DeepLearning.AI

Introduction to TorchVision

Working with images using TorchVision

In Module 2 you'll dive into:

Optimization

Images

Text

Efficiency

Deep Learning for computer vision


Image loading


Data augmentation


Normalization


Batching


Model definition


Visual debugging

TorchVision offers:

Consistency

Integrates with PyTorch Dataset and DataLoader

TorchVision offers:

Consistency

Integrates with PyTorch Dataset and DataLoader

Efficiency

Transforms optimized in C or TorchScript

TorchVision offers:

Consistency

Integrates with PyTorch Dataset and DataLoader

Efficiency

Transforms optimized in C or TorchScript

Reliability

Standardized code for reduced bugs

TorchVision tools

Transforms

Utility functions

Datasets

Models

TorchVision tools

Transforms

Utility functions

Datasets

Models

Transforms

- Resizing
- Cropping
- Rotating
- Flipping
- Adjusting brightness/contrast
- Converting to tensors
- Normalizing pixel values
- Custom

TorchVision tools

Transforms

Utility functions

Datasets

Models

Utility functions

`decode_image()`

`make_grid()`

`save_image()`

`draw_bounding_boxes()`

`draw_segmentation_masks()`

Utility functions

`decode_image()`

`make_grid()`

`save_image()`

`draw_bounding_boxes()`

`draw_segmentation_masks()`

Loading images: `decode_image()`

```
from PIL import Image
from torchvision.io import decode_image

# Load the image
image = decode_image('./images/apples.jpg')

print(f"Image tensor dimensions: {image.shape}")
print(f"Image tensor dtype: {image.dtype}")
```

Loading images: `decode_image()`

Output:

```
Image tensor dimensions: torch.Size([3, 2048, 2048])
Image tensor dtype: torch.uint8
```

Loading images: `decode_image()`

Output:

```
Image tensor dimensions: torch.Size([3, 2048, 2048])
Image tensor dtype: torch.uint8
```

channels

width

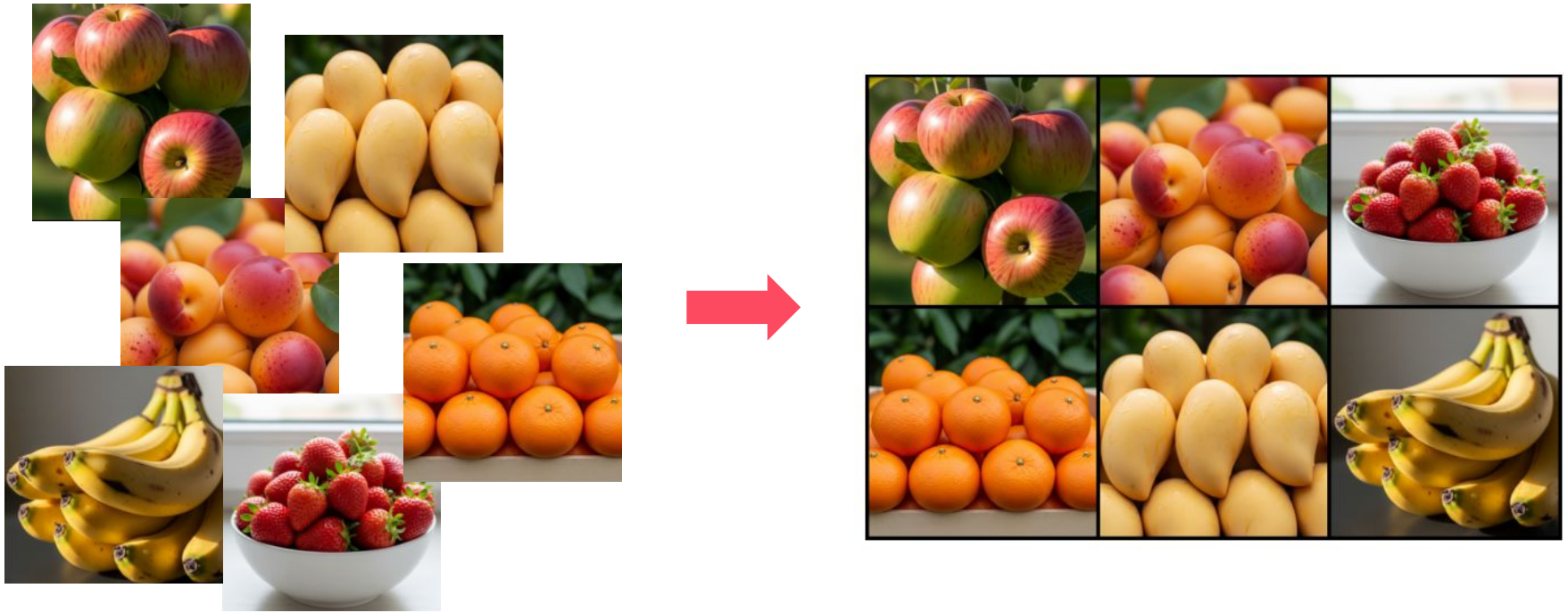
height

Loading images: `decode_image()`

Output:

```
Image tensor dimensions: torch.Size([3, 2048, 2048])
Image tensor dtype: torch.uint8
```

Displaying multiple images: `make_grid()`



Displaying multiple images: `make_grid()`

```
import torchvision.utils as vutils

# Create a batch of images (./images/ contain only 6 images)
images_tensor = helper_utils.load_images("./images/")

# Make a grid from the loaded images (2 rows of 3 for 6 images)
grid = vutils.make_grid(tensor=images_tensor, nrow=3, padding=5, normalize=True)
```

Displaying multiple images: `make_grid()`

```
import torchvision.utils as vutils

# Create a batch of images (./images/ contain only 6 images)
images_tensor = helper_utils.load_images("./images/")

# Make a grid from the loaded images (2 rows of 3 for 6 images)
grid = vutils.make_grid(tensor=images_tensor, nrow=3, padding=5, normalize=True)
```

Displaying multiple images: `make_grid()`

```
import torchvision.utils as vutils

# Create a batch of images (./images/ contain only 6 images)
images_tensor = helper_utils.load_images("./images/")

# Make a grid from the loaded images (2 rows of 3 for 6 images)
grid = vutils.make_grid(tensor=images_tensor, nrow=3, padding=5, normalize=True)
```

Displaying multiple images: `make_grid()`

```
import torchvision.utils as vutils

# Create a batch of images (./images/ contain only 6 images)
images_tensor = helper_utils.load_images("./images/")

# Make a grid from the loaded images (2 rows of 3 for 6 images)
grid = vutils.make_grid(tensor=images_tensor, nrow=3, padding=5, normalize=True)
```

Displaying multiple images: `make_grid()`

```
import torchvision.utils as vutils

# Create a batch of images (./images/ contain only 6 images)
images_tensor = helper_utils.load_images("./images/")

# Make a grid from the loaded images (2 rows of 3 for 6 images)
grid = vutils.make_grid(tensor=images_tensor, nrow=3, padding=5, normalize=True)
```

Saving images: save_image()

```
# Save the grid as a PNG image
vutils.save_image(tensor=grid, fp="./fruits_grid.png")
```

TorchVision tools

Transforms

Utility functions

Datasets

Models

Built-in datasets

- **Image classification**: CIFAR, MNIST, Fashion-MNIST, SVHN
- **Detection and segmentation:** COCO, VOC, SVHN
- **Video**: Kinetics

Fashion MNIST

CIFAR

COCO

TorchVision tools

Transforms

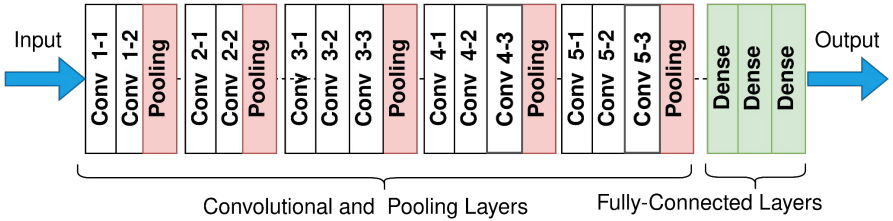
Utility functions

Datasets

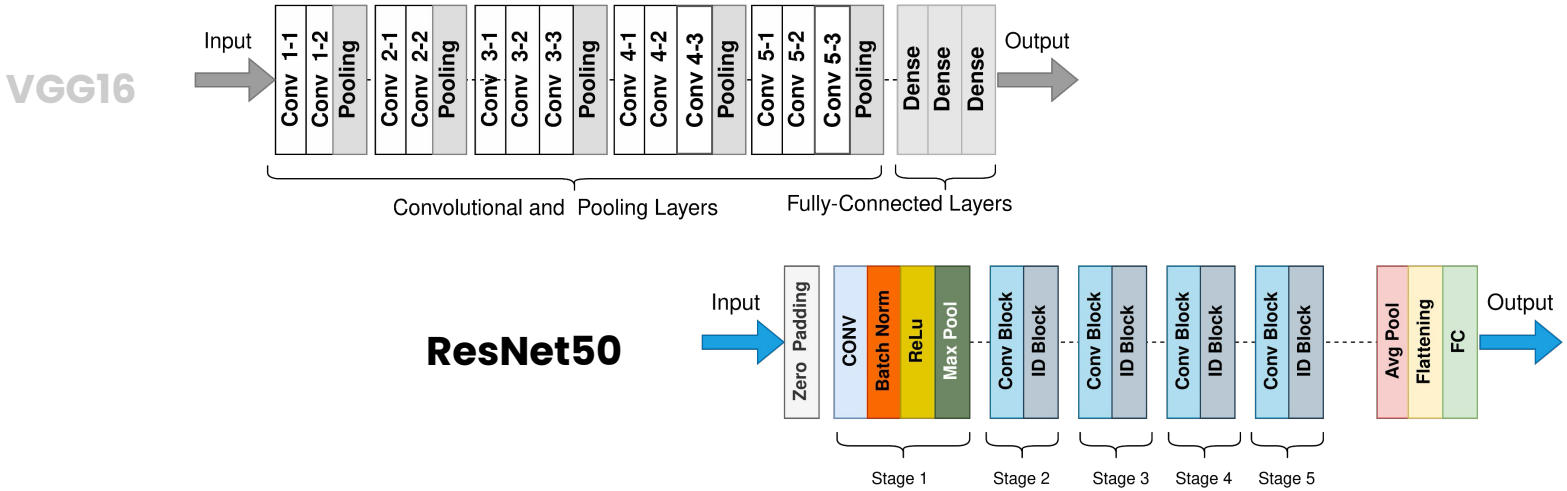
Models

Computer vision models

VGG16

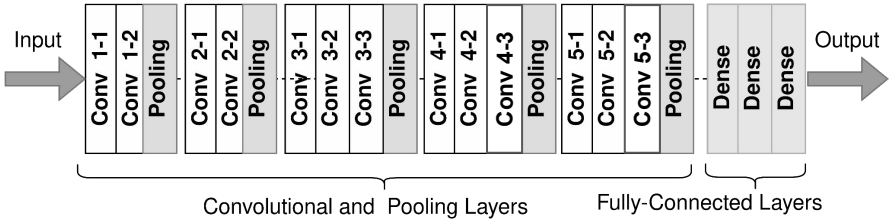


Computer vision models

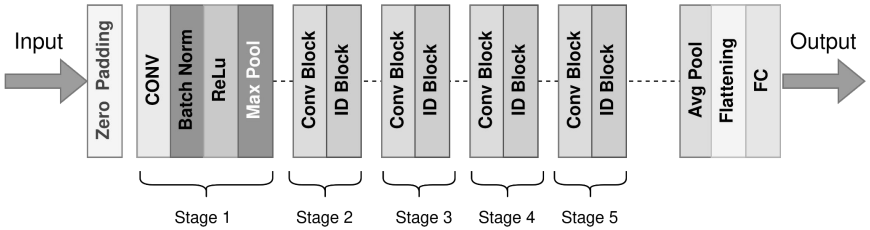


Computer vision models

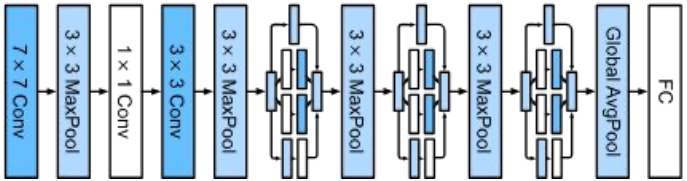
VGG16



ResNet50



GoogLeNet



Computer vision models

Load popular models with pretrained weights

```
model = torchvision.models.resnet50(pretrained=True)
```

DeepLearning.AI

TorchVision Transforms

Working with images using TorchVision

TorchVision tools

Transforms

Utility functions

Datasets

Models

TorchVision tools

Transforms

Utility functions

Datasets

Models

TorchVision transforms

`ToTensor()`

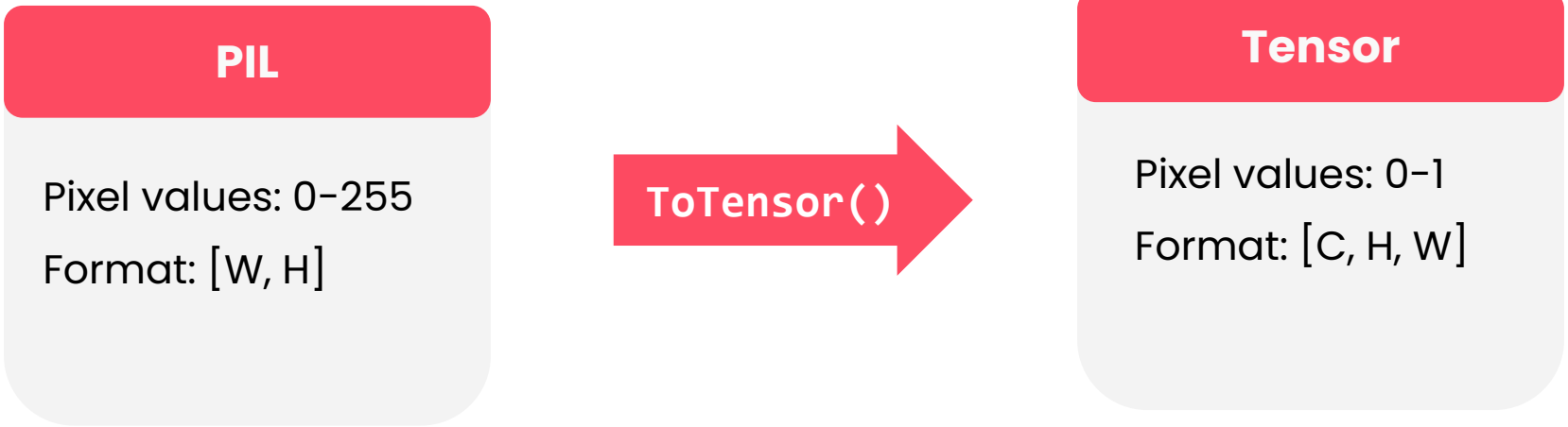
`ToPILImage()`

`Resize()`

`CenterCrop()`

`Normalize()`

ToTensor()


PIL
Pixel values: 0-255
Format: [W, H]
ToTensor()
Tensor
Pixel values: 0-1
Format: [C, H, W]

ToTensor()

```
import torchvision.transforms as transforms
from PIL import Image

# Load an image
image = Image.open('./images/mangoes.jpg')

# Dimensions of the original PIL image
print("Original PIL Image Dimensions:", image.size)

# Convert the PIL Image to a PyTorch Tensor
img_tensor = transforms.ToTensor()(image)

# Dimensions (shape) of the tensor
print(f"Dimensions After Converting to a Tensor: {img_tensor.shape}")
```

ToTensor()

```
import torchvision.transforms as transforms
from PIL import Image

# Load an image
image = Image.open('./images/mangoes.jpg')

# Dimensions of the original PIL image
print("Original PIL Image Dimensions:", image.size)

# Convert the PIL Image to a PyTorch Tensor
img_tensor = transforms.ToTensor()(image)

# Dimensions (shape) of the tensor
print(f"Dimensions After Converting to a Tensor: {img_tensor.shape}")
```

ToTensor()

```
import torchvision.transforms as transforms
from PIL import Image

# Load an image
image = Image.open('./images/mangoes.jpg')

# Dimensions of the original PIL image
print("Original PIL Image Dimensions:", image.size)

# Convert the PIL Image to a PyTorch Tensor
img_tensor = transforms.ToTensor()(image)

# Dimensions (shape) of the tensor
print(f"Dimensions After Converting to a Tensor: {img_tensor.shape}")
```

ToTensor()

```
# Dimensions of the original PIL image
print("Original PIL Image Dimensions:", image.size)
```

ToTensor()

```
# Dimensions of the original PIL image
print("Original PIL Image Dimensions:", image.size)
```

Output:

```
Original PIL Image Dimensions: (2048, 2048)
```

ToTensor()

```
import torchvision.transforms as transforms
from PIL import Image

# Load an image
image = Image.open('./images/mangoes.jpg')

# Dimensions of the original PIL image
print("Original PIL Image Dimensions:", image.size)

# Convert the PIL Image to a PyTorch Tensor
img_tensor = transforms.ToTensor()(image)

# Dimensions (shape) of the tensor
print(f"Dimensions After Converting to a Tensor: {img_tensor.shape}")
```

ToTensor()

```
# Dimensions (shape) of the tensor
print(f"Dimensions After Converting to a Tensor: {img_tensor.shape}")
```

Output:

```
Dimensions After Converting to a Tensor: torch.Size([3, 2048, 2048])
```

ToPILImage()


PIL
Pixel values: 0-255
Format: [H, W, C]
ToPILImage()
Tensor
Pixel values: 0-1
Format: [C, H, W]

ToPILImage()

```
# Converting back to PIL
img_pil = transforms.ToPILImage()(img_tensor)

# Dimensions of the converted back PIL image
print("Dimensions After Converting Back to PIL:", img_pil.size)
```

Output:

```
Dimensions After Converting Back to PIL: (2048, 2048)
```

ToPILImage()

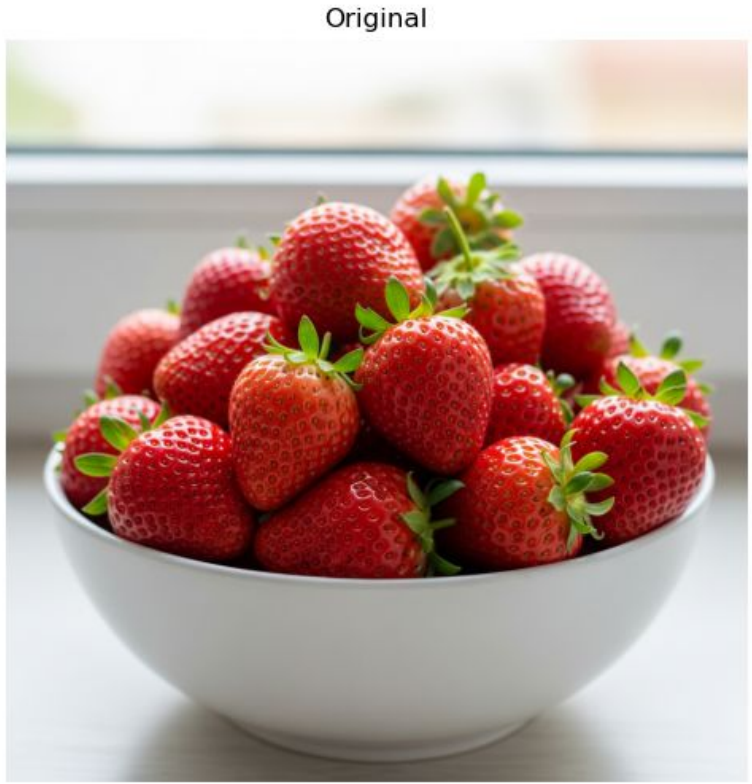
Resize()

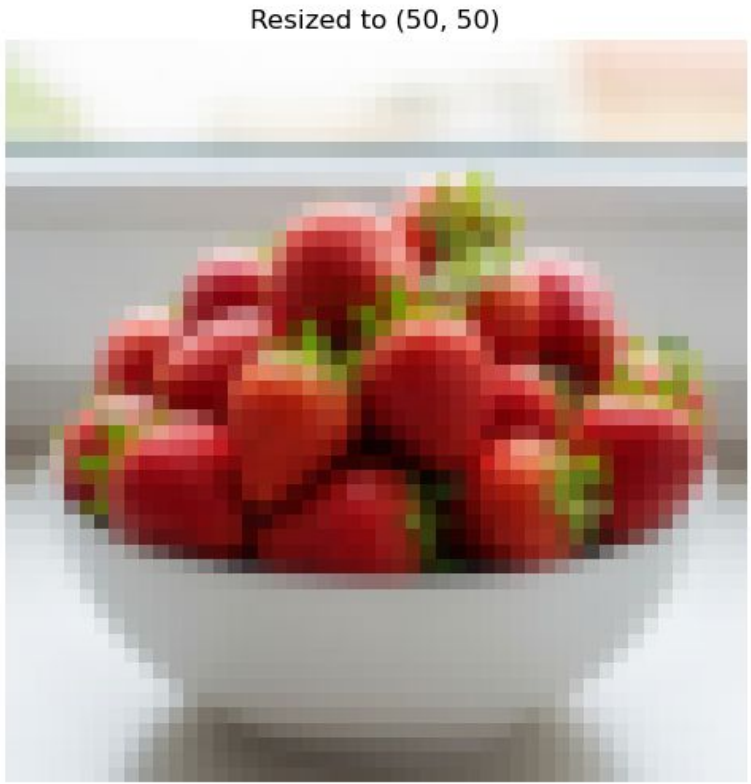
```
# Define the resize transformation (50x50 square)
resize_transform = transforms.Resize(size=50)

# Apply the transformation
resized_image = resize_transform(original_image)
```

Resize()

```
# Define the resize transformation (50x50 square)
resize_transform = transforms.Resize(size=50)

# Apply the transformation
resized_image = resize_transform(original_image)
```

Resize()





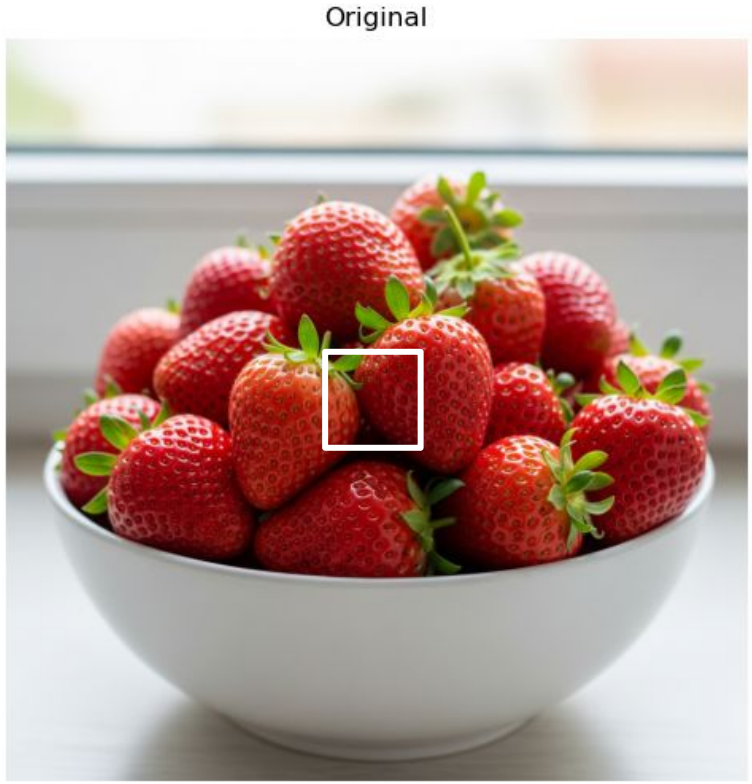
CenterCrop()

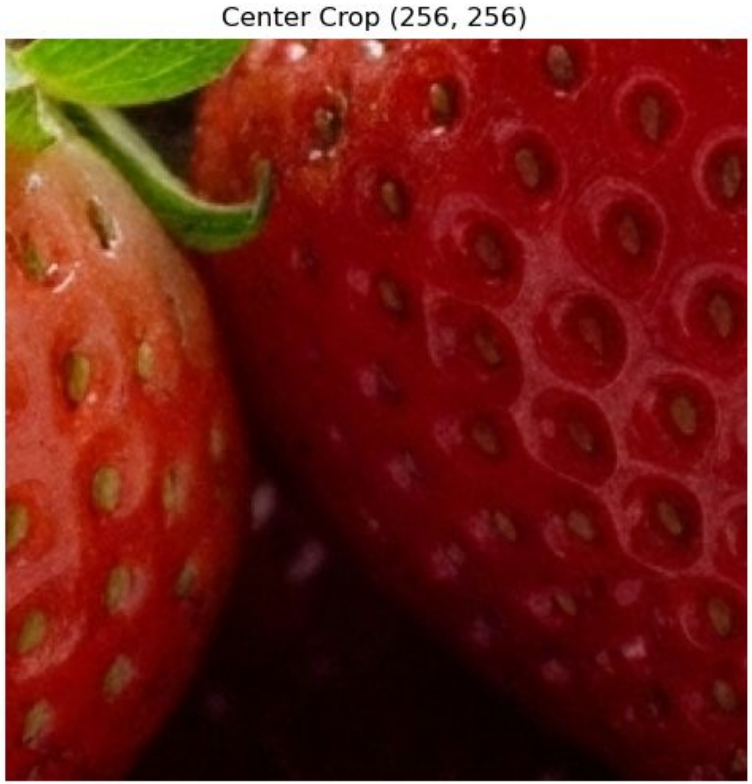
```
# Define the center crop transformation (256x256)
center_crop_transform = transforms.CenterCrop(size=256)

# Apply the transformation
cropped_image = center_crop_transform(original_image)
```

CenterCrop()

```
# Define the center crop transformation (256x256)
center_crop_transform = transforms.CenterCrop(size=256)

# Apply the transformation
cropped_image = center_crop_transform(original_image)
```

CenterCrop()





Normalize()

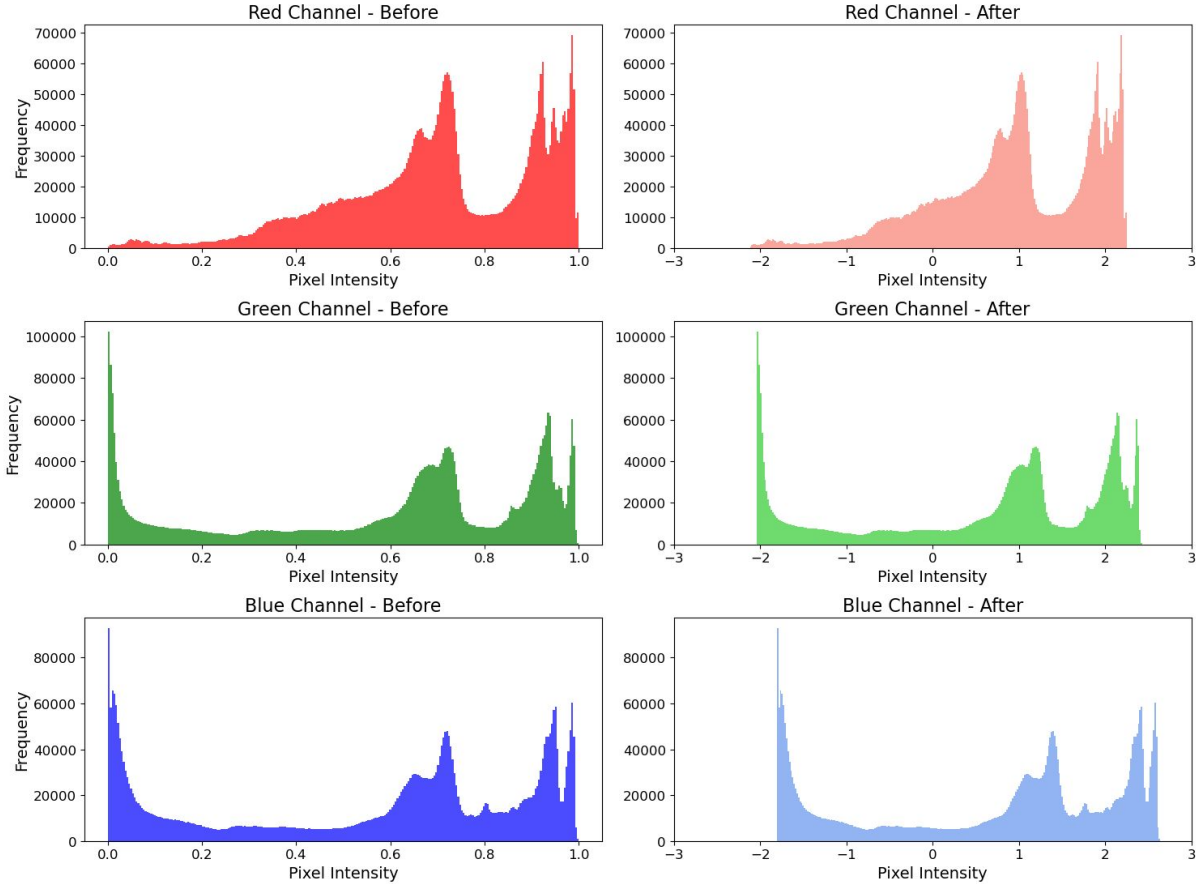
```
transforms.Normalize(mean=[0.485, 0.456, 0.406],
                     std=[0.229, 0.224, 0.225])
```

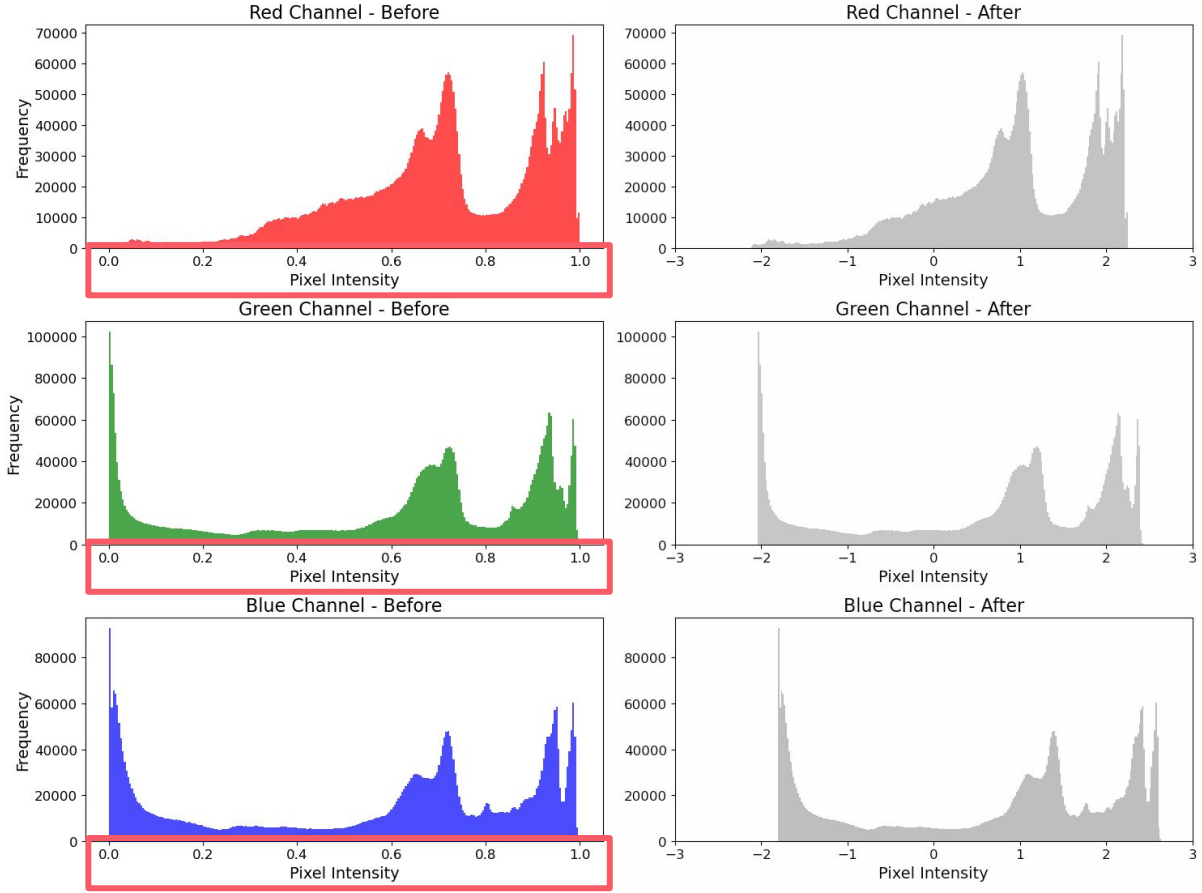
Normalize()

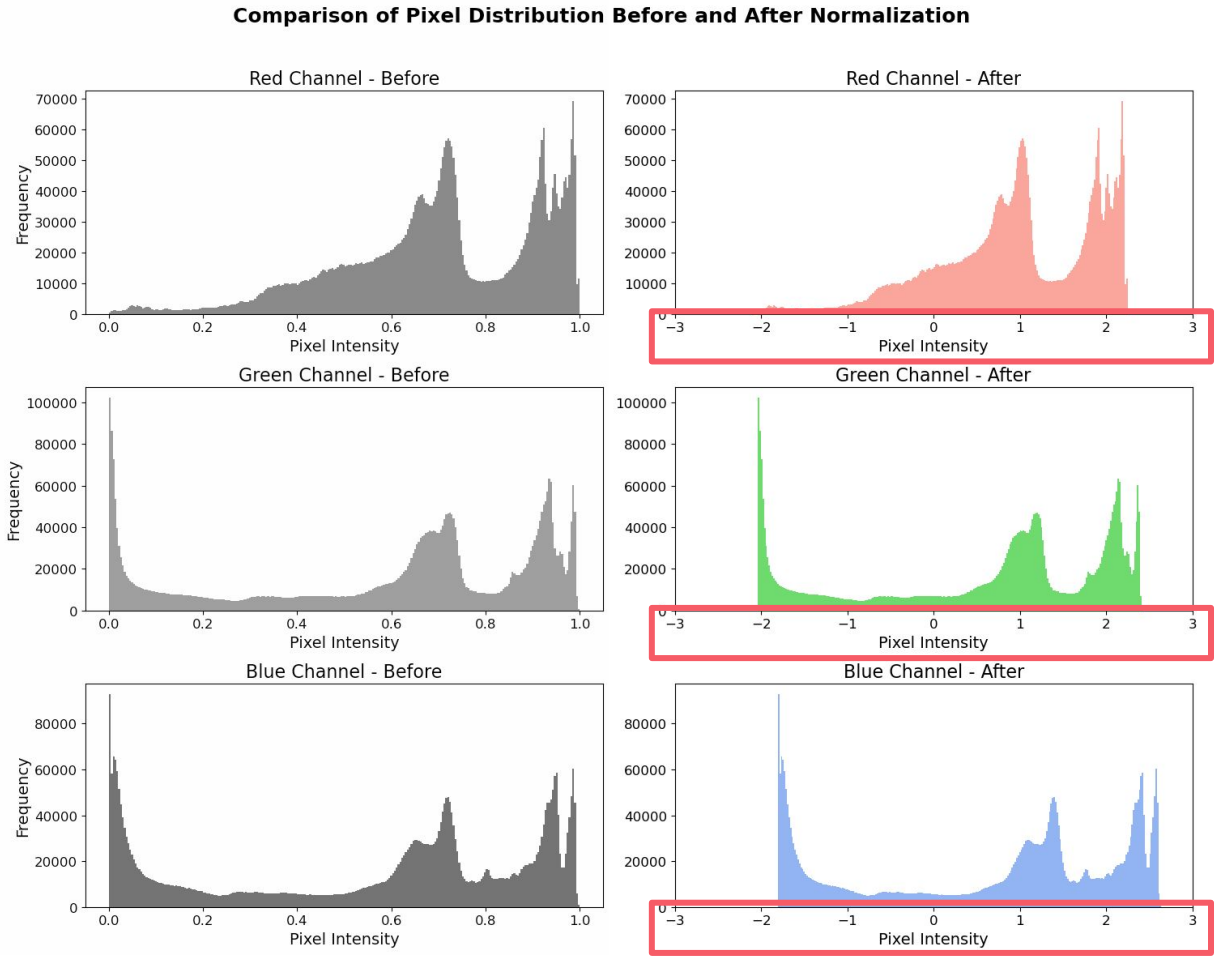
```
transforms.Normalize(mean=[0.485, 0.456, 0.406],
                     std=[0.229, 0.224, 0.225])
```

Computing the z-score

$$z = \frac{x - \mu}{\sigma}$$


Comparison of Pixel Distribution Before and After Normalization
Red Channel - Before
Red Channel - After
Green Channel - Before
Green Channel - After
Blue Channel - Before
Blue Channel - After
Frequency
Pixel Intensity


Comparison of Pixel Distribution Before and After Normalization
Red Channel - Before
Red Channel - After
Green Channel - Before
Green Channel - After
Blue Channel - Before
Blue Channel - After
Frequency
Pixel Intensity


Comparison of Pixel Distribution Before and After Normalization
Red Channel - Before
Red Channel - After
Green Channel - Before
Green Channel - After
Blue Channel - Before
Blue Channel - After
Frequency
Pixel Intensity

Normalize()

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

Normalize()

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

Manual calculation

- Uses training dataset
- Not very accurate for small datasets

Normalize()

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

Manual calculation

- Uses training dataset
- Not very accurate for small datasets

Pre-Computed

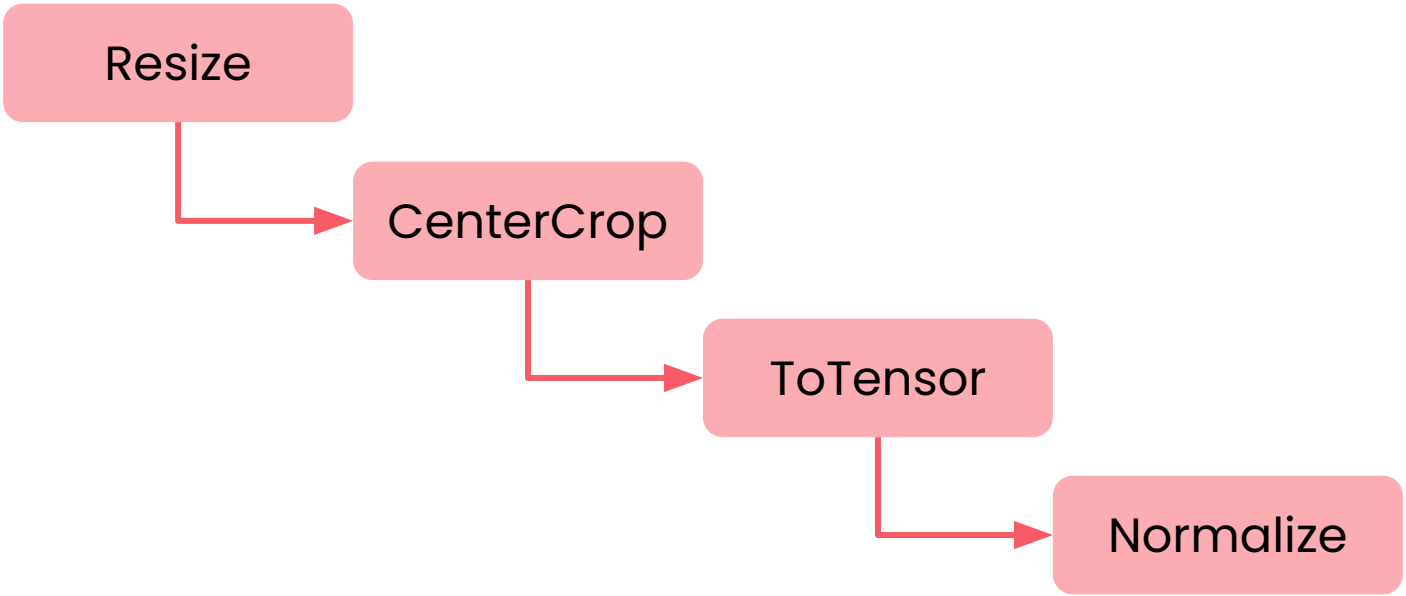
- Uses a large external dataset (ImageNet)
- Useful for most applications

DeepLearning.AI

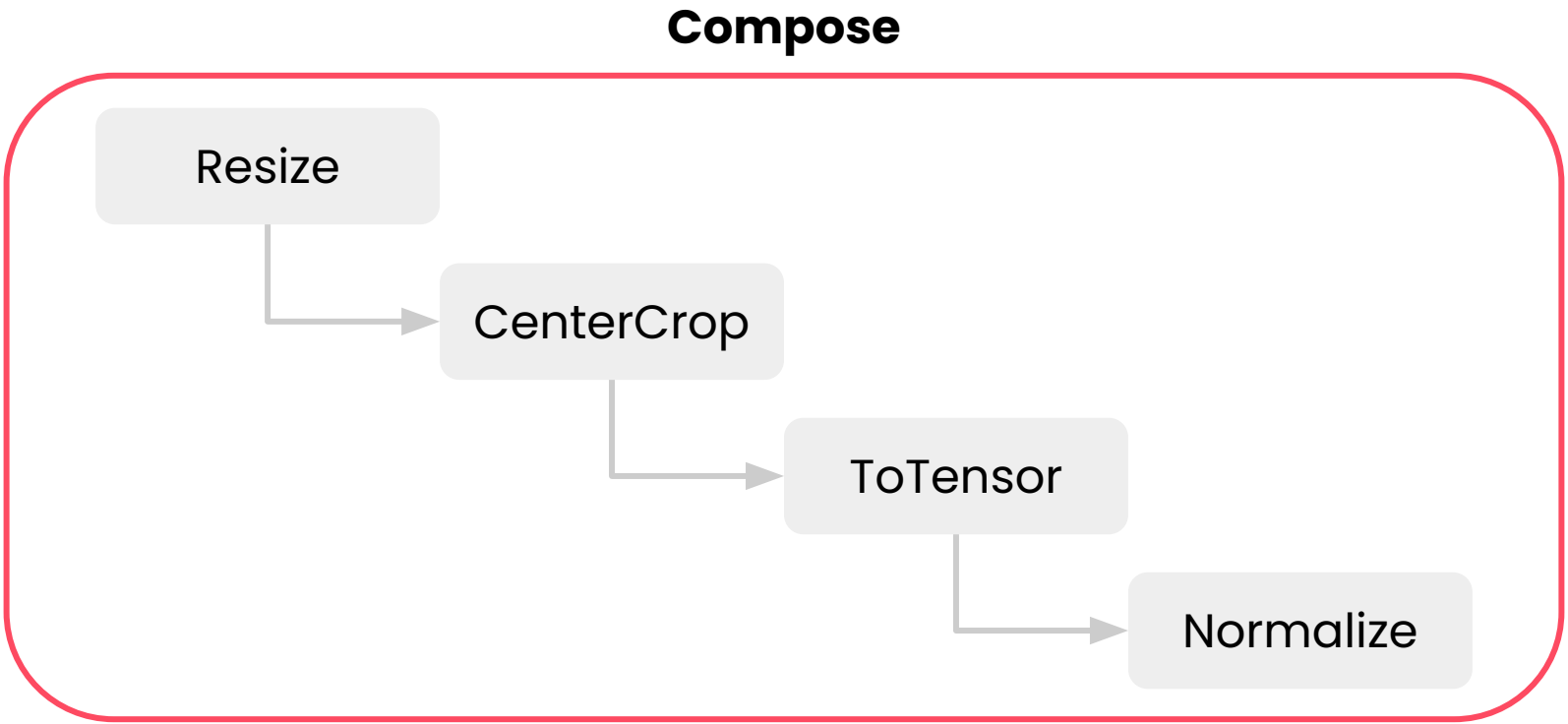
TorchVision Preprocessing Pipeline and Augmentation

Working with images using TorchVision

Preprocessing pipeline



Preprocessing pipeline



Preprocessing pipeline

```
# A simple transform to get a clean, un-augmented version of the images
base_transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])
```

Preprocessing pipeline

```
# A simple transform to get a clean, un-augmented version of the images
base_transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])
```

Preprocessing pipeline

```
# A simple transform to get a clean, un-augmented version of the images
base_transform = transforms.Compose([
   transforms.Resize(256),
   transforms.CenterCrop(224),
   transforms.ToTensor(),
   transforms.Normalize(mean=[0.485, 0.456, 0.406],
                        std=[0.229, 0.224, 0.225])
])
```

Preprocessing pipeline

```
# A simple transform to get a clean, un-augmented version of the images
base_transform = transforms.Compose([
   transforms.Resize(256),
   transforms.CenterCrop(224),
   transforms.ToTensor(),
   transforms.Normalize(mean=[0.485, 0.456, 0.406],
                        std=[0.229, 0.224, 0.225])
])
```

Preprocessing pipeline

```
# A simple transform to get a clean, un-augmented version of the images
base_transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])
```

Preprocessing pipeline

```
# A simple transform to get a clean, un-augmented version of the images
base_transform = transforms.Compose([
   transforms.Resize(256),
   transforms.CenterCrop(224),
   transforms.ToTensor(),
   transforms.Normalize(mean=[0.485, 0.456, 0.406],
                        std=[0.229, 0.224, 0.225])
])
```

DataLoader transforms

```
# Load the OxfordIIITPet dataset, applying the simple transform to each image
my_dataset = datasets.OxfordIIITPet(root=ox3_pet_data_path,
                                    split='test',
                                    download=ox3_pet_download,
                                    transform=base_transform
                                   )
```

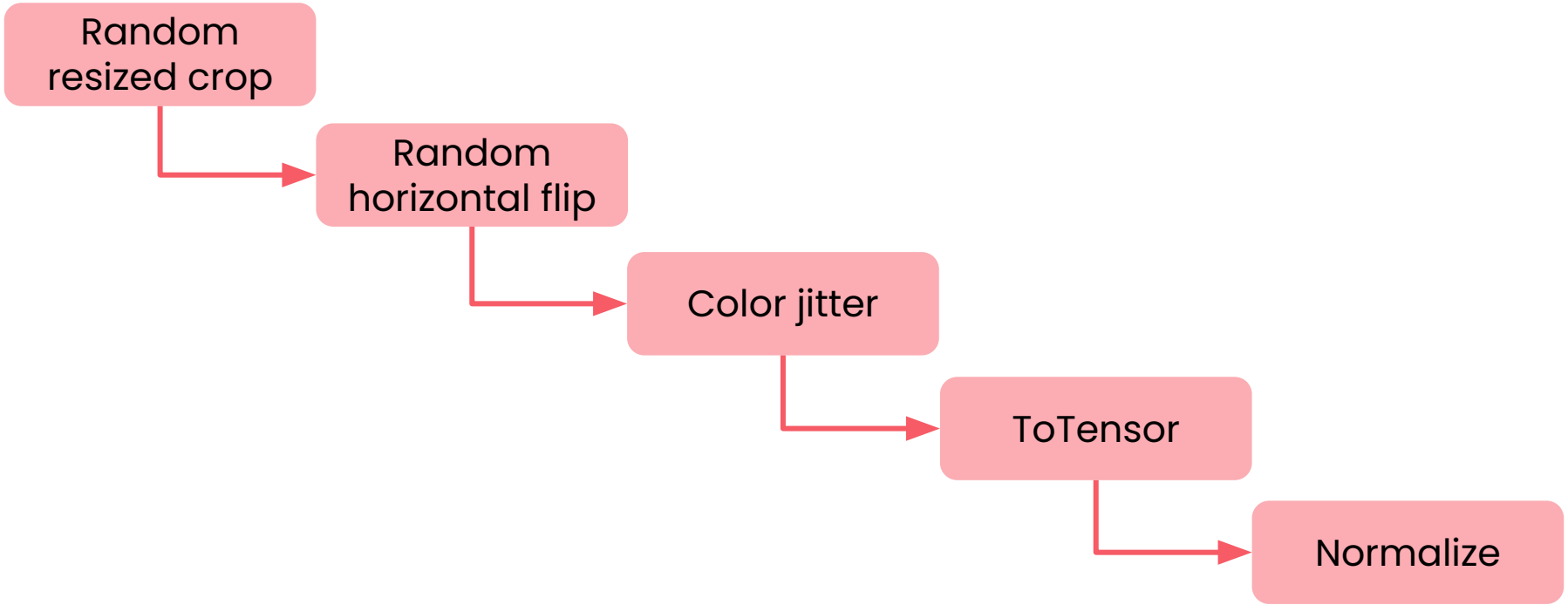
DataLoader transforms

```
# Load the OxfordIIITPet dataset, applying the simple transform to each image
my_dataset = datasets.OxfordIIITPet(root=ox3_pet_data_path,
                                    split='test',
                                    download=ox3_pet_download,
                                    transform=base_transform
                                    )
```

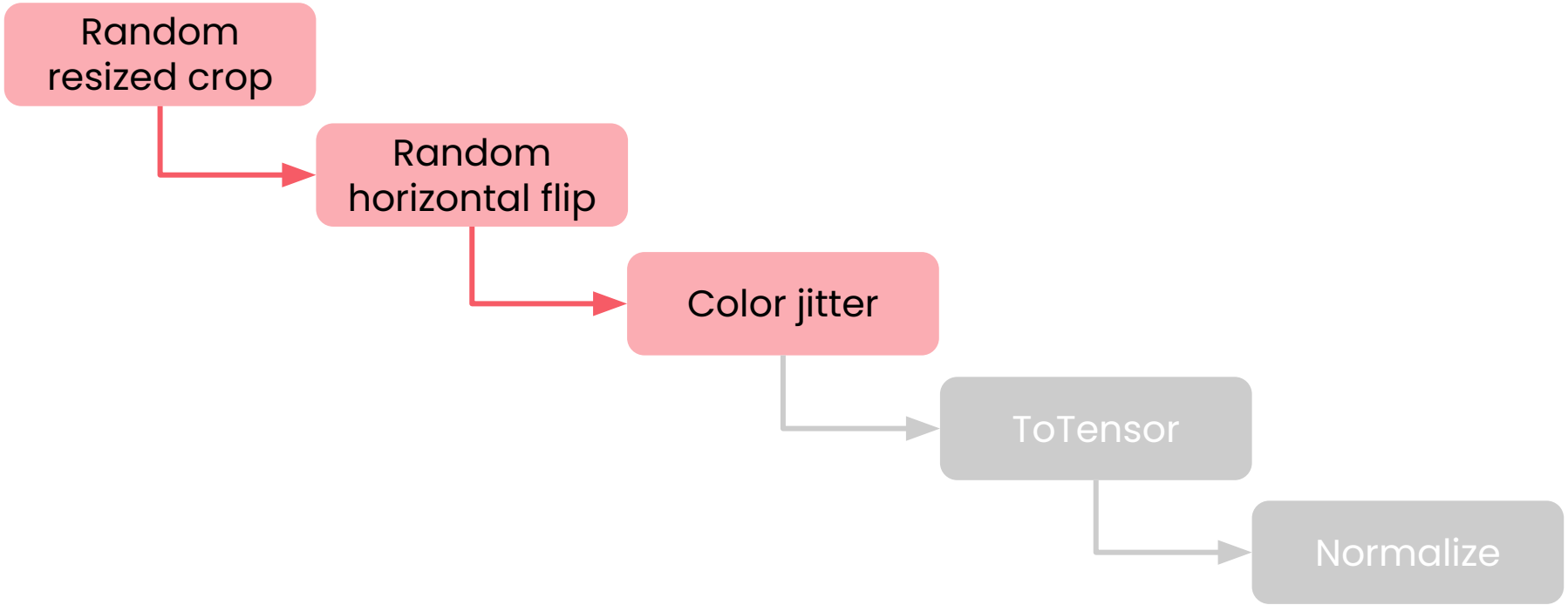
Image augmentation

Image augmentation

Augmentation pipeline



Augmentation pipeline



Augmentation pipeline

```
# The full augmentation pipeline with all random transformations
full_augmentation_pipeline = transforms.Compose([
    transforms.RandomResizedCrop(size=224),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.ColorJitter(brightness=0.5, contrast=0.5),
    transforms.ToTensor(),
    # Using 'mean' and 'std' values as calculated on the 100x100 images
    transforms.Normalize(mean=dataset_mean,
                         std=dataset_std)
])
```

RandomResizedCrop()

```
# The full augmentation pipeline with all random transformations
full_augmentation_pipeline = transforms.Compose([
    transforms.RandomResizedCrop(size=224),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.ColorJitter(brightness=0.5, contrast=0.5),
    transforms.ToTensor(),
    # Using 'mean' and 'std' values as calculated on the 100x100 images
    transforms.Normalize(mean=dataset_mean,
                         std=dataset_std)
])
```

RandomResizedCrop()

RandomHorizontalFlip()

```
# The full augmentation pipeline with all random transformations
full_augmentation_pipeline = transforms.Compose([
    transforms.RandomResizedCrop(size=224),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.ColorJitter(brightness=0.5, contrast=0.5),
    transforms.ToTensor(),
    # Using 'mean' and 'std' values as calculated on the 100x100 images
    transforms.Normalize(mean=dataset_mean,
                         std=dataset_std)
])
```

RandomHorizontalFlip()

ColorJitter()

```
# The full augmentation pipeline with all random transformations
full_augmentation_pipeline = transforms.Compose([
    transforms.RandomResizedCrop(size=224),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.ColorJitter(brightness=0.5, contrast=0.5),
    transforms.ToTensor(),
    # Using 'mean' and 'std' values as calculated on the 100x100 images
    transforms.Normalize(mean=dataset_mean,
                         std=dataset_std)
])
```

ColorJitter()

Original

ColorJitter

Augmentation pipeline

```
# The full augmentation pipeline with all random transformations
full_augmentation_pipeline = transforms.Compose([
    transforms.RandomResizedCrop(size=224),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.ColorJitter(brightness=0.5, contrast=0.5),
    transforms.ToTensor(),
    # Using 'mean' and 'std' values as calculated on the 100x100 images
    transforms.Normalize(mean=dataset_mean,
                         std=dataset_std)
])
```

Salt and pepper noise

```
class SaltAndPepperNoise:

    def __init__(self, salt_vs_pepper=0.5, amount=0.04):
        self.s_vs_p = salt_vs_pepper
        self.amount = amount

    def __call__(self, image):

        output = np.copy(np.array(image))

        num_salt = np.ceil(self.amount * image.size[0] * image.size[1] * self.s_vs_p)
        coords = [np.random.randint(0, i - 1, int(num_salt)) for i in image.size]
        output[coords[1], coords[0]] = 255

        num_pepper = np.ceil(self.amount * image.size[0] * image.size[1] * (1.0 - self.s_vs_p))
        coords = [np.random.randint(0, i - 1, int(num_pepper)) for i in image.size]
        output[coords[1], coords[0]] = 0

        return Image.fromarray(output)
```

```
class SaltAndPepperNoise:

    def __init__(self, salt_vs_pepper=0.5, amount=0.04):
        self.s_vs_p = salt_vs_pepper
        self.amount = amount

    def __call__(self, image):

        output = np.copy(np.array(image))

        num_salt = np.ceil(self.amount * image.size[0] * image.size[1] * self.s_vs_p)
        coords = [np.random.randint(0, i - 1, int(num_salt)) for i in image.size]
        output[coords[1], coords[0]] = 255

        num_pepper = np.ceil(self.amount * image.size[0] * image.size[1] * (1.0 - self.s_vs_p))
        coords = [np.random.randint(0, i - 1, int(num_pepper)) for i in image.size]
        output[coords[1], coords[0]] = 0

        return Image.fromarray(output)
```

```
class SaltAndPepperNoise:

    def __init__(self, salt_vs_pepper=0.5, amount=0.04):
        self.s_vs_p = salt_vs_pepper
        self.amount = amount

    def __call__(self, image):

        output = np.copy(np.array(image))

        num_salt = np.ceil(self.amount * image.size[0] * image.size[1] * self.s_vs_p)
        coords = [np.random.randint(0, i - 1, int(num_salt)) for i in image.size]
        output[coords[1], coords[0]] = 255

        num_pepper = np.ceil(self.amount * image.size[0] * image.size[1] * (1.0 - self.s_vs_p))
        coords = [np.random.randint(0, i - 1, int(num_pepper)) for i in image.size]
        output[coords[1], coords[0]] = 0

        return Image.fromarray(output)
```

```
class SaltAndPepperNoise:

    def __init__(self, salt_vs_pepper=0.5, amount=0.04):
        self.s_vs_p = salt_vs_pepper
        self.amount = amount

    def __call__(self, image):

        output = np.copy(np.array(image))

        num_salt = np.ceil(self.amount * image.size[0] * image.size[1] * self.s_vs_p)
        coords = [np.random.randint(0, i - 1, int(num_salt)) for i in image.size]
        output[coords[1], coords[0]] = 255

        num_pepper = np.ceil(self.amount * image.size[0] * image.size[1] * (1.0 - self.s_vs_p))
        coords = [np.random.randint(0, i - 1, int(num_pepper)) for i in image.size]
        output[coords[1], coords[0]] = 0

        return Image.fromarray(output)
```

```
class SaltAndPepperNoise:

    def __init__(self, salt_vs_pepper=0.5, amount=0.04):
        self.s_vs_p = salt_vs_pepper
        self.amount = amount

    def __call__(self, image):

        output = np.copy(np.array(image))

        num_salt = np.ceil(self.amount * image.size[0] * image.size[1] * self.s_vs_p)
        coords = [np.random.randint(0, i - 1, int(num_salt)) for i in image.size]
        output[coords[1], coords[0]] = 255

        num_pepper = np.ceil(self.amount * image.size[0] * image.size[1] * (1.0 - self.s_vs_p))
        coords = [np.random.randint(0, i - 1, int(num_pepper)) for i in image.size]
        output[coords[1], coords[0]] = 0

        return Image.fromarray(output)
```

```
class SaltAndPepperNoise:

    def __init__(self, salt_vs_pepper=0.5, amount=0.04):
        self.s_vs_p = salt_vs_pepper
        self.amount = amount

    def __call__(self, image):

        output = np.copy(np.array(image))

        num_salt = np.ceil(self.amount * image.size[0] * image.size[1] * self.s_vs_p)
        coords = [np.random.randint(0, i - 1, int(num_salt)) for i in image.size]
        output[coords[1], coords[0]] = 255

        num_pepper = np.ceil(self.amount * image.size[0] * image.size[1] * (1.0 - self.s_vs_p))
        coords = [np.random.randint(0, i - 1, int(num_pepper)) for i in image.size]
        output[coords[1], coords[0]] = 0

        return Image.fromarray(output)
```

Adding to the Augmentation Pipeline

```
# The full augmentation pipeline with all random transformations
full_augmentation_pipeline = transforms.Compose([
    transforms.RandomResizedCrop(size=224),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.ColorJitter(brightness=0.05, contrast=0.05),
    SaltAndPepperNoise(amount=0.001),
    transforms.ToTensor(),
    # Using 'mean' and 'std' values as calculated on the 100x100 images
    transforms.Normalize(mean=dataset_mean,
                         std=dataset_std)
])
```

Other types of noise

Gaussian Noise

Poisson Noise

DeepLearning.AI

TorchVision Datasets

Working with images using TorchVision

Use cases for TorchVision datasets

Pre-built datasets

- Have labels
- Established benchmarks
- Faster development
- Test and debug

Use cases for TorchVision datasets

Pre-built datasets

- Have labels
- Established benchmarks
- Faster development
- Test and debug

Custom datasets

- Load, preprocess, and augment own data
- Solve unique challenges

Use cases for TorchVision datasets

Pre-built datasets

- Have labels
- Established benchmarks
- Faster development
- Test and debug

Custom datasets

- Load, preprocess, and augment own data
- Solve unique challenges

Fake datasets

- Fast data generation
- Test and debug

Datasets for a variety of tasks:


Image classification


Object detection


Segmentation


Optical flow

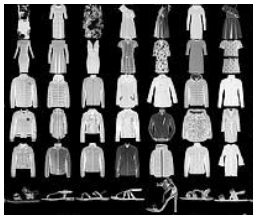

Stereo matching


Image captioning


Video classification and prediction

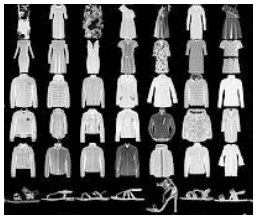
TorchVision pre-built datasets

Fashion MNIST



TorchVision pre-built datasets

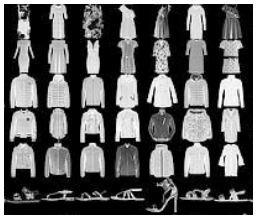
Fashion MNIST



SVHN: Street View House Numbers

TorchVision pre-built datasets

Fashion MNIST

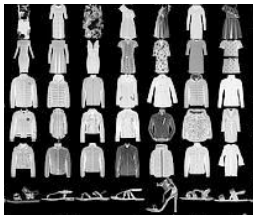


SVHN: Street View House Numbers

Celeb-A

TorchVision pre-built datasets

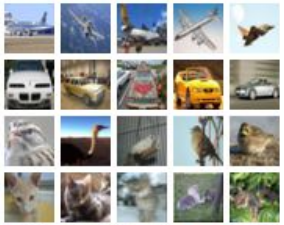
Fashion MNIST



SVHN: Street View House Numbers

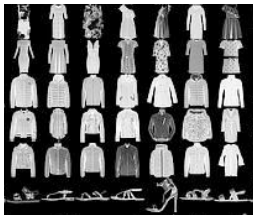
Celeb-A

COCO: Common Objects in Context



TorchVision pre-built datasets

Fashion MNIST



SVHN: Street View House Numbers

Celeb-A

COCO: Common Objects in Context

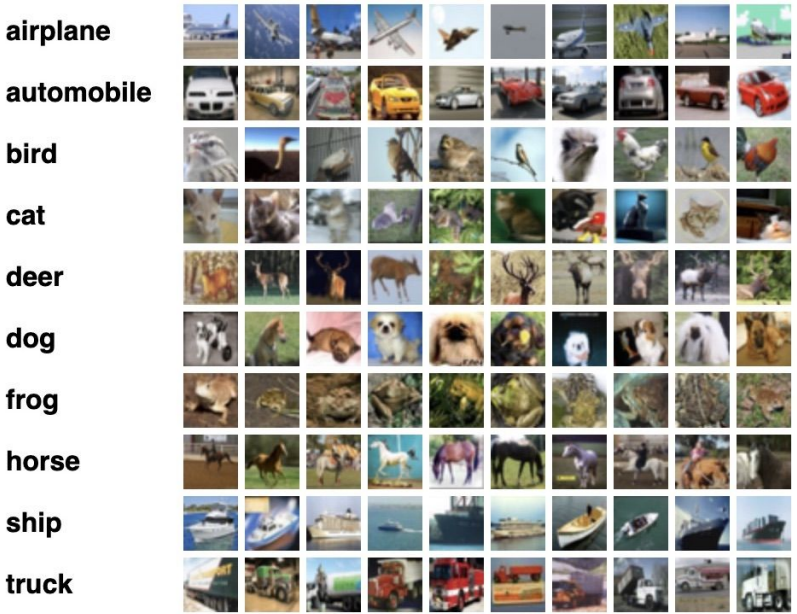
Pascal VOC

ImageNet

- High-resolution images across 1,000 categories
- 1,000,000+ samples
- Gold standard for pretraining deep neural networks

CIFAR

- 32x32 color images
- 60,000 images:
  - 50,000 train
  - 10,000 test
- CIFAR-10 → 10 classes
- CIFAR-100 → 100 classes



Loading a built-in dataset: CIFAR-10

```
import torchvision.datasets as datasets
import torch.utils.data as data

root_dir = './pytorch_datasets'

# Load sample images
cifar_dataset = datasets.CIFAR10(root=root_dir,
                                 train=False,
                                 download=True,
                                 transform=cifar_transformation
                                )

cifar_dataloader = data.DataLoader(cifar_dataset, batch_size=8, shuffle=True)
```

Loading a built-in dataset: CIFAR-10

```
import torchvision.datasets as datasets
import torch.utils.data as data

root_dir = './pytorch_datasets'

# Load sample images
cifar_dataset = datasets.CIFAR10(root=root_dir,
                                 train=False,
                                 download=True,
                                 transform=cifar_transformation
                                )

cifar_dataloader = data.DataLoader(cifar_dataset, batch_size=8, shuffle=True)
```

Loading a built-in dataset: CIFAR-10

```
import torchvision.datasets as datasets
import torch.utils.data as data

root_dir = './pytorch_datasets'

# Load sample images
cifar_dataset = datasets.CIFAR10(root=root_dir,
                                 train=False,
                                 download=True,
                                 transform=cifar_transformation
                                )

cifar_dataloader = data.DataLoader(cifar_dataset, batch_size=8, shuffle=True)
```

Loading a built-in dataset: CIFAR-10

```
import torchvision.datasets as datasets
import torch.utils.data as data

root_dir = './pytorch_datasets'

# Load sample images
cifar_dataset = datasets.CIFAR10(root=root_dir,
                                 train=False,
                                 download=True,
                                 transform=cifar_transformation
                                )

cifar_dataloader = data.DataLoader(cifar_dataset, batch_size=8, shuffle=True)
```

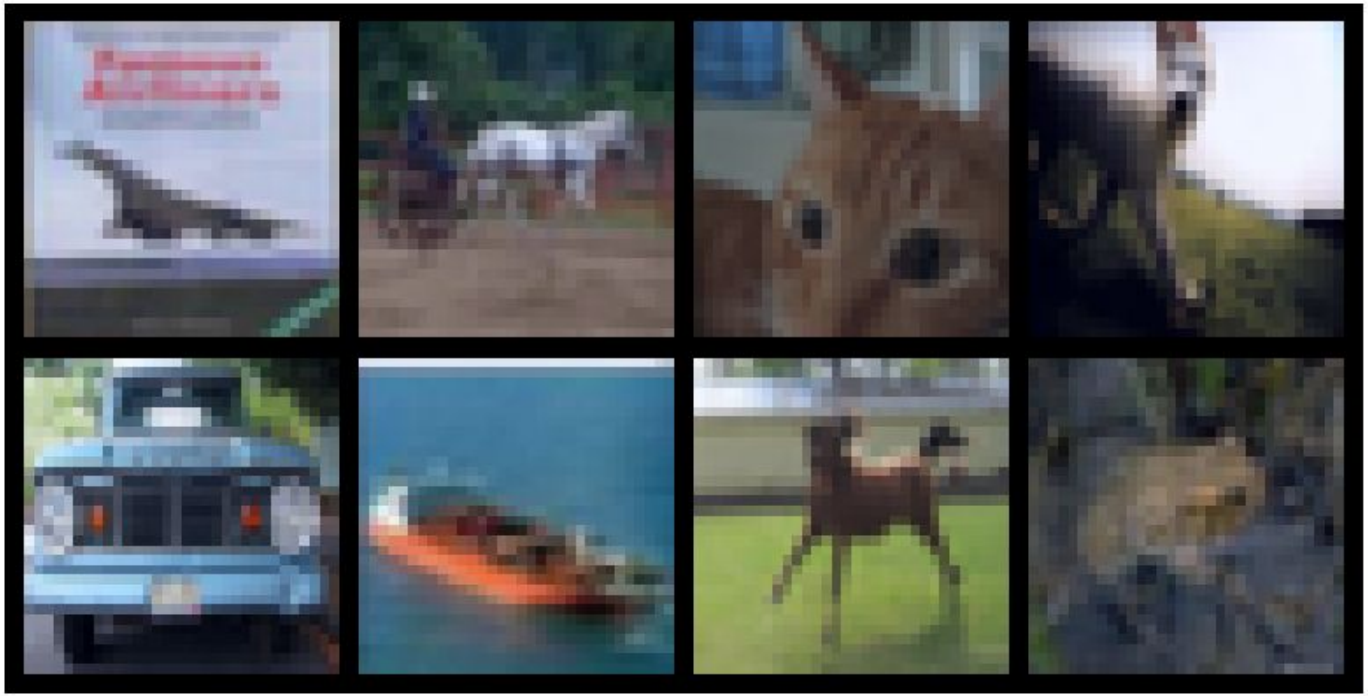
Loading a built-in dataset: CIFAR-10

```
import torchvision.datasets as datasets
import torch.utils.data as data

root_dir = './pytorch_datasets'

# Load sample images
cifar_dataset = datasets.CIFAR10(root=root_dir,
                                 train=False,
                                 download=True,
                                 transform=cifar_transformation
                                 )

cifar_dataloader = data.DataLoader(cifar_dataset, batch_size=8, shuffle=True)
```

Loading a built-in dataset: CIFAR-10

```
import torchvision.datasets as datasets
import torch.utils.data as data

root_dir = './pytorch_datasets'

# Load sample images
cifar_dataset = datasets.CIFAR10(root=root_dir,
                                 train=False,
                                 download=True,
                                 transform=cifar_transformation
                                )

cifar_dataloader = data.DataLoader(cifar_dataset, batch_size=8, shuffle=True)
```

Loading a built-in dataset: CIFAR-10



EMNIST - Digits

- Grayscale 28×28 images of digits 0–9
- 70,000 images:
  - 60,000 train
  - 10,000 test

Loading a built-in dataset: EMNIST

```
emnist_digits_dataset = datasets.EMNIST(root=root_dir,
                                        split='digits',
                                        train=False,
                                        download=True,
                                        transform=emnist_transformation
                                       )

emnist_digits_dataloader = data.DataLoader(emnist_digits_dataset,batch_size=8,shuffle=True)
```

Loading a built-in dataset: EMNIST

```
emnist_digits_dataset = datasets.EMNIST(root=root_dir,
                                        split='digits',
                                        train=False,
                                        download=True,
                                        transform=emnist_transformation
                                       )

emnist_digits_dataloader = data.DataLoader(emnist_digits_dataset,batch_size=8,shuffle=True)
```

Loading a built-in dataset: EMNIST

```
emnist_digits_dataset = datasets.EMNIST(root=root_dir,
                                        split='digits',
                                        train=False,
                                        download=True,
                                        transform=emnist_transformation
                                        )

emnist_digits_dataloader = data.DataLoader(emnist_digits_dataset,batch_size=8,shuffle=True)
```

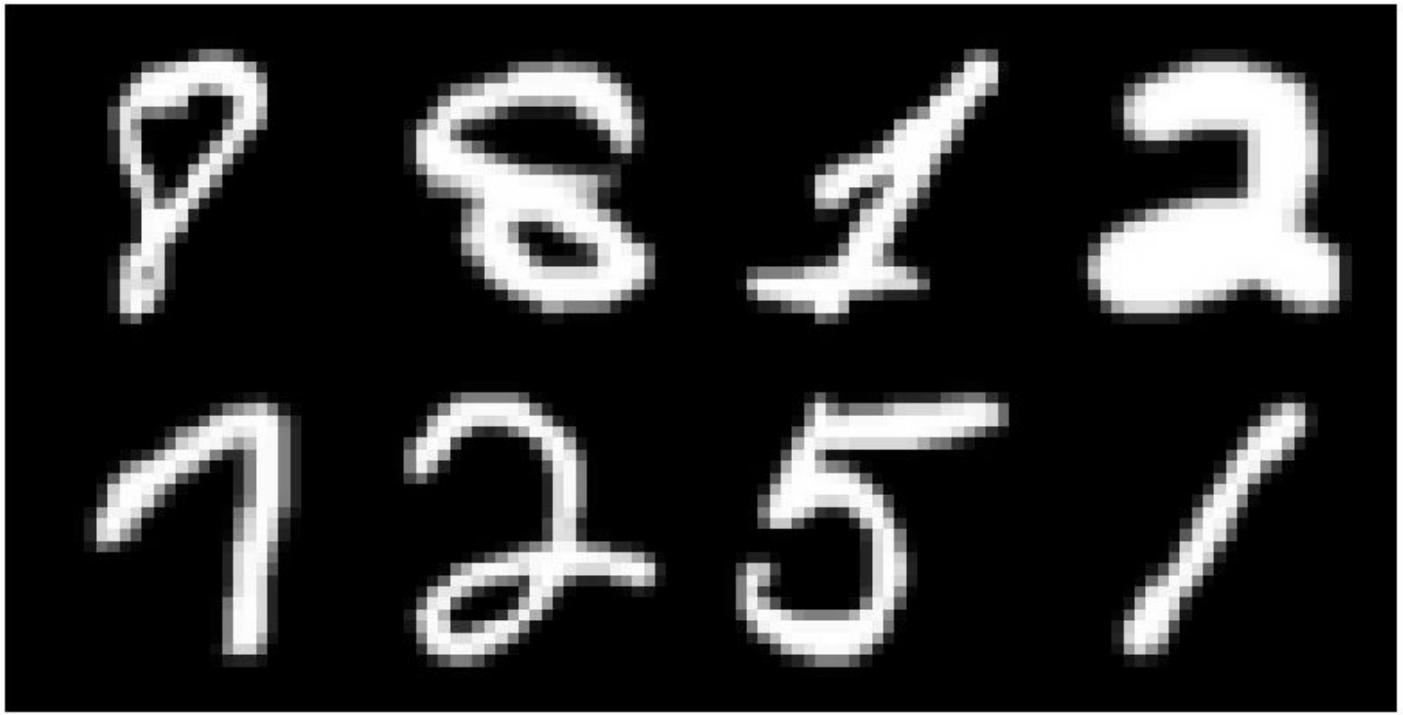
Loading a built-in dataset: EMNIST

```
# Define the transformation pipeline
emnist_transformation = transforms.Compose([
    transforms.RandomRotation(degrees=(90, 90)),
    transforms.RandomVerticalFlip(p=1.0),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,)) # The mean and std must be in a tuple
])
```

Loading a built-in dataset: EMNIST

```
emnist_digits_dataset = datasets.EMNIST(root=root_dir,
                                        split='digits',
                                        train=False,
                                        download=True,
                                        transform=emnist_transformation
                                       )

emnist_digits_dataloader = data.DataLoader(emnist_digits_dataset,batch_size=8,shuffle=True)
```

Loading a built-in dataset: EMNIST



Use cases for TorchVision datasets

Pre-built datasets

- Have labels
- Established benchmarks
- Faster development
- Test and debug

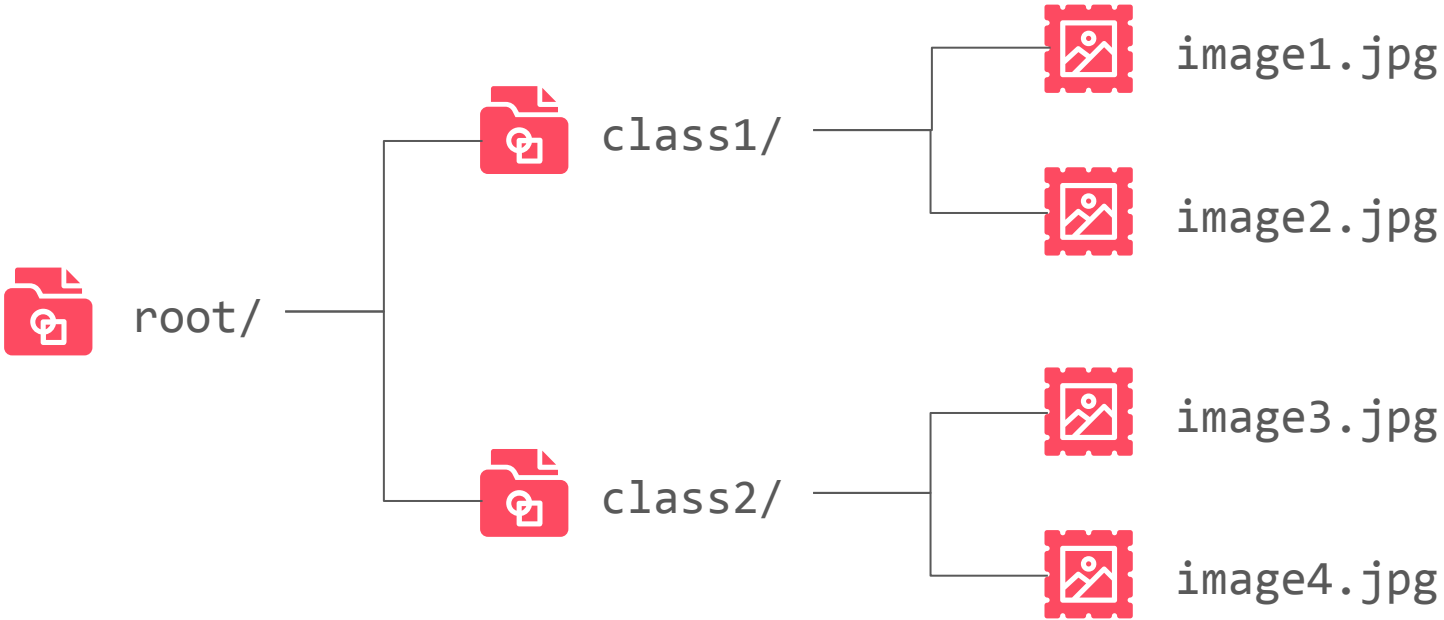
Custom datasets

- Load, preprocess, and augment own data
- Solve unique challenges

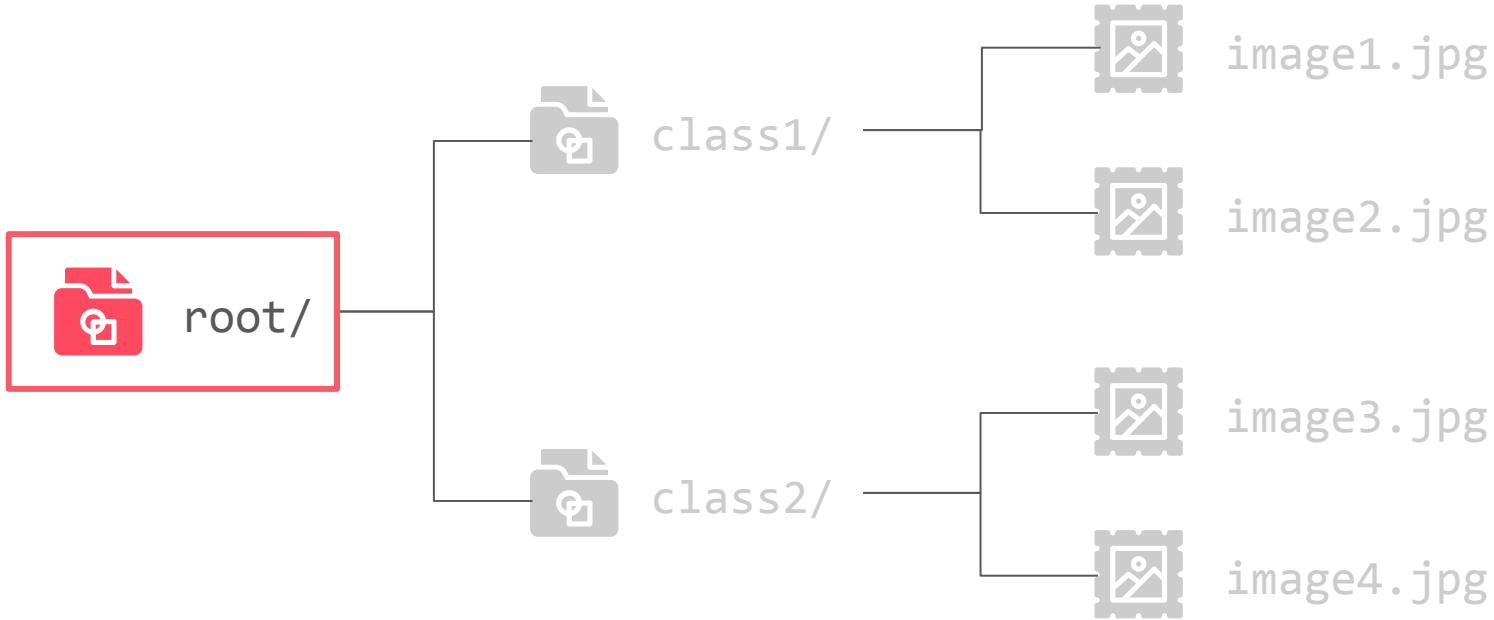
Fake datasets

- Fast data generation
- Test and debug

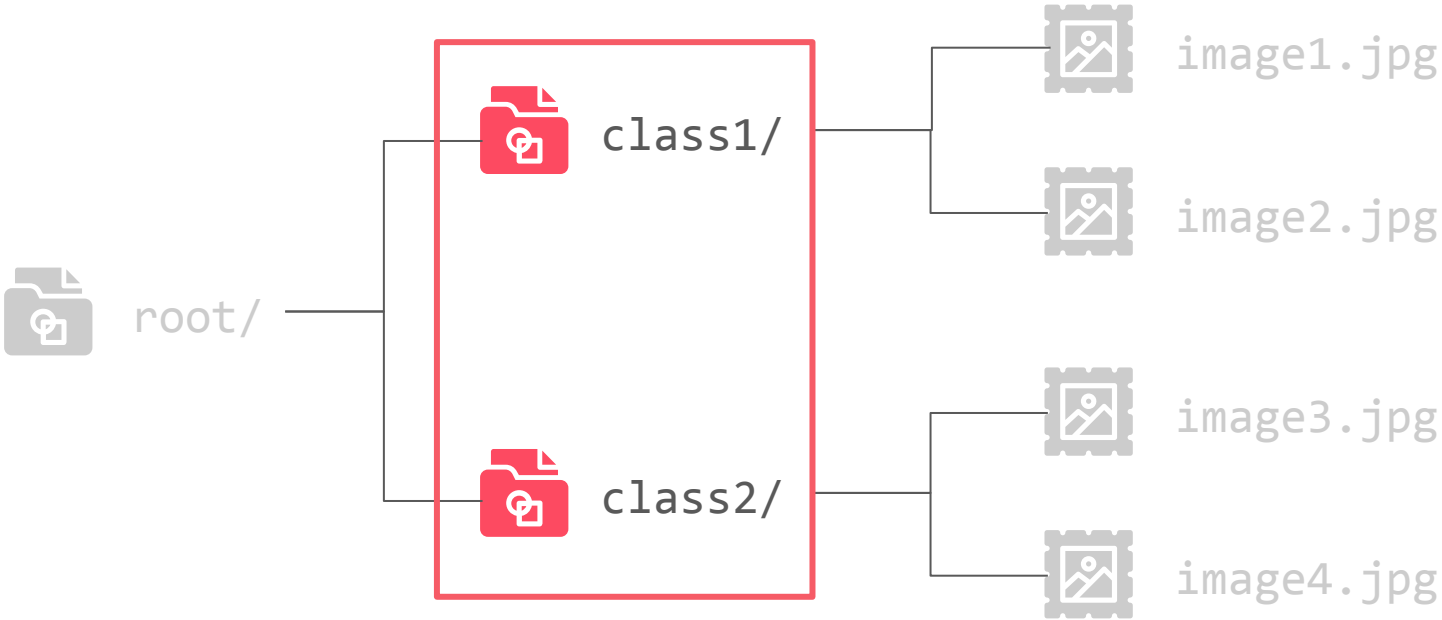
Custom dataset folder structure



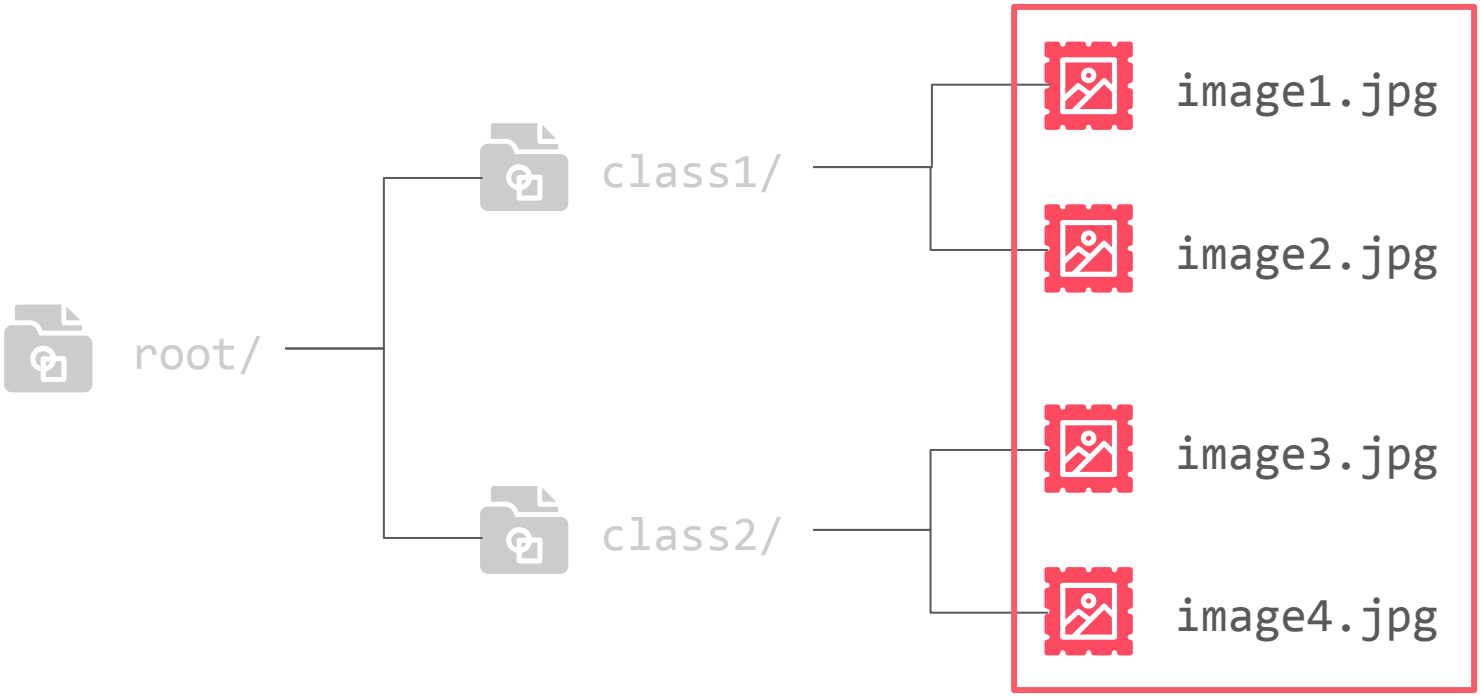
Custom dataset folder structure



Custom dataset folder structure



Custom dataset folder structure



Loading a custom dataset

```
root_dir = './tiny_fruit_and_vegetable'

fruit_dataset = datasets.ImageFolder(root=root_dir,
                                     transform=image_transformation
                                     )

fruit_dataloader = data.DataLoader(fruit_dataset,
                                   batch_size=8,
                                   shuffle=True
                                  )
```

Loading a custom dataset

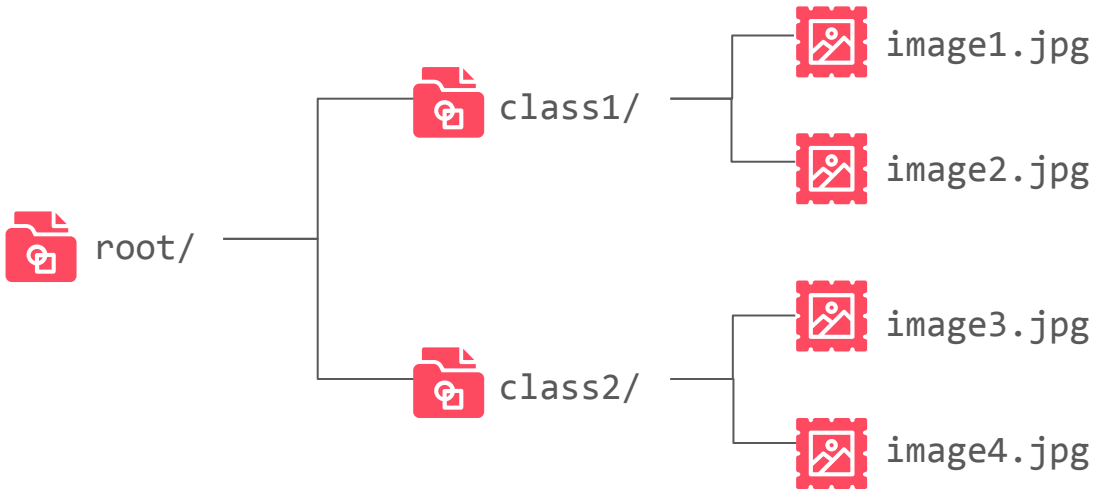
```
root_dir = './tiny_fruit_and_vegetable'

fruit_dataset = datasets.ImageFolder(root=root_dir,
                                     transform=image_transformation
                                     )

fruit_dataloader = data.DataLoader(fruit_dataset,
                                   batch_size=8,
                                   shuffle=True
                                  )
```

Loading a custom dataset

```
root_dir = './tiny_fruit_and_vegetable'

fruit_dataset = datasets.ImageFolder(root=root_dir,
                                     transform=image_transformation
                                     )

fruit_dataloader = data.DataLoader(fruit_dataset,
                                   batch_size=8,
                                   shuffle=True
                                  )
```

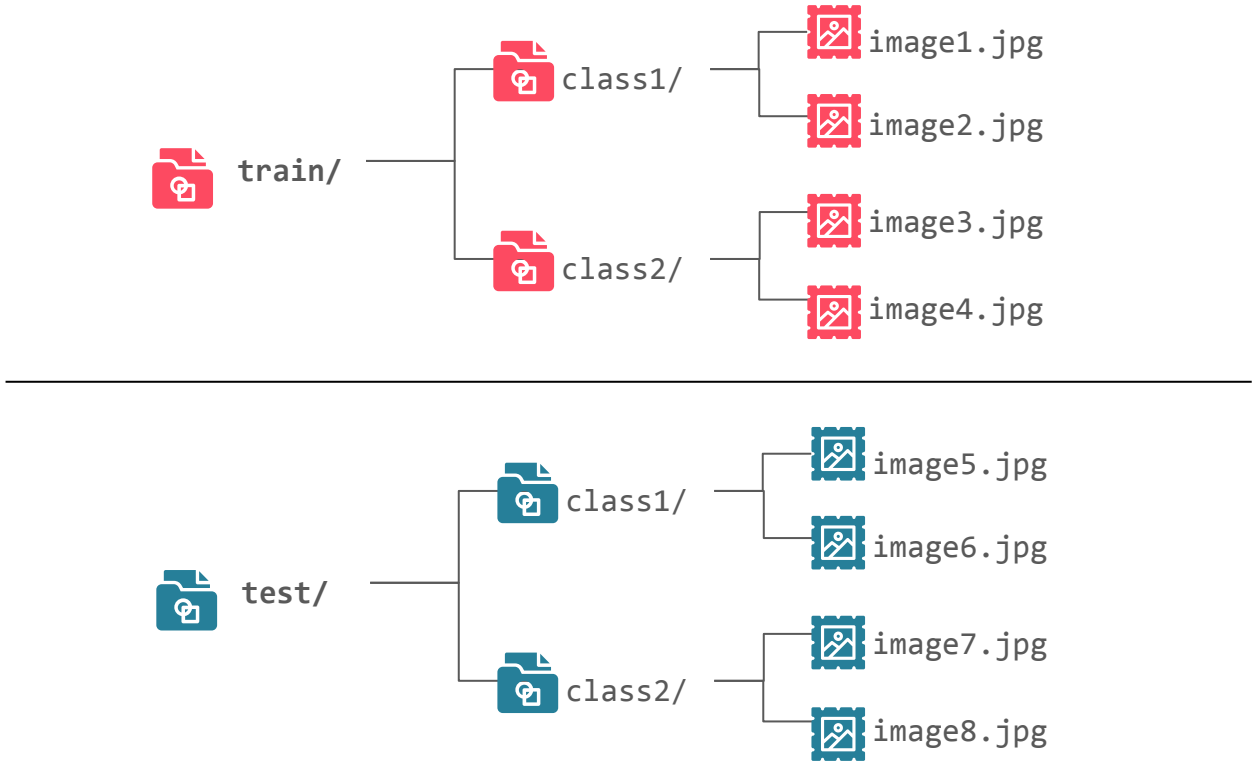
Loading a custom dataset

Train-test split




```
torch.utils.data.random_split(dataset, lengths, generator)
```

Train-test split



Use cases for TorchVision datasets

Pre-built datasets

- Have labels
- Established benchmarks
- Faster development
- Test and debug

Custom datasets

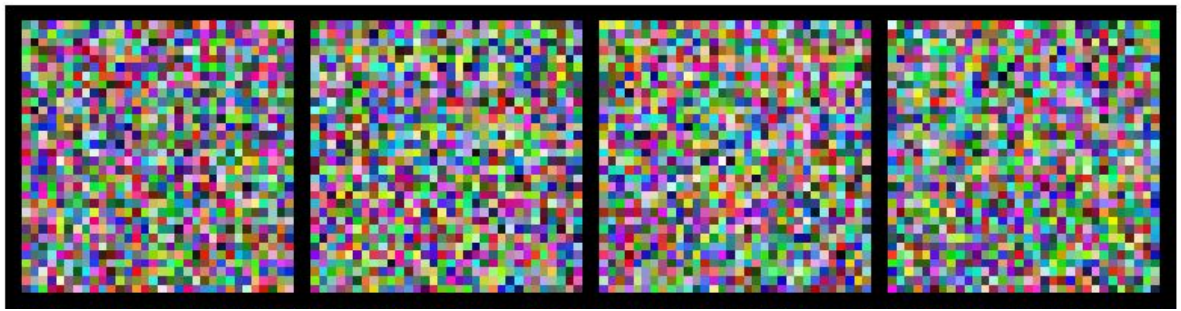
- Load, preprocess, and augment own data
- Solve unique challenges

Fake datasets

- Fast data generation
- Test and debug

Use FakeData to test or debug

```
fake_dataset = datasets.FakeData(size=1000,
                                 image_size=(3, 32, 32),
                                 num_classes=10,
                                 transform=fake_data_transform
                                 )
```



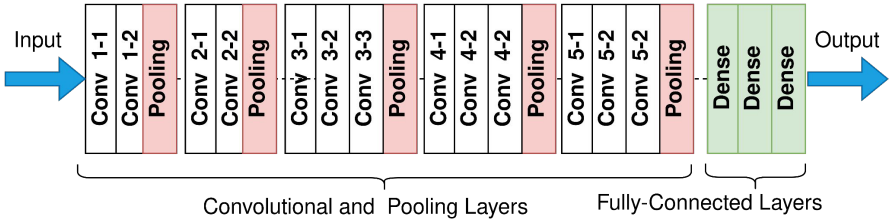
DeepLearning.AI

TorchVision Models

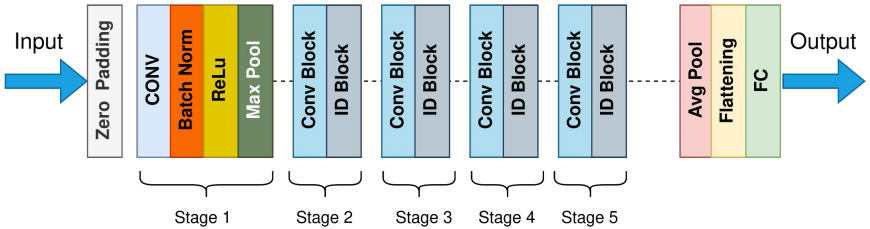
Working with images using TorchVision

Computer vision models

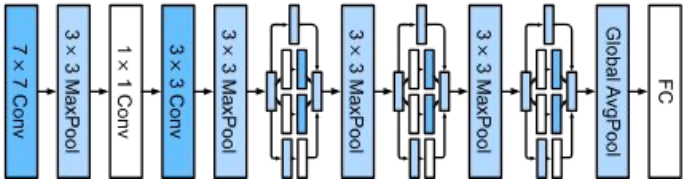
VGG16



ResNet50



GoogLeNet



Classification with a pretrained model

ResNet50

Pre-trained with ImageNet (1,000 classes)

DeepLearning.AI

Classification with a pretrained model

```
resnet50_model = tv_models.resnet50(pretrained=True).eval()

imagenet_classes = helper_utils.load_imagenet_classes('./imagenet_class_index.json')

img = Image.open(image_path)

transform = transforms.Compose([
   transforms.Resize(256),
   transforms.CenterCrop(224),
   transforms.ToTensor(),
   transforms.Normalize(mean=[0.485, 0.456, 0.406],
                        std=[0.229, 0.224, 0.225])
])

input_tensor = transform(img)
input_batch = input_tensor.unsqueeze(0)  # Add a batch dimension
```

Classification with a pretrained model

```
resnet50_model = tv_models.resnet50(pretrained=True).eval()

imagenet_classes = helper_utils.load_imagenet_classes('./imagenet_class_index.json')

img = Image.open(image_path)

transform = transforms.Compose([
   transforms.Resize(256),
   transforms.CenterCrop(224),
   transforms.ToTensor(),
   transforms.Normalize(mean=[0.485, 0.456, 0.406],
                        std=[0.229, 0.224, 0.225])
])

input_tensor = transform(img)
input_batch = input_tensor.unsqueeze(0)  # Add a batch dimension
```

Classification with a pretrained model

```
resnet50_model = tv_models.resnet50(pretrained=True).eval()

imagenet_classes = helper_utils.load_imagenet_classes('./imagenet_class_index.json')

img = Image.open(image_path)

transform = transforms.Compose([
   transforms.Resize(256),
   transforms.CenterCrop(224),
   transforms.ToTensor(),
   transforms.Normalize(mean=[0.485, 0.456, 0.406],
                        std=[0.229, 0.224, 0.225])
])

input_tensor = transform(img)
input_batch = input_tensor.unsqueeze(0)  # Add a batch dimension
```

Classification with a pretrained model

```
resnet50_model = tv_models.resnet50(pretrained=True).eval()

imagenet_classes = helper_utils.load_imagenet_classes('./imagenet_class_index.json')

img = Image.open(image_path)

transform = transforms.Compose([
   transforms.Resize(256),
   transforms.CenterCrop(224),
   transforms.ToTensor(),
   transforms.Normalize(mean=[0.485, 0.456, 0.406],
                        std=[0.229, 0.224, 0.225])
])

input_tensor = transform(img)
input_batch = input_tensor.unsqueeze(0)  # Add a batch dimension
```

Classification with a pretrained model

```
resnet50_model = tv_models.resnet50(pretrained=True).eval()

imagenet_classes = helper_utils.load_imagenet_classes('./imagenet_class_index.json')

img = Image.open(image_path)

transform = transforms.Compose([
   transforms.Resize(256),
   transforms.CenterCrop(224),
   transforms.ToTensor(),
   transforms.Normalize(mean=[0.485, 0.456, 0.406],
                        std=[0.229, 0.224, 0.225])
])

input_tensor = transform(img)
input_batch = input_tensor.unsqueeze(0)  # Add a batch dimension
```

```
with torch.no_grad():
    output = resnet50_model(input_batch)

probabilities = torch.nn.functional.softmax(output[0], dim=0)

top = 5
top_prob, top_catid = torch.topk(probabilities, top)

print(f"Top {top} predictions:")
for i in range(top_prob.size(0)):
    # Get the string representation of the class ID
    class_id_str = str(top_catid[i].item())

    # Look up the class name in the dictionary
    class_name = imagenet_classes[class_id_str][1]
    confidence = top_prob[i].item() * 100
    print(f"\tTop-{i+1}: {class_name} ({confidence:.2f}%)")
```

```
with torch.no_grad():
    output = resnet50_model(input_batch)

probabilities = torch.nn.functional.softmax(output[0], dim=0)

top = 5
top_prob, top_catid = torch.topk(probabilities, top)

print(f"Top {top} predictions:")
for i in range(top_prob.size(0)):
    # Get the string representation of the class ID
    class_id_str = str(top_catid[i].item())

    # Look up the class name in the dictionary
    class_name = imagenet_classes[class_id_str][1]
    confidence = top_prob[i].item() * 100
    print(f"\tTop-{i+1}: {class_name} ({confidence:.2f}%)")
```

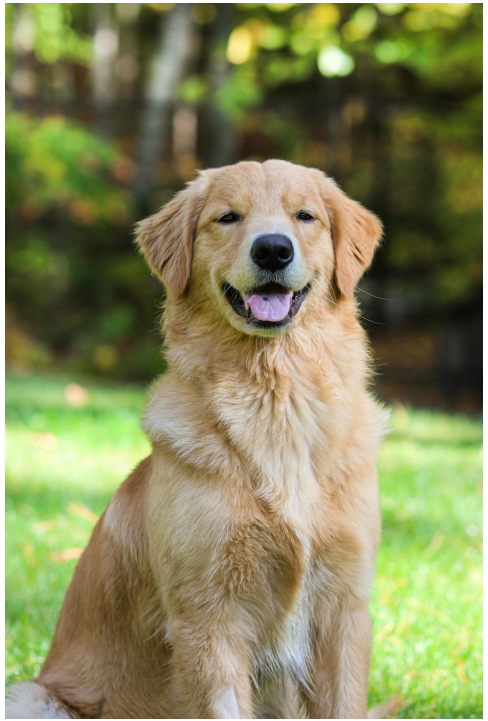
```
with torch.no_grad():
    output = resnet50_model(input_batch)

probabilities = torch.nn.functional.softmax(output[0], dim=0)

top = 5
top_prob, top_catid = torch.topk(probabilities, top)

print(f"Top {top} predictions:")
for i in range(top_prob.size(0)):
    # Get the string representation of the class ID
    class_id_str = str(top_catid[i].item())

    # Look up the class name in the dictionary
    class_name = imagenet_classes[class_id_str][1]
    confidence = top_prob[i].item() * 100
    print(f"\tTop-{i+1}: {class_name} ({confidence:.2f}%)")
```

```
with torch.no_grad():
    output = resnet50_model(input_batch)

probabilities = torch.nn.functional.softmax(output[0], dim=0)

top = 5
top_prob, top_catid = torch.topk(probabilities, top)

print(f"Top {top} predictions:")
for i in range(top_prob.size(0)):
    # Get the string representation of the class ID
    class_id_str = str(top_catid[i].item())

    # Look up the class name in the dictionary
    class_name = imagenet_classes[class_id_str][1]
    confidence = top_prob[i].item() * 100
    print(f"\tTop-{i+1}: {class_name} ({confidence:.2f}%)")
```

```
with torch.no_grad():
    output = resnet50_model(input_batch)

probabilities = torch.nn.functional.softmax(output[0], dim=0)

top = 5
top_prob, top_catid = torch.topk(probabilities, top)

print(f"Top {top} predictions:")
for i in range(top_prob.size(0)):
    # Get the string representation of the class ID
    class_id_str = str(top_catid[i].item())

    # Look up the class name in the dictionary
    class_name = imagenet_classes[class_id_str][1]
    confidence = top_prob[i].item() * 100
    print(f"\tTop-{i+1}: {class_name} ({confidence:.2f}%)")
```

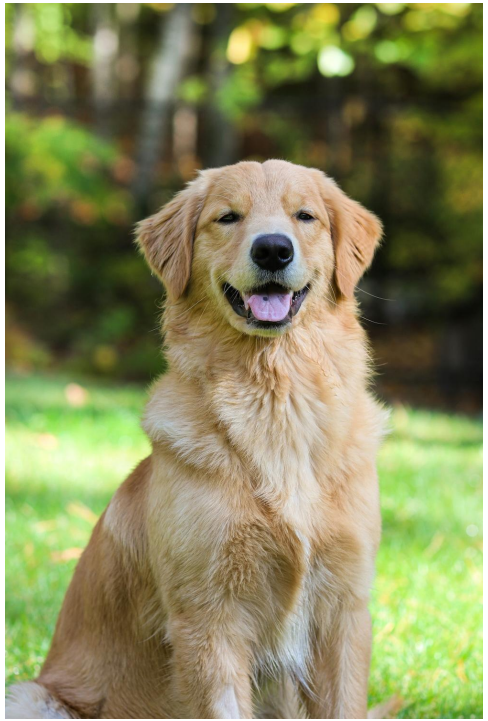
Classification results with a pretrained model

```
Top 5 predictions:
  Top-1: golden_retriever (96.83%)
  Top-2: Labrador_retriever (2.12%)
  Top-3: flat-coated_retriever (0.27%)
  Top-4: kuvasz (0.17%)
  Top-5: tennis_ball (0.08%)
```



Classification results with a pretrained model

```
Top 5 predictions:
  Top-1: golden_retriever (96.83%)
  Top-2: Labrador_retriever (2.12%)
  Top-3: flat-coated_retriever (0.27%)
  Top-4: kuvasz (0.17%)
  Top-5: tennis_ball (0.08%)
```



Determining class names

Manual approach

- Older models
- From JSON files

Determining class names

Manual approach

- Older models
- From JSON files

Modern approach

- Newer models
- Metadata in weights object

Loading a model with no metadata

```
resnet50_model = tv_models.resnet50(pretrained=True)

print(resnet50_model)

# Get the number of output features from the layer named 'fc'
num_classes = resnet50_model.fc.out_features

print(f"Inspecting the model's .fc layer: It has {num_classes} output classes.")
```

Loading a model with no metadata

```
resnet50_model = tv_models.resnet50(pretrained=True)

print(resnet50_model)

# Get the number of output features from the layer named 'fc'
num_classes = resnet50_model.fc.out_features

print(f"Inspecting the model's .fc layer: It has {num_classes} output classes.")
```

Loading a model with no metadata

```
resnet50_model = tv_models.resnet50(pretrained=True)

print(resnet50_model)

# Get the number of output features from the layer named 'fc'
num_classes = resnet50_model.fc.out_features

print(f"Inspecting the model's .fc layer: It has {num_classes} output classes.")
```

Loading a model with no metadata

Output:

```
...
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=2048, out_features=1000, bias=True)
)
```

Loading a model with no metadata

Output:

```
...
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=2048, out_features=1000, bias=True)
)
```

Loading a model with no metadata

```
resnet50_model = tv_models.resnet50(pretrained=True)

print(resnet50_model)

# Get the number of output features from the layer named 'fc'
num_classes = resnet50_model.fc.out_features

print(f"Inspecting the model's .fc layer: It has {num_classes} output classes.")
```

Loading a model with no metadata

Output:

```
Inspecting the model's .fc layer: It has 1000 output classes.
```

Finding class names

PyTorch documentation

Helper files from tutorials or repositories

Finding class names

```
imagenet_classes =
helper_utils.load_imagenet_classes('./imagenet
_class_index.json')
print("Total Classes:", len(imagenet_classes),
"\n")

start_index = 200
num_to_print = 10
print(f"Printing {num_to_print} classes
starting from index {start_index}:\n")

for i in range(start_index, start_index +
num_to_print):
    key = str(i)
    value = imagenet_classes[key]
    print(f"Index {key}: {value}")
```

Finding class names

```
imagenet_classes =
helper_utils.load_imagenet_classes('./imagenet
_class_index.json')
print("Total Classes:", len(imagenet_classes),
"\n")

start_index = 200
num_to_print = 10
print(f"Printing {num_to_print} classes
starting from index {start_index}:\n")

for i in range(start_index, start_index +
num_to_print):
    key = str(i)
    value = imagenet_classes[key]
    print(f"Index {key}: {value}")
```

Output

```
Total Classes: 1000

Printing 10 classes starting from index 200:

Index 200: ['n02097474', 'Tibetan_terrier']
Index 201: ['n02097658', 'silky_terrier']
Index 202: ['n02098105',
'soft-coated_wheaten_terrier']
Index 203: ['n02098286',
'West_Highland_white_terrier']
Index 204: ['n02098413', 'Lhasa']
Index 205: ['n02099267', 'flat-coated_retriever']
Index 206: ['n02099429', 'curly-coated_retriever']
Index 207: ['n02099601', 'golden_retriever']
Index 208: ['n02099712', 'Labrador_retriever']
Index 209: ['n02099849',
'Chesapeake_Bay_retriever']
```

Determining class names

Manual approach

- Older models
- From JSON files

Modern approach

- Newer models
- Metadata in weights object

```
def get_model_classes_from_weights_meta(model, weights_obj=None):

    num_classes = None
    class_names = None

    # Check if a weights object was provided and if it has the necessary metadata
    if weights_obj and hasattr(weights_obj, 'meta') and "categories" in weights_obj.meta:
        class_names = weights_obj.meta["categories"]
        num_classes = len(class_names)
        print(f"Model is configured for {num_classes} classes based on Weights Metadata.
These classes are:\n")
        # For nice printing, let's display the list
        print(class_names)

        return num_classes, class_names

    else:
        print("'categories' metadata not found for this model.")
        return num_classes, class_names
```

```
def get_model_classes_from_weights_meta(model, weights_obj=None):

    num_classes = None
    class_names = None

    # Check if a weights object was provided and if it has the necessary metadata
    if weights_obj and hasattr(weights_obj, 'meta') and "categories" in weights_obj.meta:
        class_names = weights_obj.meta["categories"]
        num_classes = len(class_names)
        print(f"Model is configured for {num_classes} classes based on Weights Metadata.
These classes are:\n")
        # For nice printing, let's display the list
        print(class_names)

        return num_classes, class_names

    else:
        print("'categories' metadata not found for this model.")
        return num_classes, class_names
```

```
def get_model_classes_from_weights_meta(model, weights_obj=None):

    num_classes = None
    class_names = None

    # Check if a weights object was provided and if it has the necessary metadata
    if weights_obj and hasattr(weights_obj, 'meta') and "categories" in weights_obj.meta:
        class_names = weights_obj.meta["categories"]
        num_classes = len(class_names)
        print(f"Model is configured for {num_classes} classes based on Weights Metadata.
These classes are:\n")
        # For nice printing, let's display the list
        print(class_names)

        return num_classes, class_names

    else:
        print("'categories' metadata not found for this model.")
        return num_classes, class_names
```

Loading a model with pretrained weights

```
from torchvision import models as tv_models

# Select your model architecture
seg_model = tv_models.segmentation.deeplabv3_resnet50(weights=None)

# Select the specific pre-trained weights you want to inspect
seg_model_weights = tv_models.segmentation.DeepLabV3_ResNet50_Weights.DEFAULT

num_classes, class_names = get_model_classes_from_weights_meta(
    model=seg_model,
    weights_obj=seg_model_weights
)
```

Loading a model with pretrained weights

```
from torchvision import models as tv_models

# Select your model architecture
seg_model = tv_models.segmentation.deeplabv3_resnet50(weights=None)

# Select the specific pre-trained weights you want to inspect
seg_model_weights = tv_models.segmentation.DeepLabV3_ResNet50_Weights.DEFAULT

num_classes, class_names = get_model_classes_from_weights_meta(
    model=seg_model,
    weights_obj=seg_model_weights
)
```

Loading a model with pretrained weights

```
from torchvision import models as tv_models

# Select your model architecture
seg_model = tv_models.segmentation.deeplabv3_resnet50(weights=None)

# Select the specific pre-trained weights you want to inspect
seg_model_weights = tv_models.segmentation.DeepLabV3_ResNet50_Weights.DEFAULT

num_classes, class_names = get_model_classes_from_weights_meta(
    model=seg_model,
    weights_obj=seg_model_weights
)
```

Loading a model with pretrained weights

```
from torchvision import models as tv_models

# Select your model architecture
seg_model = tv_models.segmentation.deeplabv3_resnet50(weights=None)

# Select the specific pre-trained weights you want to inspect
seg_model_weights = tv_models.segmentation.DeepLabV3_ResNet50_Weights.DEFAULT

num_classes, class_names_deeplabv3 = get_model_classes_from_weights_meta(
    model=seg_model,
    weights_obj=seg_model_weights
)
```

Loading a model with pretrained weights

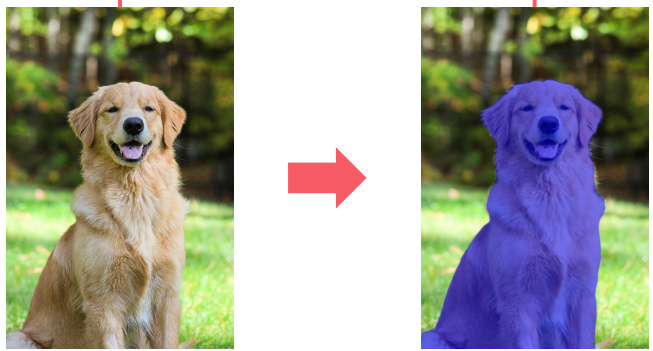
Output:

```
Model is configured for 21 classes based on Weights Metadata. These classes are:

['__background__', 'aeroplane', 'bicycle', 'bird', 'boat', 'bottle', 'bus', 'car',
'cat', 'chair', 'cow', 'diningtable', 'dog', 'horse', 'motorbike', 'person',
'pottedplant', 'sheep', 'sofa', 'train', 'tvmonitor']
```

Loading a model with pretrained weights

Output:

```
Model is configured for 21 classes based on Weights Metadata. These classes are:

['__background__', 'aeroplane', 'bicycle', 'bird', 'boat', 'bottle', 'bus', 'car',
'cat', 'chair', 'cow', 'diningtable', 'dog', 'horse', 'motorbike', 'person',
'pottedplant', 'sheep', 'sofa', 'train', 'tvmonitor']
```



DeepLearning.AI

Transfer Learning and Fine Tuning

Working with images using TorchVision

Common challenges in deep learning models

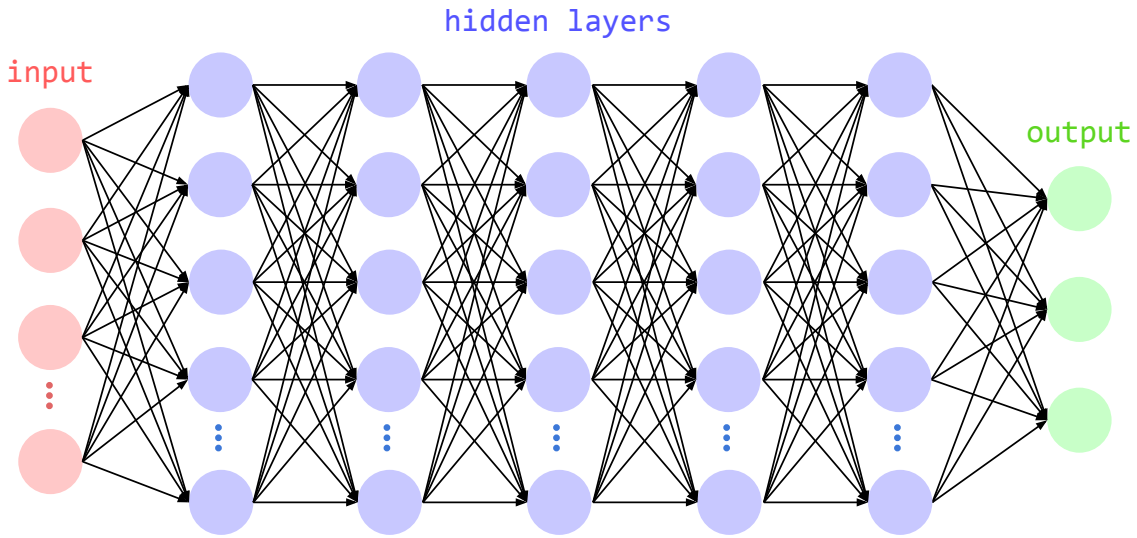
Not enough data

Common challenges in deep learning models

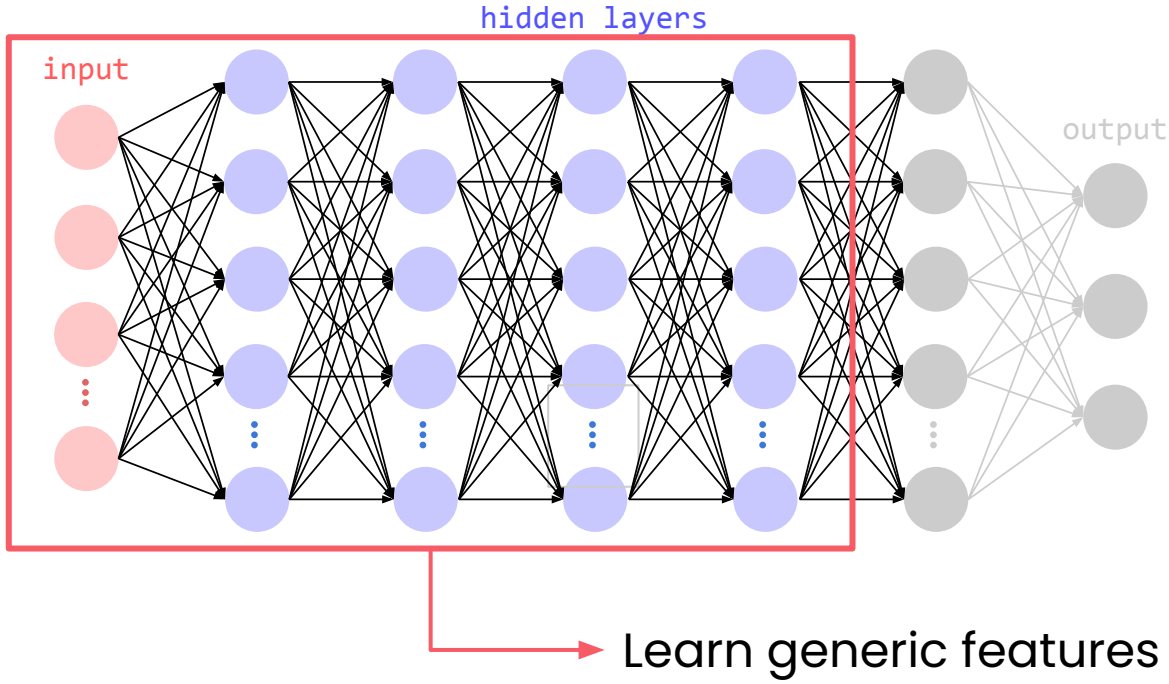
Not enough data

Not enough resources or time

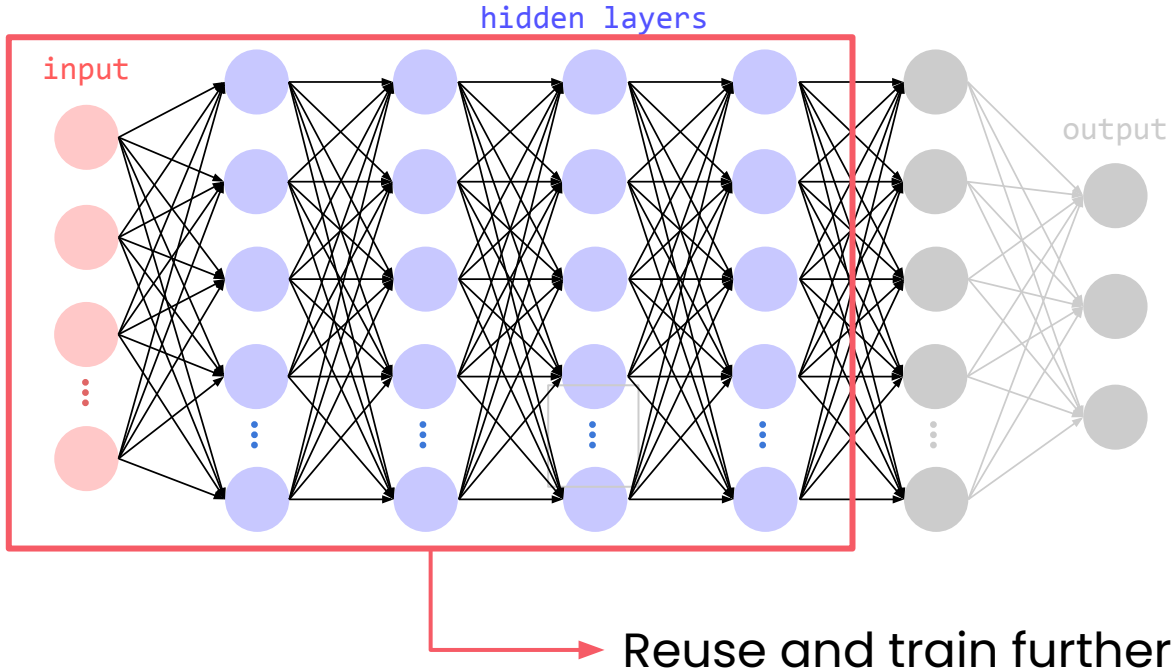
Transfer learning



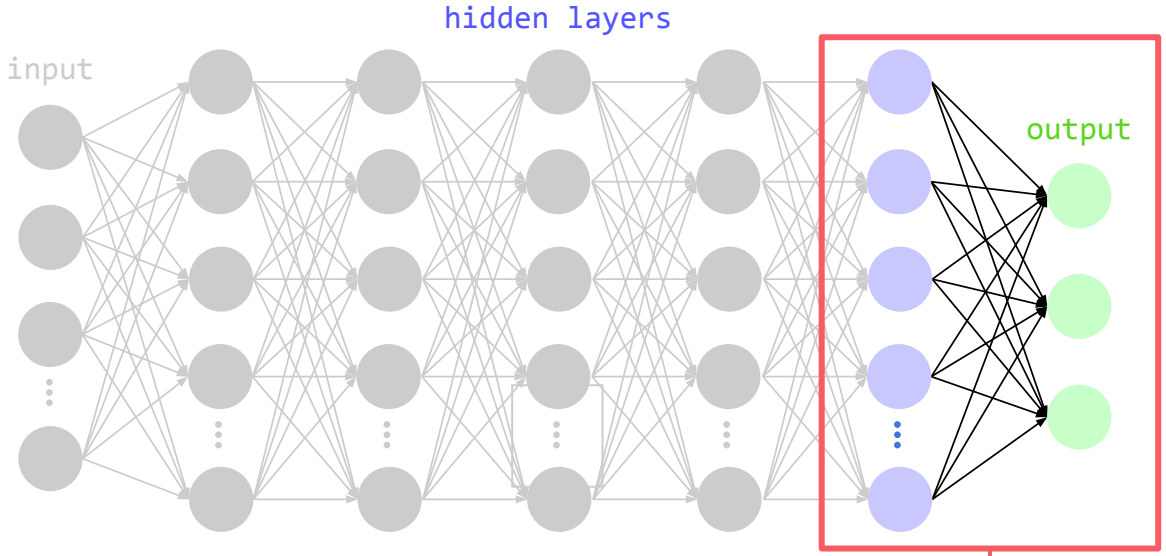
Transfer learning



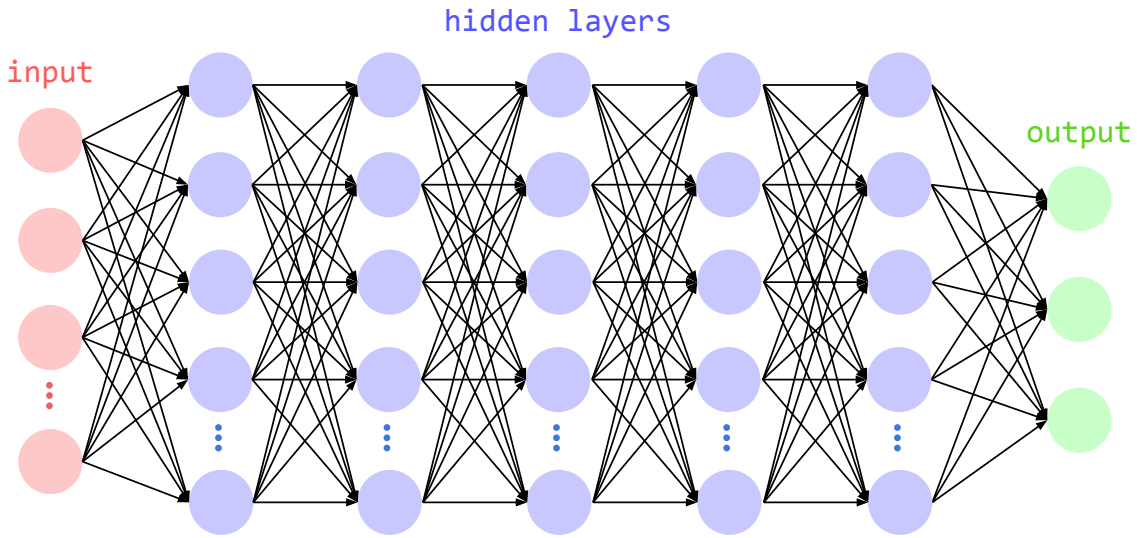
Transfer learning



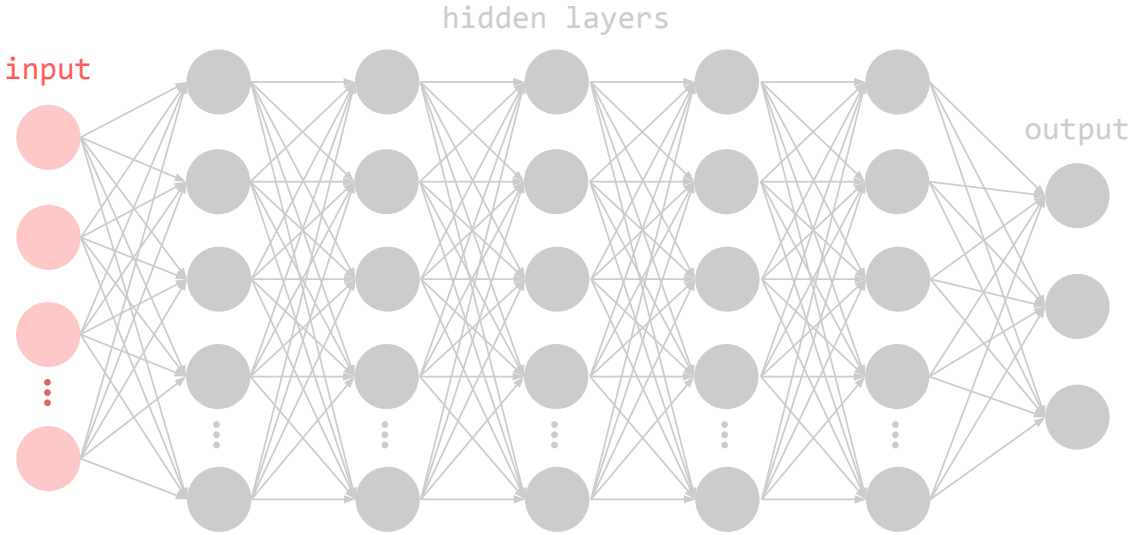
Transfer learning



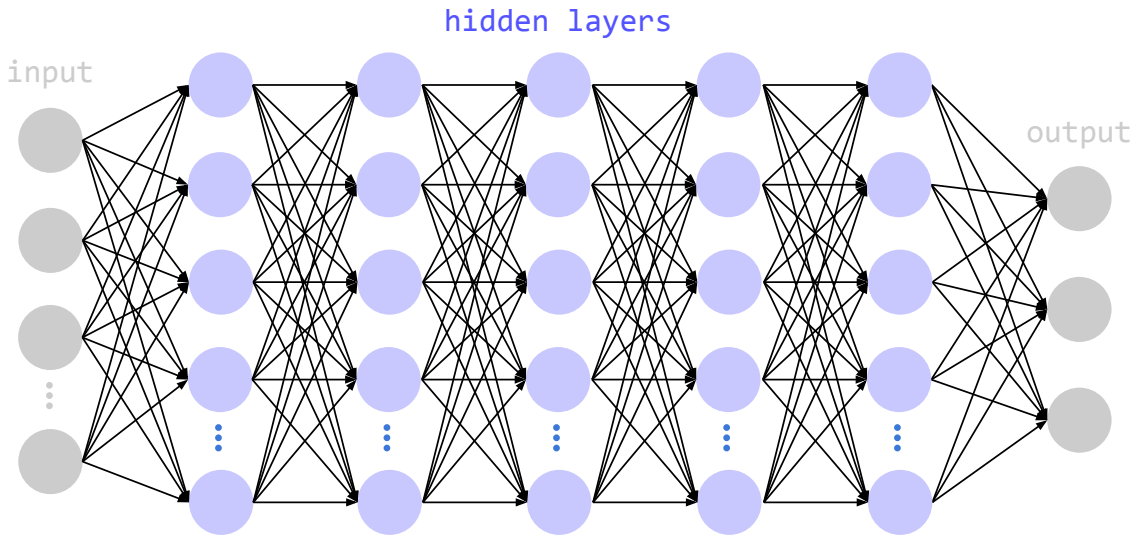
Transfer learning



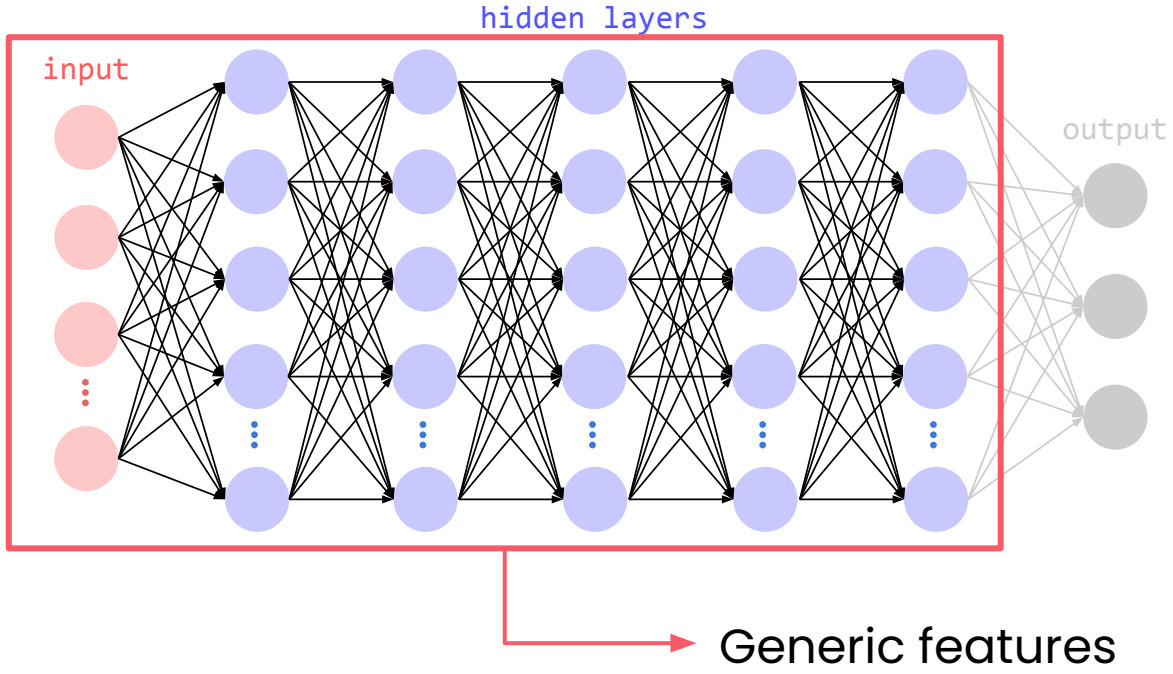
Transfer learning



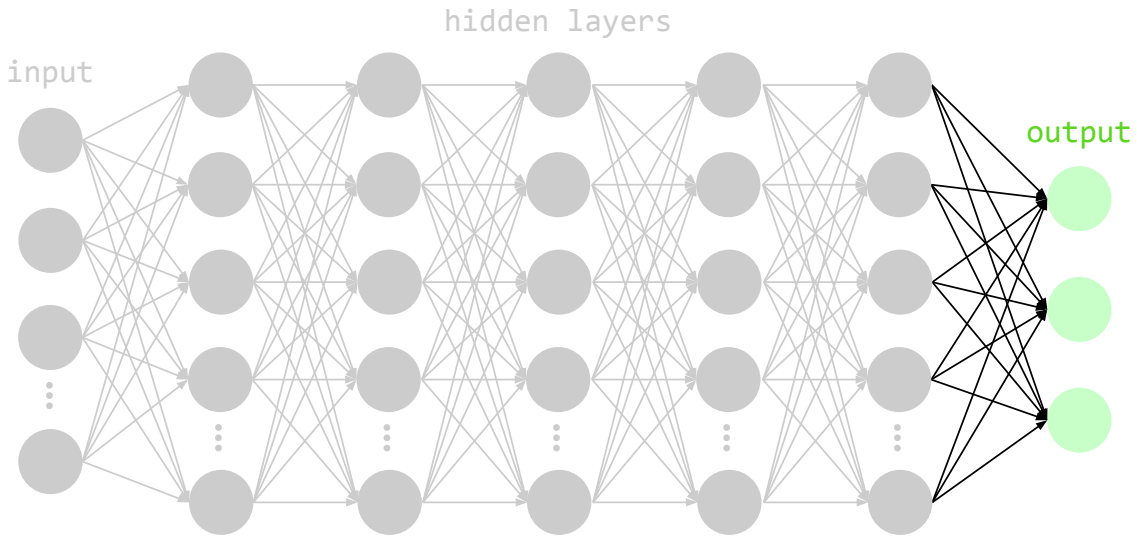
Transfer learning



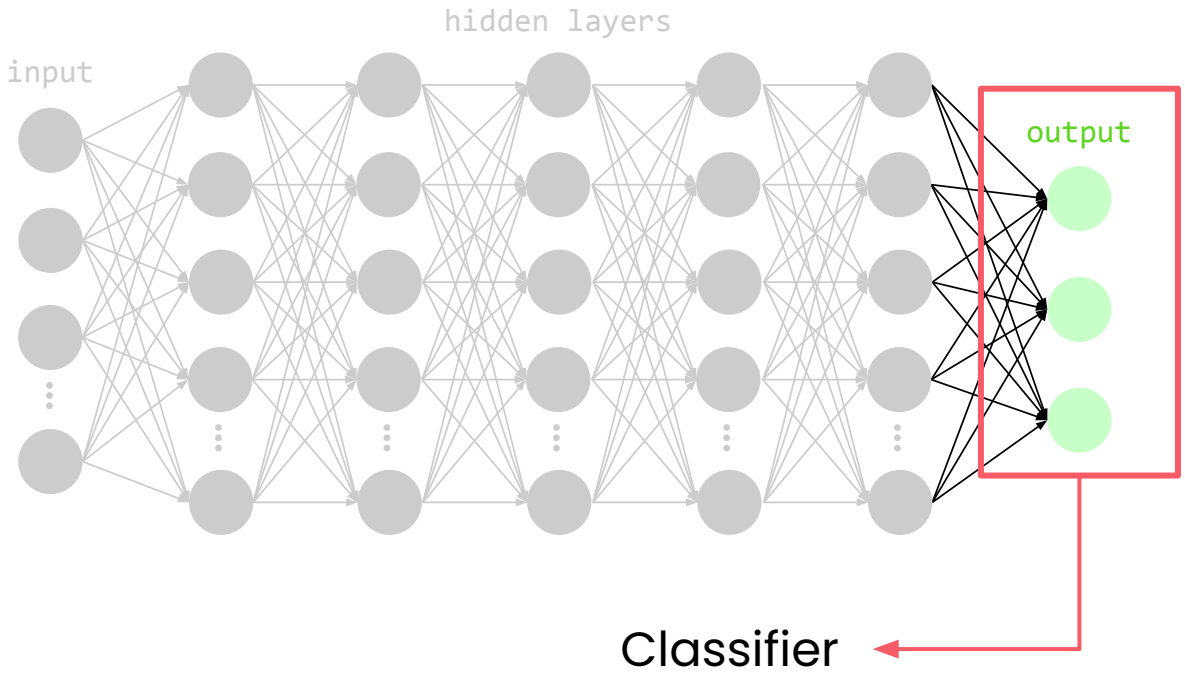
Transfer learning



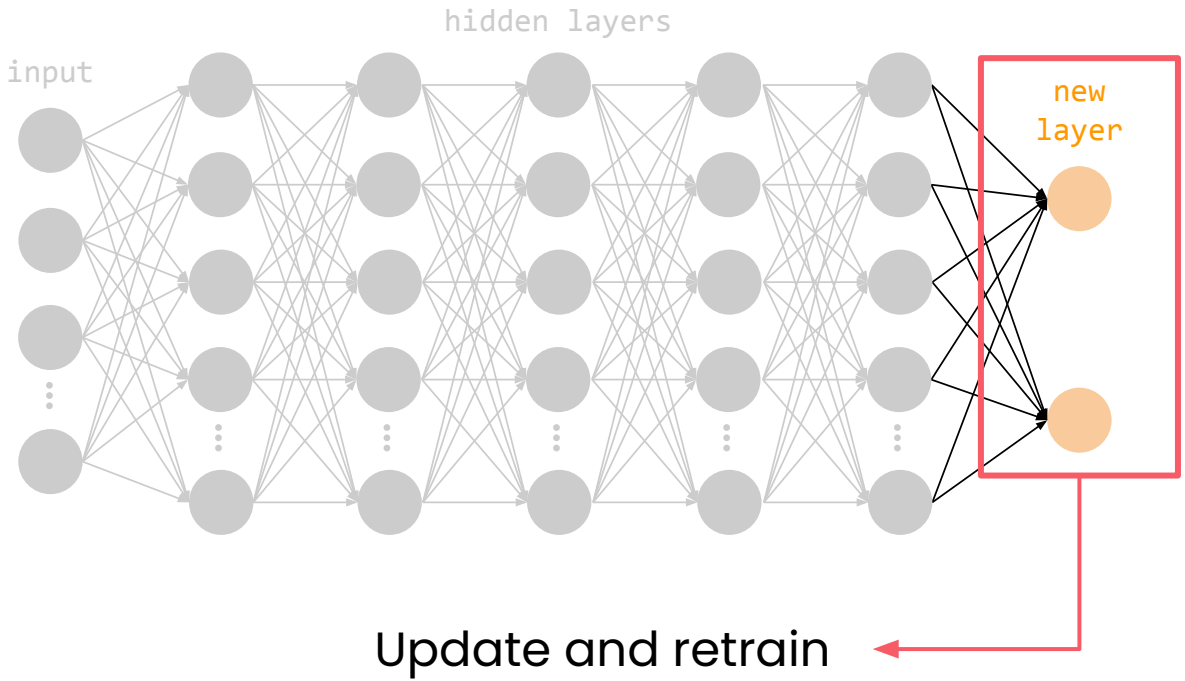
Transfer learning



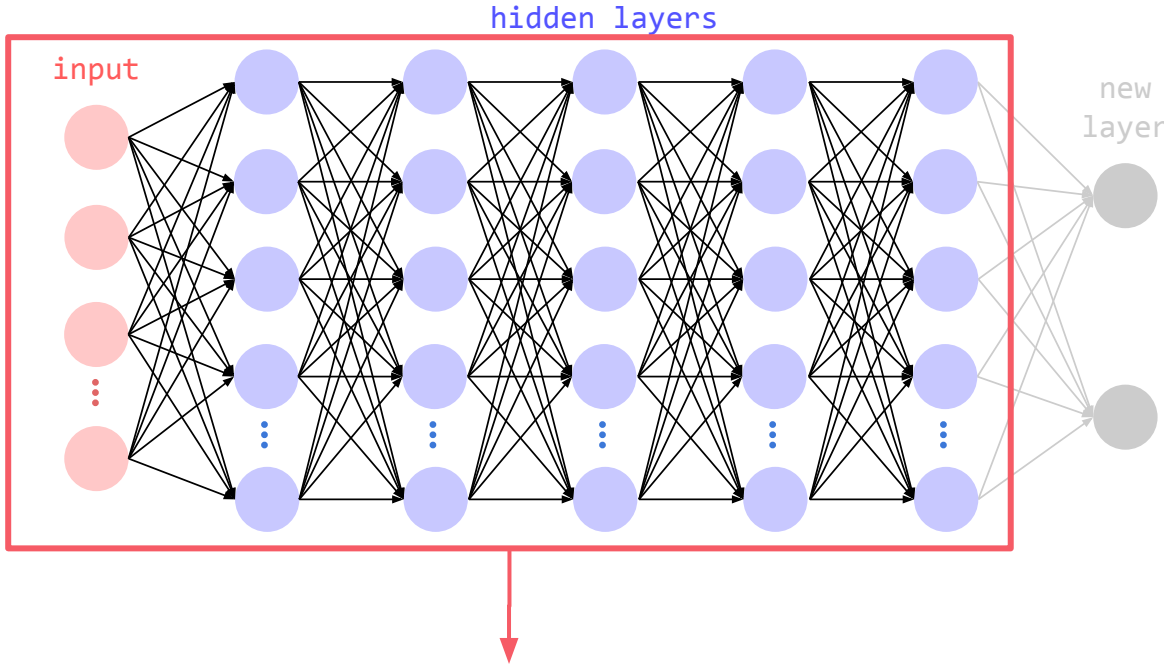
Transfer learning



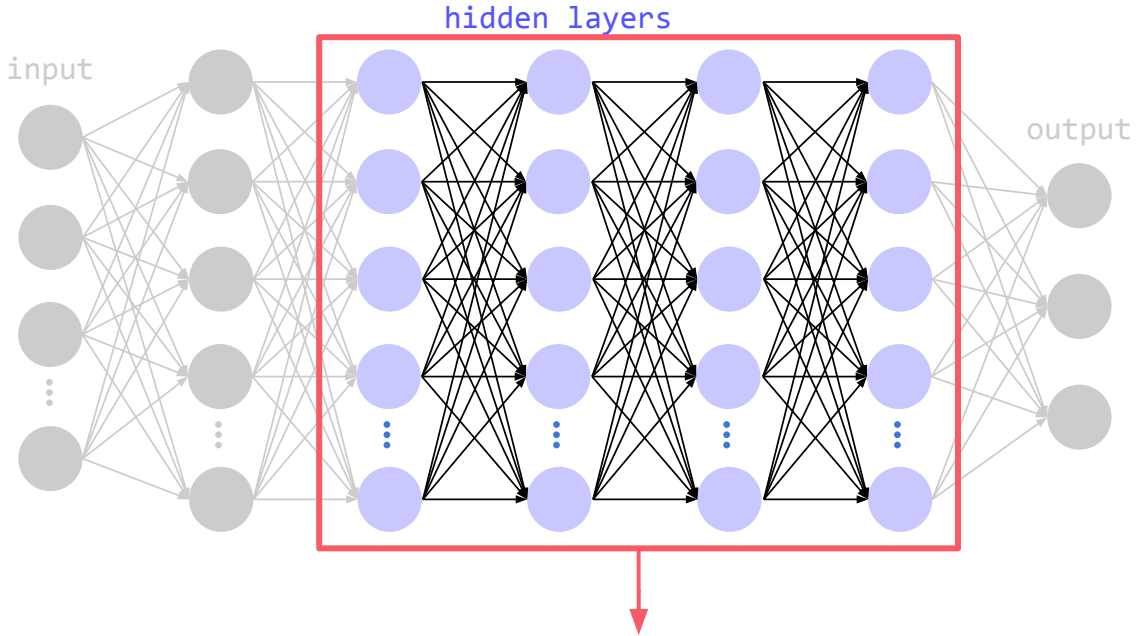
Transfer learning



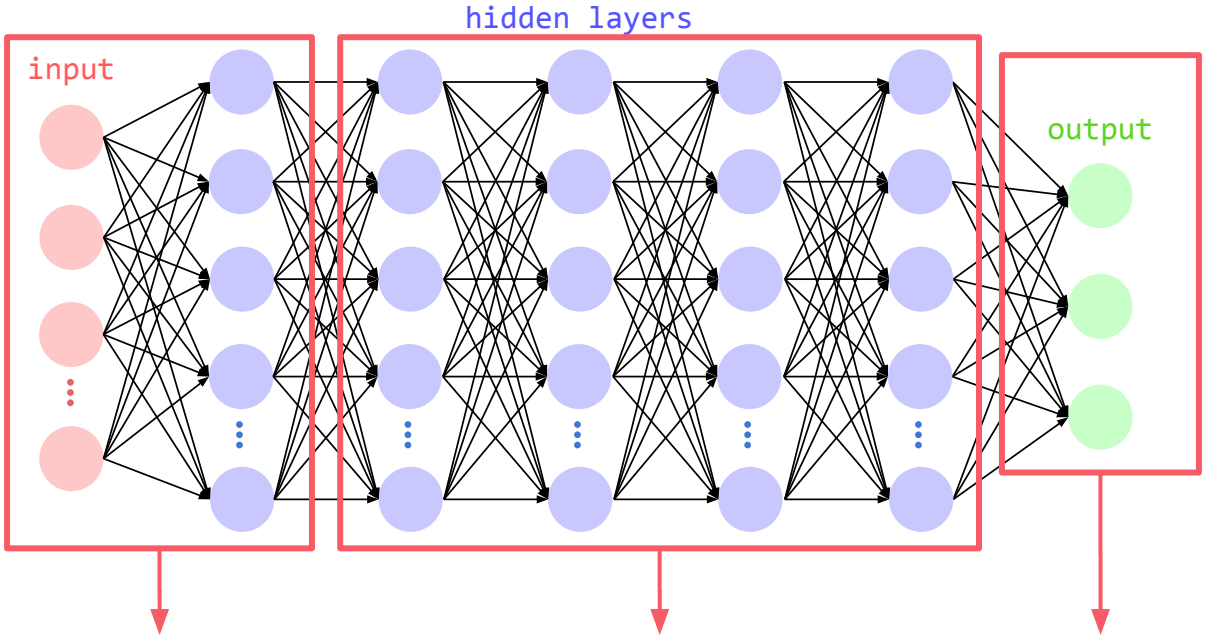
Transfer learning



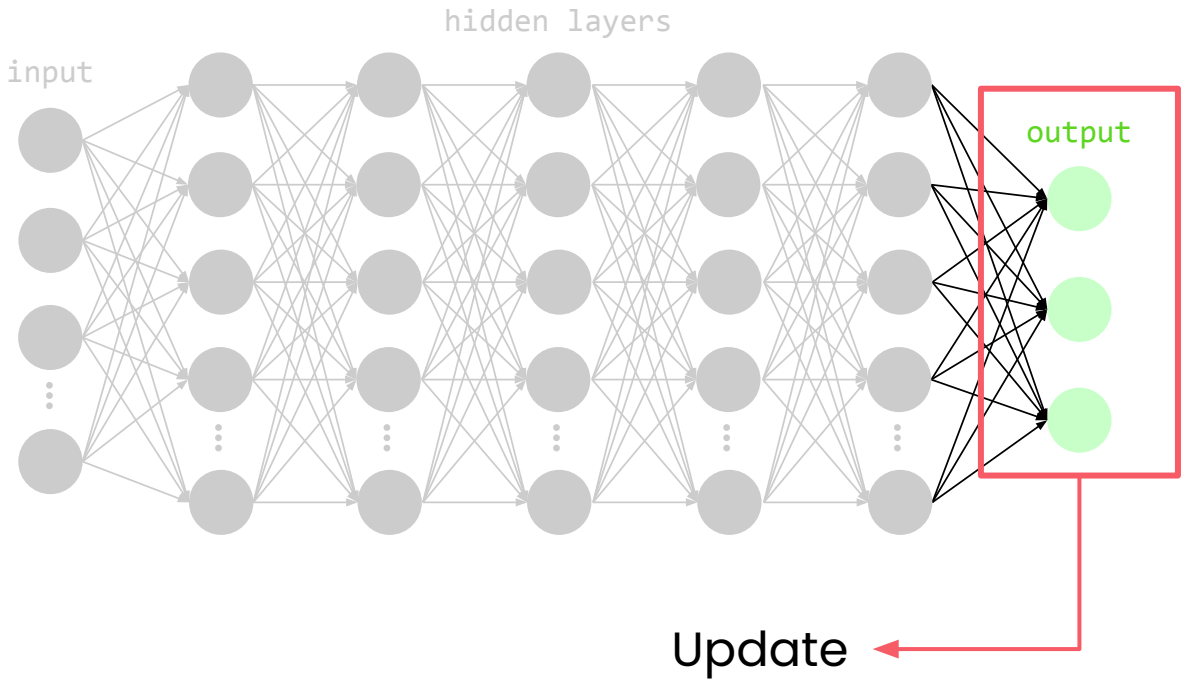
Fine tuning



Fine tuning


hidden layers
input
output
Freeze
Unfreeze and train further
Update and retrain

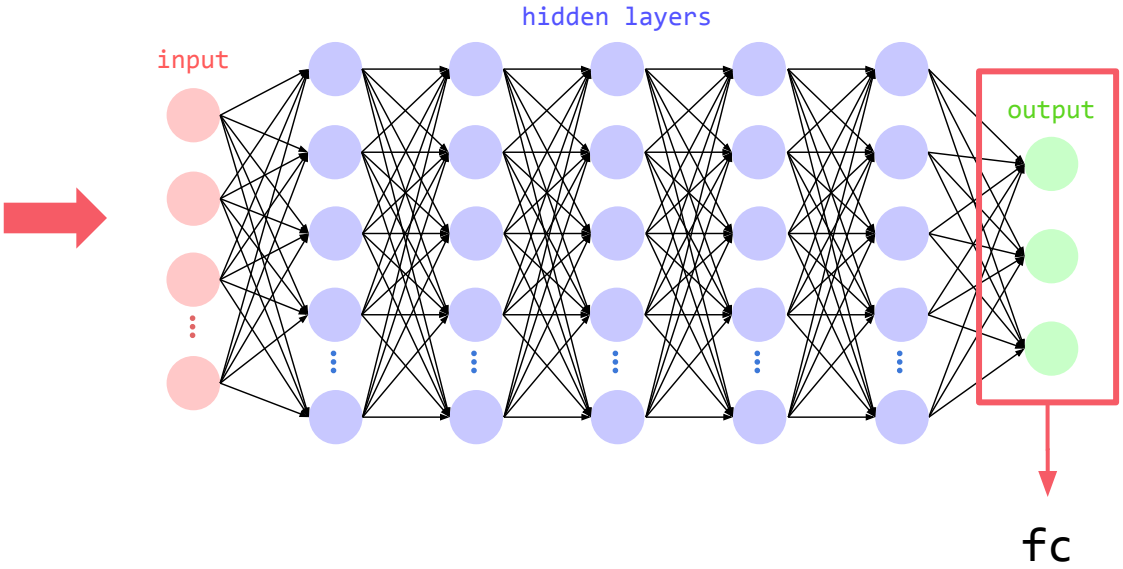
Transfer learning with TorchVision



Transfer learning with TorchVision

Direct attribute

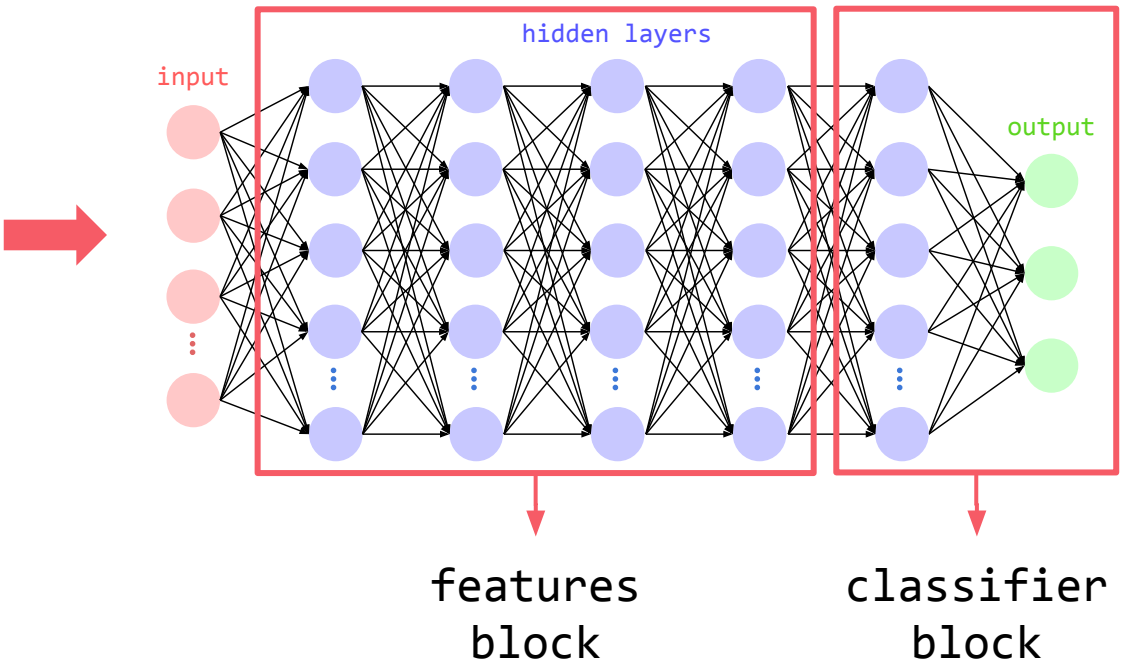
ResNet18



Transfer learning with TorchVision

Modular block

MobileNetV3



ResNet18

```
resnet18_model =
tv_models.resnet18(weights='IMAGENET1K_V1')

# Freeze the parameters of the feature layers
for param in resnet18_model.parameters():
    param.requires_grad = False

# Access the fully connected layer
original_fc_layer = resnet18_model.fc
```

MobileNetV3

```
mobilenet_model =
tv_models.mobilenet_v3_small(weights='IMAGENET1
K_V1')

# Freeze the parameters of the feature layers
for feature_parameter in
mobilenet_model.features.parameters():
   feature_parameter.requires_grad = False

# Access the final classification layer of the
model
last_classifier_layer =
mobilenet_model.classifier[-1]
```

ResNet18

```
resnet18_model = tv_models.resnet18(weights='IMAGENET1K_V1')

# Freeze the parameters of the feature layers
for param in resnet18_model.parameters():
    param.requires_grad = False

# Access the fully connected layer
original_fc_layer = resnet18_model.fc
```

MobileNetV3

```
mobilenet_model = tv_models.mobilenet_v3_small(weights='IMAGENET1K_V1')

# Freeze the parameters of the feature layers
for feature_parameter in mobilenet_model.features.parameters():
   feature_parameter.requires_grad = False

# Access the final classification layer of the model
last_classifier_layer = mobilenet_model.classifier[-1]
```

ResNet18

```
resnet18_model = tv_models.resnet18(weights='IMAGENET1K_V1')

# Freeze the parameters of the feature layers
for param in resnet18_model.parameters():
    param.requires_grad = False

# Access the fully connected layer
original_fc_layer = resnet18_model.fc
```

MobileNetV3

```
mobilenet_model = tv_models.mobilenet_v3_small(weights='IMAGENET1K_V1')

# Freeze the parameters of the feature layers
for feature_parameter in mobilenet_model.features.parameters():
   feature_parameter.requires_grad = False

# Access the final classification layer of the model
last_classifier_layer = mobilenet_model.classifier[-1]
```

ResNet18

```
resnet18_model =
tv_models.resnet18(weights='IMAGENET1K_V1')

# Freeze the parameters of the feature layers
for param in resnet18_model.parameters():
    param.requires_grad = False

# Access the fully connected layer
original_fc_layer = resnet18_model.fc
```

MobileNetV3

```
mobilenet_model =
tv_models.mobilenet_v3_small(weights='IMAGENET1
K_V1')

# Freeze the parameters of the feature layers
for feature_parameter in
mobilenet_model.features.parameters():
   feature_parameter.requires_grad = False

# Access the final classification layer of the
model
last_classifier_layer =
mobilenet_model.classifier[-1]
```

ResNet18

```
resnet18_model =
tv_models.resnet18(weights='IMAGENET1K_V1')

# Freeze the parameters of the feature layers
for param in resnet18_model.parameters():
    param.requires_grad = False

# Access the fully connected layer
original_fc_layer = resnet18_model.fc
```

MobileNetV3

```
mobilenet_model =
tv_models.mobilenet_v3_small(weights='IMAGENET1
K_V1')

# Freeze the parameters of the feature layers
for feature_parameter in
mobilenet_model.features.parameters():
   feature_parameter.requires_grad = False

# Access the final classification layer of the
model
last_classifier_layer =
mobilenet_model.classifier[-1]
```

ResNet18

```
resnet18_model =
tv_models.resnet18(weights='IMAGENET1K_V1')

# Freeze the parameters of the feature layers
for param in resnet18_model.parameters():
    param.requires_grad = False

# Access the fully connected layer
original_fc_layer = resnet18_model.fc
```

MobileNetV3

```
mobilenet_model =
tv_models.mobilenet_v3_small(weights='IMAGENET1
K_V1')

# Freeze the parameters of the feature layers
for feature_parameter in
mobilenet_model.features.parameters():
   feature_parameter.requires_grad = False

# Access the final classification layer of the
model
last_classifier_layer =
mobilenet_model.classifier[-1]
```

ResNet18

```
# Get the number of input features in fc layer
num_features = original_fc_layer.in_features

# Define the number of output classes
num_classes = 10

# Create a new fully connected layer
new_fc_layer =
nn.Linear(in_features=num_features,
out_features=num_classes)
```

```
# Replace the fc_layer
resnet18_model.fc = new_fc_layer
```

MobileNetV3

```
# Access the in_features attribute of
last_classifier_layer
num_features = last_classifier_layer.in_features

# Define the number of output classes
num_classes = 10

# Create a new classification layer
new_classifier =
nn.Linear(in_features=num_features,
out_features=num_classes)
```

```
# Replace the last classification layer
mobilenet_model.classifier[-1] = new_classifier
```

ResNet18

```
# Get the number of input features in fc layer
num_features = original_fc_layer.in_features

# Define the number of output classes
num_classes = 10

# Create a new fully connected layer
new_fc_layer =
nn.Linear(in_features=num_features,
out_features=num_classes)

# Replace the fc_layer
resnet18_model.fc = new_fc_layer
```

MobileNetV3

```
# Access the in_features attribute of
last_classifier_layer
num_features = last_classifier_layer.in_features

# Define the number of output classes
num_classes = 10

# Create a new classification layer
new_classifier =
nn.Linear(in_features=num_features,
out_features=num_classes)

# Replace the last classification layer
mobilenet_model.classifier[-1] = new_classifier
```

ResNet18

```
# Get the number of input features in fc layer
num_features = original_fc_layer.in_features

# Define the number of output classes
num_classes = 10

# Create a new fully connected layer
new_fc_layer =
nn.Linear(in_features=num_features,
out_features=num_classes)

# Replace the fc_layer
resnet18_model.fc = new_fc_layer
```

MobileNetV3

```
# Access the in_features attribute of
last_classifier_layer
num_features = last_classifier_layer.in_features

# Define the number of output classes
num_classes = 10

# Create a new classification layer
new_classifier =
nn.Linear(in_features=num_features,
out_features=num_classes)

# Replace the last classification layer
mobilenet_model.classifier[-1] = new_classifier
```

ResNet18

```
# Get the number of input features in fc layer
num_features = original_fc_layer.in_features

# Define the number of output classes
num_classes = 10

# Create a new fully connected layer
new_fc_layer =
nn.Linear(in_features=num_features,
out_features=num_classes)

# Replace the fc_layer
resnet18_model.fc = new_fc_layer
```

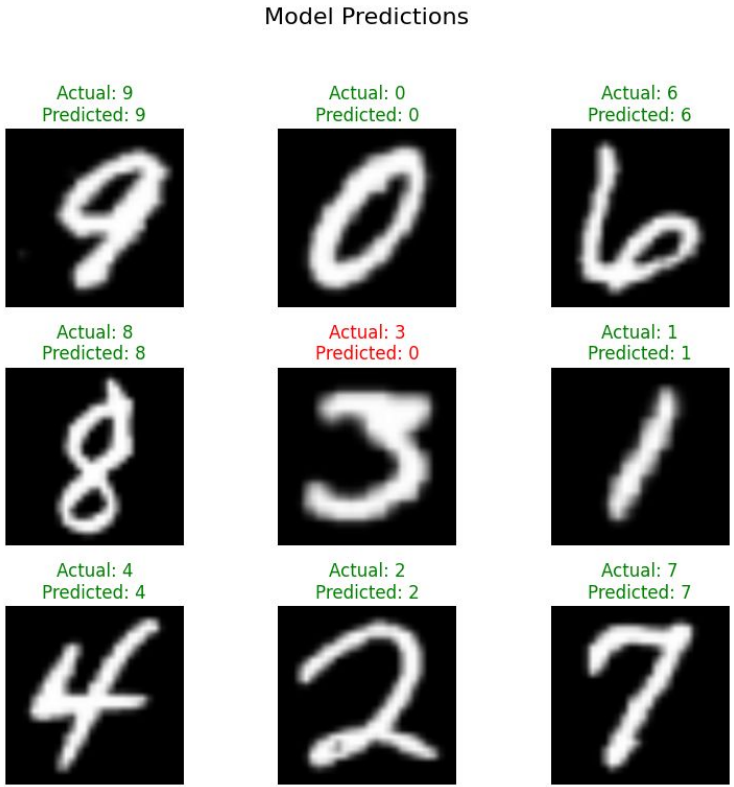
MobileNetV3

```
# Access the in_features attribute of
last_classifier_layer
num_features = last_classifier_layer.in_features

# Define the number of output classes
num_classes = 10

# Create a new classification layer
new_classifier =
nn.Linear(in_features=num_features,
out_features=num_classes)

# Replace the last classification layer
mobilenet_model.classifier[-1] = new_classifier
```

Transfer learning results - EMNIST digits data

```
Finished Training!

Final Validation Accuracy: 0.8238
```

Transfer learning results - EMNIST digits data



DeepLearning.AI

TorchVision Utility Functions for Visualization

Working with images using TorchVision

Visualization lets you inspect your data

- Check augmentation correctness
- See where the model is making predictions
- Measure confidence in specific regions
- Diagnose failures
- Fine-tune performance
- Build trust with users

TorchVision's visualization utilities

`draw_bounding_boxes`

Visualize object detection results

TorchVision's visualization utilities

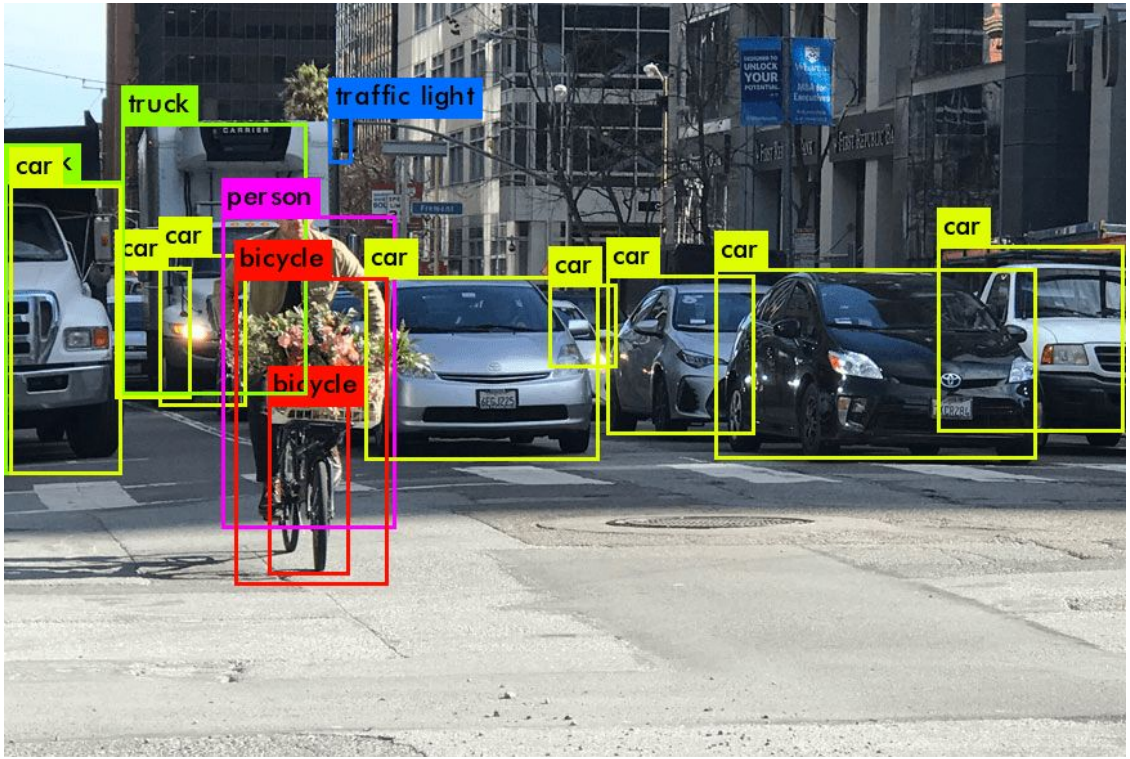
`draw_bounding_boxes`

Visualize object detection results

`draw_segmentation_masks`

Overlay segmentation masks on images

Bounding boxes



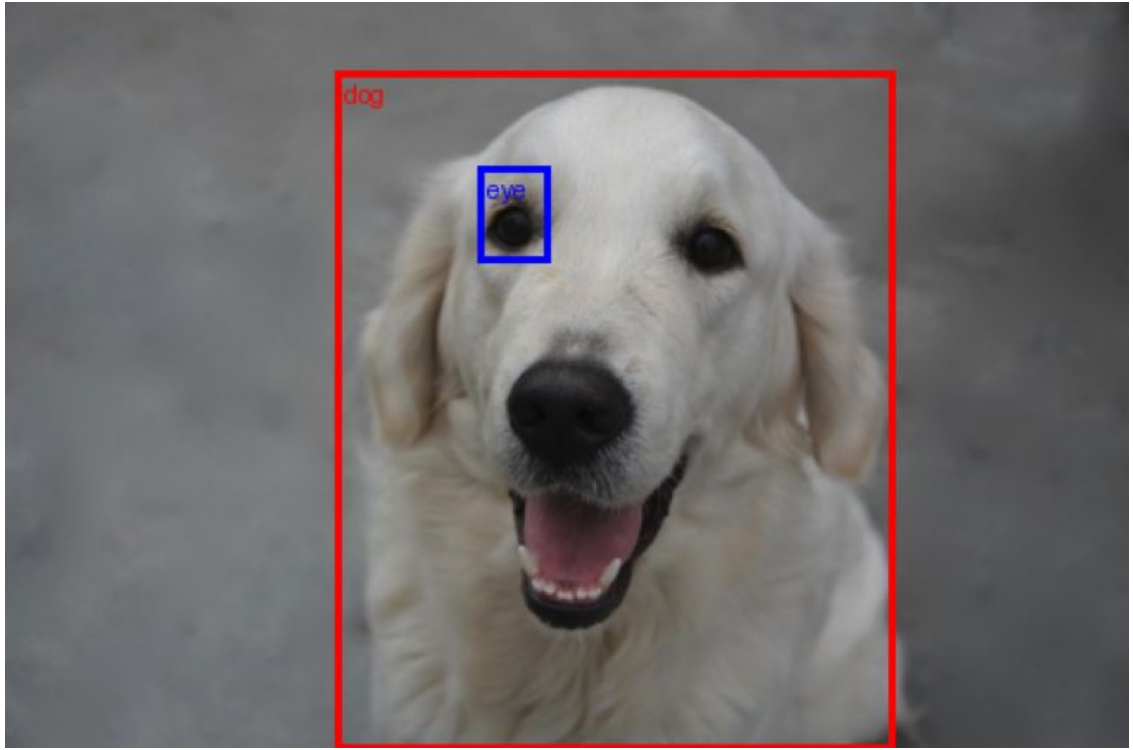
```
import torchvision.utils as vutils

# Load an image
image = decode_image('./images/dog1.jpg')

# Sample bounding boxes [x1, y1, x2, y2]
boxes = torch.tensor([[140, 30, 375, 315], [200, 70, 230, 110]], dtype=torch.float)
labels = ["dog", "eye"]

# Draw boxes on the image
result = vutils.draw_bounding_boxes(image=image,
                                    boxes=boxes,
                                    labels=labels,
                                    colors=["red", "blue"],
                                    width=3
                                   )
```

```
import torchvision.utils as vutils

# Load an image
image = decode_image('./images/dog1.jpg')

# Sample bounding boxes [x1, y1, x2, y2]
boxes = torch.tensor([[140, 30, 375, 315], [200, 70, 230, 110]], dtype=torch.float)
labels = ["dog", "eye"]

# Draw boxes on the image
result = vutils.draw_bounding_boxes(image=image,
                                    boxes=boxes,
                                    labels=labels,
                                    colors=["red", "blue"],
                                    width=3
                                   )
```

```
import torchvision.utils as vutils

# Load an image
image = decode_image('./images/dog1.jpg')

# Sample bounding boxes [x1, y1, x2, y2]
boxes = torch.tensor([[140, 30, 375, 315], [200, 70, 230, 110]], dtype=torch.float)
labels = ["dog", "eye"]

# Draw boxes on the image
result = vutils.draw_bounding_boxes(image=image,
                                    boxes=boxes,
                                    labels=labels,
                                    colors=["red", "blue"],
                                    width=3
                                   )
```

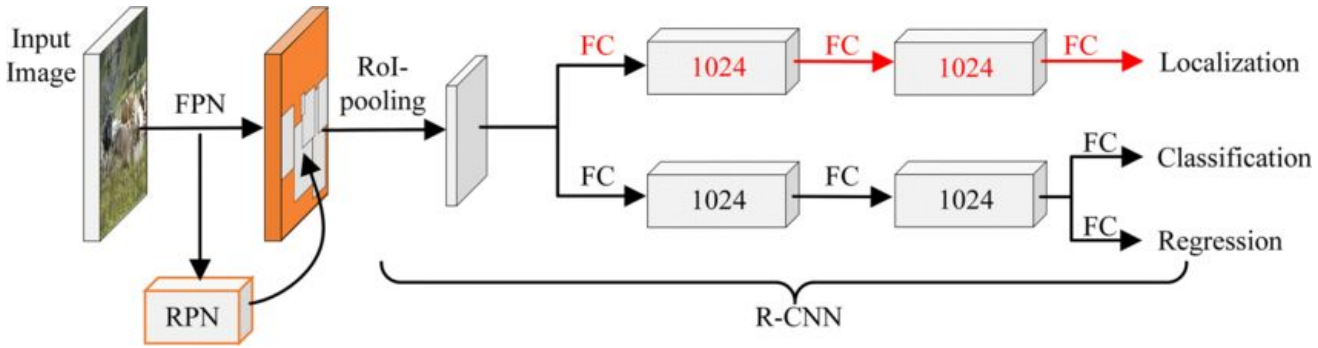
```
import torchvision.utils as vutils

# Load an image
image = decode_image('./images/dog1.jpg')

# Sample bounding boxes [x1, y1, x2, y2]
boxes = torch.tensor([[140, 30, 375, 315], [200, 70, 230, 110]], dtype=torch.float)
labels = ["dog", "eye"]

# Draw boxes on the image
result = vutils.draw_bounding_boxes(image=image,
                                    boxes=boxes,
                                    labels=labels,
                                    colors=["red", "blue"],
                                    width=3
                                   )
```

Drawing bounding boxes manually



Using a model to draw bounding boxes

- Faster R-CNN: for object detection
- ResNet-50: to learn rich features from images
- FPN: to detect objects at multiple scales



Using a model to draw bounding boxes

```
# Load a pre-trained object detection model and set to evaluation mode
bb_model_weights = tv_models.detection.FasterRCNN_ResNet50_FPN_Weights.DEFAULT
bb_model = tv_models.detection.fasterrcnn_resnet50_fpn(weights=bb_model_weights).eval()

# Use the helper function to inspect the weights object of the object detection model.
num_classes, classes = get_model_classes_from_weights_meta(
    model=bb_model,
    weights_obj=bb_model_weights
)
```

Using a model to draw bounding boxes

```
# Load a pre-trained object detection model and set to evaluation mode
bb_model_weights = tv_models.detection.FasterRCNN_ResNet50_FPN_Weights.DEFAULT
bb_model = tv_models.detection.fasterrcnn_resnet50_fpn(weights=bb_model_weights).eval()

# Use the helper function to inspect the weights object of the object detection model.
num_classes, classes = get_model_classes_from_weights_meta(
    model=bb_model,
    weights_obj=bb_model_weights
)
```

Using a model to draw bounding boxes

```
# Load a pre-trained object detection model and set to evaluation mode
bb_model_weights = tv_models.detection.FasterRCNN_ResNet50_FPN_Weights.DEFAULT
bb_model = tv_models.detection.fasterrcnn_resnet50_fpn(weights=bb_model_weights).eval()

# Use the helper function to inspect the weights object of the object detection model.
num_classes, classes = get_model_classes_from_weights_meta(
    model=bb_model,
    weights_obj=bb_model_weights
)
```

```
Model is configured for 91 classes based on Weights Metadata. These classes are:

[  '__background__', 'person', 'bicycle', 'car', 'motorcycle', 'airplane', 'bus',
   'train', 'truck', 'boat', 'traffic light', 'fire hydrant', 'N/A', 'stop sign',
   'parking meter', 'bench', 'bird', 'cat', 'dog', 'horse', 'sheep', 'cow',
   'elephant', 'bear', 'zebra', 'giraffe', 'N/A', 'backpack', 'umbrella', 'N/A',
   'N/A', 'handbag', 'tie', 'suitcase', 'frisbee', 'skis', 'snowboard', 'sports ball',
   'kite', 'baseball bat', 'baseball glove', 'skateboard', 'surfboard', 'tennis racket',
   'bottle', 'N/A', 'wine glass', 'cup', 'fork', 'knife', 'spoon', 'bowl',
   'banana', 'apple', 'sandwich', 'orange', 'broccoli', 'carrot', 'hot dog', 'pizza',
   'donut', 'cake', 'chair', 'couch', 'potted plant', 'bed', 'N/A', 'dining table',
   'N/A', 'N/A', 'toilet', 'N/A', 'tv', 'laptop', 'mouse', 'remote', 'keyboard',
   'cell phone', 'microwave', 'oven', 'toaster', 'sink', 'refrigerator', 'N/A',
   'book', 'clock', 'vase', 'scissors', 'teddy bear', 'hair drier', 'toothbrush']
```

```
Model is configured for 91 classes based on Weights Metadata. These classes are:

[  '__background__', 'person', 'bicycle', 'car', 'motorcycle', 'airplane', 'bus',
   'train', 'truck', 'boat', 'traffic light', 'fire hydrant', 'N/A', 'stop sign',
   'parking meter', 'bench', 'bird', 'cat', 'dog', 'horse', 'sheep', 'cow',
   'elephant', 'bear', 'zebra', 'giraffe', 'N/A', 'backpack', 'umbrella', 'N/A',
   'N/A', 'handbag', 'tie', 'suitcase', 'frisbee', 'skis', 'snowboard', 'sports ball',
   'kite', 'baseball bat', 'baseball glove', 'skateboard', 'surfboard', 'tennis racket',
   'bottle', 'N/A', 'wine glass', 'cup', 'fork', 'knife', 'spoon', 'bowl',
   'banana', 'apple', 'sandwich', 'orange', 'broccoli', 'carrot', 'hot dog', 'pizza',
   'donut', 'cake', 'chair', 'couch', 'potted plant', 'bed', 'N/A', 'dining table',
   'N/A', 'N/A', 'toilet', 'N/A', 'tv', 'laptop', 'mouse', 'remote', 'keyboard',
   'cell phone', 'microwave', 'oven', 'toaster', 'sink', 'refrigerator', 'N/A',
   'book', 'clock', 'vase', 'scissors', 'teddy bear', 'hair drier', 'toothbrush']
```

```
Model is configured for 91 classes based on Weights Metadata. These classes are:

[  '__background__', 'person', 'bicycle', 'car', 'motorcycle', 'airplane', 'bus',
   'train', 'truck', 'boat', 'traffic light', 'fire hydrant', 'N/A', 'stop sign',
   'parking meter', 'bench', 'bird', 'cat', 'dog', 'horse', 'sheep', 'cow',
   'elephant', 'bear', 'zebra', 'giraffe', 'N/A', 'backpack', 'umbrella', 'N/A',
   'N/A', 'handbag', 'tie', 'suitcase', 'frisbee', 'skis', 'snowboard', 'sports ball',
   'kite', 'baseball bat', 'baseball glove', 'skateboard', 'surfboard', 'tennis racket',
   'bottle', 'N/A', 'wine glass', 'cup', 'fork', 'knife', 'spoon', 'bowl',
   'banana', 'apple', 'sandwich', 'orange', 'broccoli', 'carrot', 'hot dog', 'pizza',
   'donut', 'cake', 'chair', 'couch', 'potted plant', 'bed', 'N/A', 'dining table',
   'N/A', 'N/A', 'toilet', 'N/A', 'tv', 'laptop', 'mouse', 'remote', 'keyboard',
   'cell phone', 'microwave', 'oven', 'toaster', 'sink', 'refrigerator', 'N/A',
   'book', 'clock', 'vase', 'scissors', 'teddy bear', 'hair drier', 'toothbrush']
```

Using a model to draw bounding boxes

Using a model to draw bounding boxes

```
# Define a list of target classes to detect
target_class_names = ['car', 'traffic light']

# Define a corresponding list of colors for each class's bounding box
bbox_colors = ['red', 'blue']

# Use a list comprehension to get a list of all target indices
object_indices = [classes.index(name) for name in target_class_names]
```

Using a model to draw bounding boxes

```
# Define a list of target classes to detect
target_class_names = ['car', 'traffic light']

# Define a corresponding list of colors for each class's bounding box
bbox_colors = ['red', 'blue']

# Use a list comprehension to get a list of all target indices
object_indices = [classes.index(name) for name in target_class_names]
```

Using a model to draw bounding boxes

```
# Define a list of target classes to detect
target_class_names = ['car', 'traffic light']

# Define a corresponding list of colors for each class's bounding box
bbox_colors = ['red', 'blue']

# Use a list comprehension to get a list of all target indices
object_indices = [classes.index(name) for name in target_class_names]
```

```
def detect_and_draw_bboxes(model, image_path, object_indices, labels, bbox_colors,
threshold, bbox_width=3):

    # Open and transform the image, and prepare the result tensor
    pil_image = Image.open(image_path).convert("RGB")
    transform_to_tensor = transforms.Compose([transforms.ToTensor()])
    tensor_image_batch = transform_to_tensor(pil_image).unsqueeze(0)
    result_image_tensor = (tensor_image_batch.squeeze(0) * 255).byte()

    # Perform inference to get predictions for all possible objects
    with torch.no_grad():
        prediction = model(tensor_image_batch)[0]

    # Initialize lists to collect all boxes, labels, and colors that meet the criteria
    all_boxes_to_draw = []
    all_labels_to_draw = []
    all_colors_to_draw = []
```

```
def detect_and_draw_bboxes(model, image_path, object_indices, labels, bbox_colors,
threshold, bbox_width=3):

    # Open and transform the image, and prepare the result tensor
    pil_image = Image.open(image_path).convert("RGB")
    transform_to_tensor = transforms.Compose([transforms.ToTensor()])
    tensor_image_batch = transform_to_tensor(pil_image).unsqueeze(0)
    result_image_tensor = (tensor_image_batch.squeeze(0) * 255).byte()

    # Perform inference to get predictions for all possible objects
    with torch.no_grad():
        prediction = model(tensor_image_batch)[0]

    # Initialize lists to collect all boxes, labels, and colors that meet the criteria
    all_boxes_to_draw = []
    all_labels_to_draw = []
    all_colors_to_draw = []
```

```
def detect_and_draw_bboxes(model, image_path, object_indices, labels, bbox_colors,
threshold, bbox_width=3):

    # Open and transform the image, and prepare the result tensor
    pil_image = Image.open(image_path).convert("RGB")
    transform_to_tensor = transforms.Compose([transforms.ToTensor()])
    tensor_image_batch = transform_to_tensor(pil_image).unsqueeze(0)
    result_image_tensor = (tensor_image_batch.squeeze(0) * 255).byte()

    # Perform inference to get predictions for all possible objects
    with torch.no_grad():
        prediction = model(tensor_image_batch)[0]

    # Initialize lists to collect all boxes, labels, and colors that meet the criteria
    all_boxes_to_draw = []
    all_labels_to_draw = []
    all_colors_to_draw = []
```

```
# Loop through each target class to find its boxes
for index, label, color in zip(object_indices, labels, bbox_colors):
    # Filter predictions for the current class index and confidence threshold
    class_mask = (prediction['labels'] == index) & (prediction['scores'] > threshold)

    # Get the boxes for the current class
    boxes_for_this_class = prediction['boxes'][class_mask]

    if boxes_for_this_class.nelement() > 0:
        # Add the found boxes to our master list
        all_boxes_to_draw.extend(boxes_for_this_class.tolist())
        # Create and add corresponding labels and colors
        all_labels_to_draw.extend([label] * len(boxes_for_this_class))
        all_colors_to_draw.extend([color] * len(boxes_for_this_class))
```

```
    # Loop through each target class to find its boxes
    for index, label, color in zip(object_indices, labels, bbox_colors):
        # Filter predictions for the current class index and confidence threshold
        class_mask = (prediction['labels'] == index) & (prediction['scores'] > threshold)

        # Get the boxes for the current class
        boxes_for_this_class = prediction['boxes'][class_mask]

        if boxes_for_this_class.nelement() > 0:
            # Add the found boxes to our master list
            all_boxes_to_draw.extend(boxes_for_this_class.tolist())
            # Create and add corresponding labels and colors
            all_labels_to_draw.extend([label] * len(boxes_for_this_class))
            all_colors_to_draw.extend([color] * len(boxes_for_this_class))
```

```
    # After checking all classes, draw all collected boxes at once if any were found
    if all_boxes_to_draw:
        result_image_tensor = vutils.draw_bounding_boxes(
            result_image_tensor,
            torch.tensor(all_boxes_to_draw),
            labels=all_labels_to_draw,
            colors=all_colors_to_draw,
            width=bbox_width
        )
    else:
        # If the list of boxes to draw is empty, print this information.
        print(f"No objects from the list {labels} were found with a confidence score above
{threshold}.\n")

    return result_image_tensor
```

```
    # After checking all classes, draw all collected boxes at once if any were found
    if all_boxes_to_draw:
        result_image_tensor = vutils.draw_bounding_boxes(
            result_image_tensor,
            torch.tensor(all_boxes_to_draw),
            labels=all_labels_to_draw,
            colors=all_colors_to_draw,
            width=bbox_width
        )
    else:
        # If the list of boxes to draw is empty, print this information.
        print(f"No objects from the list {labels} were found with a confidence score above
{threshold}.\n")

    return result_image_tensor
```

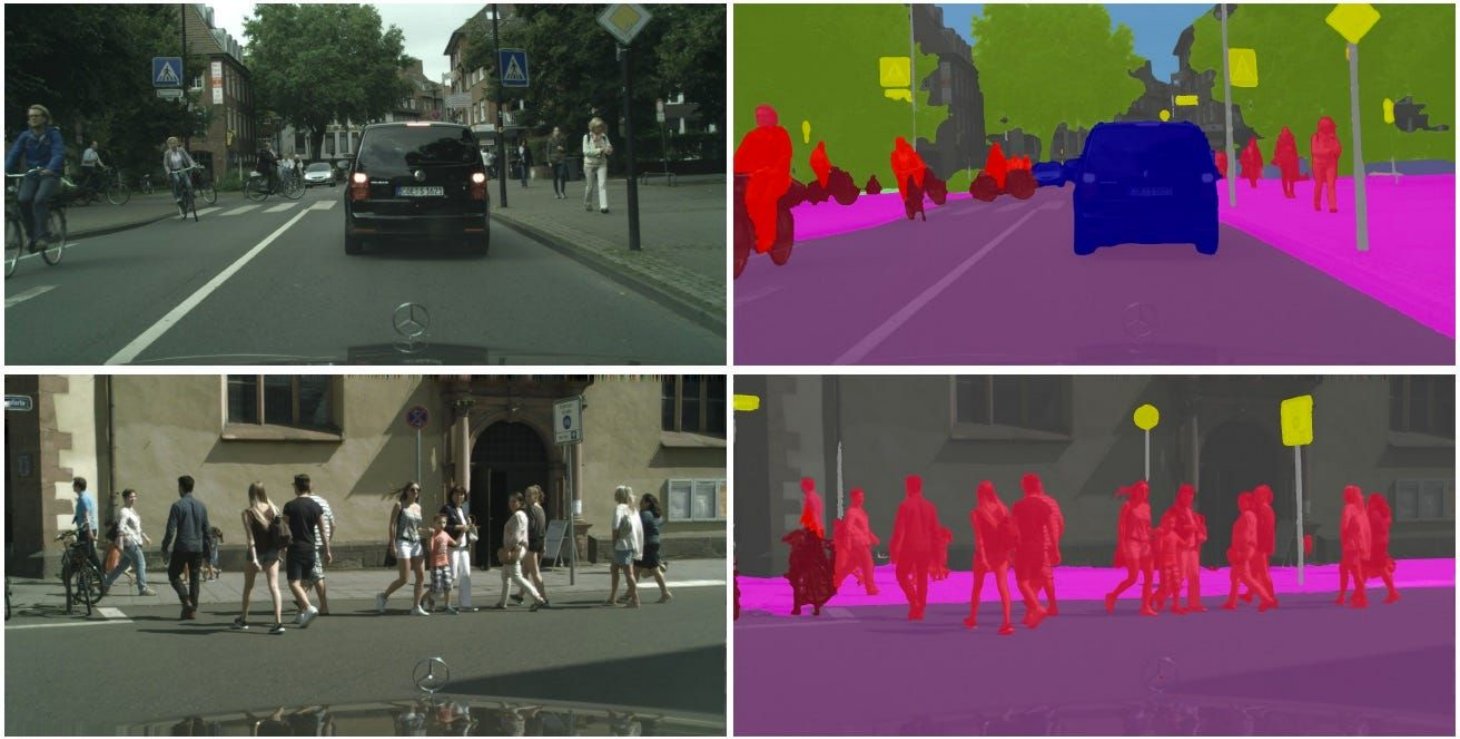
```
    # After checking all classes, draw all collected boxes at once if any were found
    if all_boxes_to_draw:
        result_image_tensor = vutils.draw_bounding_boxes(
            result_image_tensor,
            torch.tensor(all_boxes_to_draw),
            labels=all_labels_to_draw,
            colors=all_colors_to_draw,
            width=bbox_width
        )
    else:
        # If the list of boxes to draw is empty, print this information.
        print(f"No objects from the list {labels} were found with a confidence score above
{threshold}.\n")

    return result_image_tensor
```

Using a model to draw bounding boxes

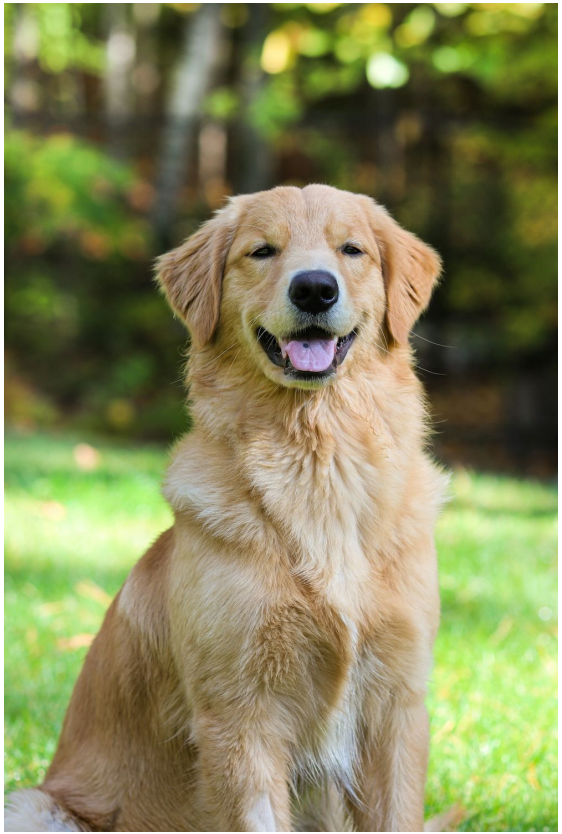
```
confidence_threshold = 0.7

# Execute the main detection function
result_image_tensor = detect_and_draw_bboxes(
    model=bb_model,                  # The pre-trained object detection model.
    image_path=image_path,           # The path to the input image.
    object_indices=object_indices,   # The list of integer indices for the target classes.
    labels=target_class_names,       # The list of string names for the box labels.
    bbox_colors=bbox_colors,         # The list of colors for the bounding boxes.
    threshold=confidence_threshold,  # The minimum confidence score for a detection.
)
```


PENINSULA
EYE SURGERY
CENTER
Daily
News
raffic light
raffic sign

Segmentation

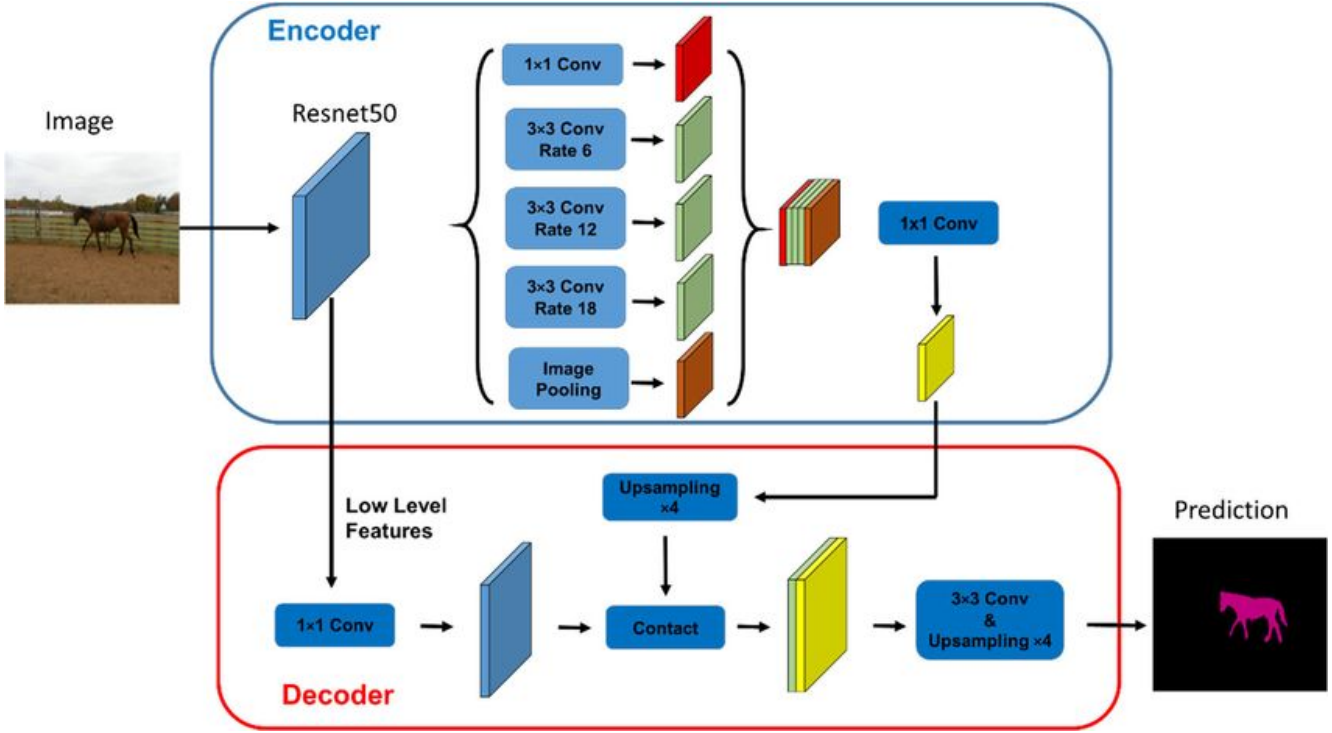


Segmentation



Laurence Moroney

DeepLabV3 with ResNet-50 for segmentation



```
import torchvision.utils as vutils
from torchvision import models as tv_models

# Instantiate the model architecture and load the pre-trained weights.
seg_model = tv_models.segmentation.deeplabv3_resnet50(weights=seg_model_weights).eval()

# Load the base PIL Image
img = Image.open(image_path)

# Create the clean, un-normalized tensor for visualization later
original_image_tensor = transforms.ToTensor()(img)

# Define the normalization transform
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])

# .unsqueeze(0) adds a batch dimension
input_tensor = normalize(original_image_tensor).unsqueeze(0)
```

```
import torchvision.utils as vutils
from torchvision import models as tv_models

# Instantiate the model architecture and load the pre-trained weights.
seg_model = tv_models.segmentation.deeplabv3_resnet50(weights=seg_model_weights).eval()

# Load the base PIL Image
img = Image.open(image_path)

# Create the clean, un-normalized tensor for visualization later
original_image_tensor = transforms.ToTensor()(img)

# Define the normalization transform
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])

# .unsqueeze(0) adds a batch dimension
input_tensor = normalize(original_image_tensor).unsqueeze(0)
```

```
import torchvision.utils as vutils
from torchvision import models as tv_models

# Instantiate the model architecture and load the pre-trained weights.
seg_model = tv_models.segmentation.deeplabv3_resnet50(weights=seg_model_weights).eval()

# Load the base PIL Image
img = Image.open(image_path)

# Create the clean, un-normalized tensor for visualization later
original_image_tensor = transforms.ToTensor()(img)

# Define the normalization transform
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])

# .unsqueeze(0) adds a batch dimension
input_tensor = normalize(original_image_tensor).unsqueeze(0)
```

```
import torchvision.utils as vutils
from torchvision import models as tv_models

# Instantiate the model architecture and load the pre-trained weights.
seg_model = tv_models.segmentation.deeplabv3_resnet50(weights=seg_model_weights).eval()

# Load the base PIL Image
img = Image.open(image_path)

# Create the clean, un-normalized tensor for visualization later
original_image_tensor = transforms.ToTensor()(img)

# Define the normalization transform
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])

# .unsqueeze(0) adds a batch dimension
input_tensor = normalize(original_image_tensor).unsqueeze(0)
```

```
# Define a list of target classes you want to find
target_class_names = ['dog'] # More classes can be added as well

# Define a corresponding list of colors for the segmentation masks for each class
seg_colors = ["blue"]

# Use a list comprehension to get a list of corresponding class indices
class_indices = [class_names_deeplabv3.index(name) for name in target_class_names]

# Print the results for confirmation
print(f"Target Classes:        {target_class_names}")
print(f"Corresponding Indices: {class_indices}")
```

```
# Define a list of target classes you want to find
target_class_names = ['dog'] # More classes can be added as well

# Define a corresponding list of colors for the segmentation masks for each class
seg_colors = ["blue"]

# Use a list comprehension to get a list of corresponding class indices
class_indices = [class_names_deeplabv3.index(name) for name in target_class_names]

# Print the results for confirmation
print(f"Target Classes:        {target_class_names}")
print(f"Corresponding Indices: {class_indices}")
```

```
# Generate prediction
with torch.no_grad():
   output = seg_model(input_tensor)['out'][0]

# Get the predicted class for each pixel by finding the class with the highest score.
output_predictions = output.argmax(0)

# Create a separate boolean mask for each of your target classes.
# The result is a list of boolean tensors, one for each class index.
individual_masks = [(output_predictions == i) for i in class_indices]

# Stack the individual masks into a single tensor of shape (num_masks, H, W).
stacked_masks = torch.stack(individual_masks, dim=0)
```

```
# Generate prediction
with torch.no_grad():
    output = seg_model(input_tensor)['out'][0]

# Get the predicted class for each pixel by finding the class with the highest score.
output_predictions = output.argmax(0)

# Create a separate boolean mask for each of your target classes.
# The result is a list of boolean tensors, one for each class index.
individual_masks = [(output_predictions == i) for i in class_indices]

# Stack the individual masks into a single tensor of shape (num_masks, H, W).
stacked_masks = torch.stack(individual_masks, dim=0)
```

```
# Generate prediction
with torch.no_grad():
    output = seg_model(input_tensor)['out'][0]

# Get the predicted class for each pixel by finding the class with the highest score.
output_predictions = output.argmax(0)

# Create a separate boolean mask for each of your target classes.
# The result is a list of boolean tensors, one for each class index.
individual_masks = [(output_predictions == i) for i in class_indices]

# Stack the individual masks into a single tensor of shape (num_masks, H, W).
stacked_masks = torch.stack(individual_masks, dim=0)
```

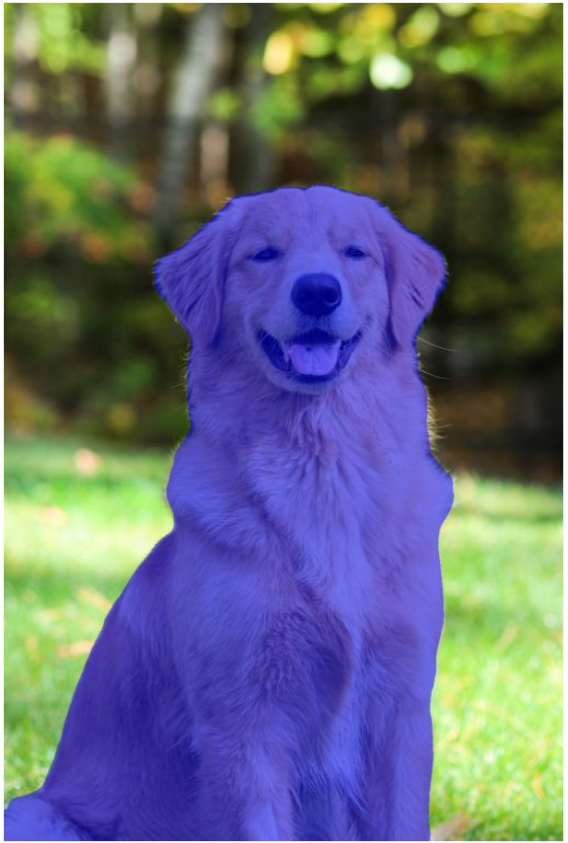
```
# Generate prediction
with torch.no_grad():
    output = seg_model(input_tensor)['out'][0]

# Get the predicted class for each pixel by finding the class with the highest score.
output_predictions = output.argmax(0)

# Create a separate boolean mask for each of your target classes.
# The result is a list of boolean tensors, one for each class index.
individual_masks = [(output_predictions == i) for i in class_indices]

# Stack the individual masks into a single tensor of shape (num_masks, H, W).
stacked_masks = torch.stack(individual_masks, dim=0)
```

```
# Apply segmentation masks using the stacked_masks tensor.
result = vutils.draw_segmentation_masks(image=(original_image_tensor * 255).byte(),
                                        masks=stacked_masks,
                                        alpha=0.5,
                                        colors=seg_colors)

# Visualize the mask
helper_utils.display_images(processed_image=result, figsize=(7, 7))
```

```
# Apply segmentation masks using the stacked_masks tensor.
result = vutils.draw_segmentation_masks(image=(original_image_tensor * 255).byte(),
                                        masks=stacked_masks,
                                        alpha=0.5,
                                        colors=seg_colors)

# Visualize the mask
helper_utils.display_images(processed_image=result, figsize=(7, 7))
```

```
# Apply segmentation masks using the stacked_masks tensor.
result = vutils.draw_segmentation_masks(image=(original_image_tensor * 255).byte(),
                                        masks=stacked_masks,
                                        alpha=0.5,
                                        colors=seg_colors)

# Visualize the mask
helper_utils.display_images(processed_image=result, figsize=(7, 7))
```

Segmentation results with a pretrained model



Laurence Moroney

The power of TorchVision

Overview of TorchVision tools

Laurence Moroney

The power of TorchVision

Overview of TorchVision tools

Preprocessing pipelines, including data augmentation

The power of TorchVision

Overview of TorchVision tools

Preprocessing pipelines, including data augmentation

Working with built-in datasets

The power of TorchVision

Overview of TorchVision tools

Preprocessing pipelines, including data augmentation

Working with built-in datasets

Pretrained models: inference and transfer learning

The power of TorchVision

Overview of TorchVision tools

Preprocessing pipelines, including data augmentation

Working with built-in datasets

Pretrained models: inference and transfer learning

Visualization utilities: bounding boxes and segmentation