#version without labels
import matplotlib.pyplot as plt
import numpy as np

# Define a function to visualize the wrongly classified images
def visualize_wrong_images(wrong_images, num_images=5):
    plt.figure(figsize=(12, 6))
    for i in range(num_images):
        plt.subplot(1, num_images, i+1)
        image = wrong_images[i].permute(1, 2, 0).numpy()  # Assuming
images are in CHW format
        image = np.clip(image, 0, 1)  # Clip values to [0, 1]
        plt.imshow(image)
        plt.axis('off')
    plt.show()

visualize_wrong_images(wrong_images, num_images=10)
```