UNC CHARLOTTE

The WILLIAM STATES LEE COLLEGE *of* ENGINEERING

Introduction to ML
Lecture 15:Image Processing for Artificial Neural Networks

Hamed Tabkhi

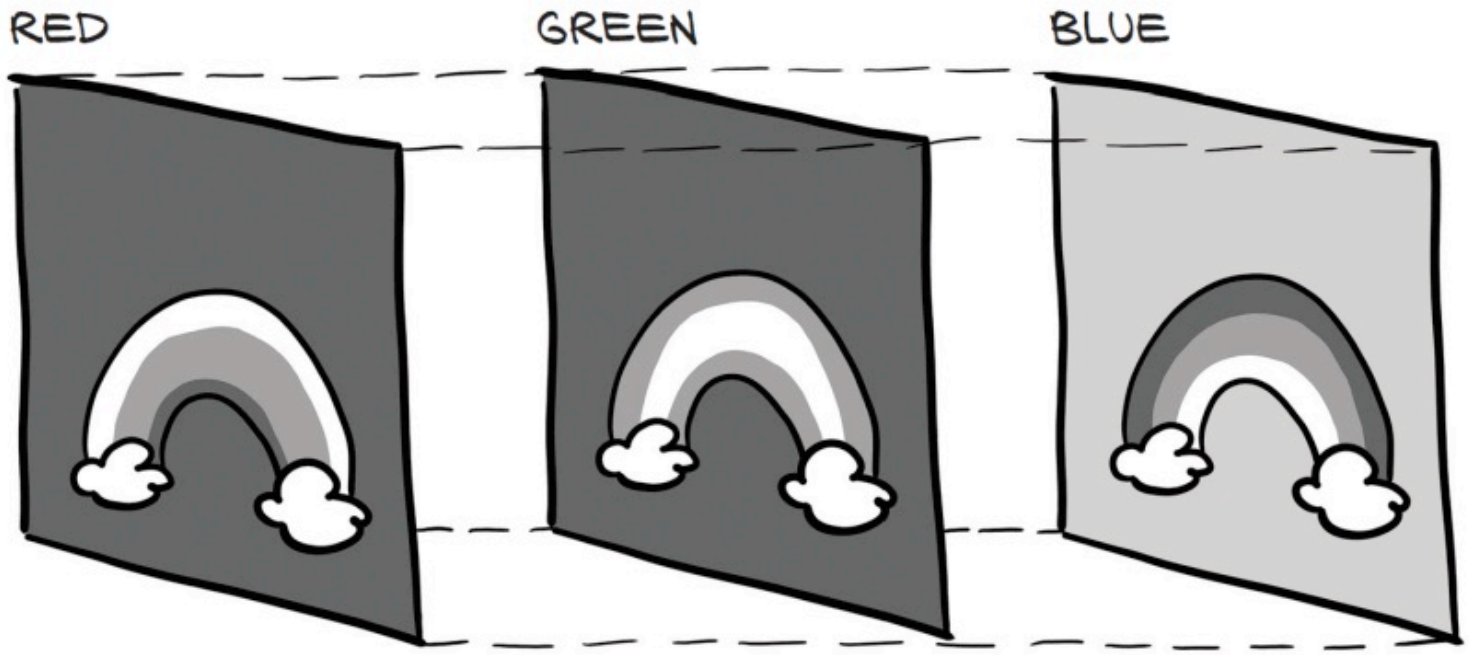
Department of Electrical and Computer Engineering,

University of North Carolina Charlotte (UNCC)

htabkhiv@uncc.edu

World as a Floating-Point Number

Our first job as deep learning practitioners is to encode heterogeneous, real-world data into a tensor of floating-point numbers, ready for consumption by a neural network.



Loading Images

- There are plenty of ways to load images in Python

```
# In[2]:
import imageio
img_arr = imageio.imread('../data/p1ch4/image-dog/bobby.jpg')
img_arr.shape

# Out[2]:
(720, 1280, 3)
```

- At this point, `img` is a NumPy array-like object with three dimensions: two spatial
- dimensions, width and height; and a third dimension corresponding to the red,
- green, and blue channels.

`Note:` TorchVision is another module in Pytorch for interacting with images

Changing the Layout

- We can use the tensor's `permute` method with the old dimensions for each new dimension to get to an appropriate layout.
- Given an input tensor $H \times W \times C$ as obtained previously, we get a proper layout by having channel 2 first and then channels 0 and 1:

$H \times W \times C$ -> $C \times H \times W$

```
# In[3]:
img = torch.from_numpy(img_arr)
out = img.permute(2, 0, 1)
```

NOTE: Note that this operation does not make a copy of the tensor data. Instead, `out` uses the same underlying storage as `img` and only plays with the size and stride information at the tensor level.

Changing the Layout

- To create a dataset of multiple images to use as an input for our neural networks, we store the images in a batch along the first dimension to obtain an $N \times C \times H \times W$ tensor.

```
# In[4]:
batch_size = 3
batch = torch.zeros(batch_size, 3, 256, 256, dtype=torch.uint8)
```

- This indicates that our batch will consist of three RGB images 256 pixels in height and 256 pixels in width.
- Notice the type of the tensor: we're expecting each color to be represented as an 8-bit integer, as in most photographic formats from standard consumer cameras.

Changing the Layout

- We can now load all PNG images from an input directory and store them in the tensor:

```
# In[5]:
import os

data_dir = '../data/p1ch4/image-cats/'
filenames = [name for name in os.listdir(data_dir)
             if os.path.splitext(name)[-1] == '.png']
for i, filename in enumerate(filenames):
    img_arr = imageio.imread(os.path.join(data_dir, filename))
    img_t = torch.from_numpy(img_arr)
    img_t = img_t.permute(2, 0, 1)
    img_t = img_t[:3]
    batch[i] = img_t
```

Here we keep only the first three channels. Sometimes images also have an alpha channel indicating transparency, but our network only wants RGB input.

enumerate() is a built-in Python function. The enumerate() function **allows you to loop over an iterable object and keep track of how many iterations have occurred**.

splitext() method is **used to split the path name into a pair root and ext**. Here, ext stands for extension and has the extension portion of the specified path while root is everything except ext part.

OS— Miscellaneous operating system interfaces
https://docs.python.org/3/library/os.html

Normalizing the Data

- **Neural networks exhibit the best training performance when the input data ranges roughly from 0 to 1, or from -1 to 1 (this is an effect of how their building blocks are defined).**
- We need to cast a tensor to a floating point and normalize the values of the pixels.

- One possibility is to just divide the values of the pixels by 255 (the maximum representable number in 8-bit unsigned):

```
# In[6]:
batch = batch.float()
batch /= 255.0
```

Normalizing the Data

- Another possibility is to compute the mean and standard deviation of the input data and scale it so that the output has zero mean and unit standard deviation across each channel:

```
# In[7]:
n_channels = batch.shape[1]
for c in range(n_channels):
mean = torch.mean(batch[:, c])
std = torch.std(batch[:, c])
batch[:, c] = (batch[:, c] - mean) / std
```

NOTE: Here, we normalize just a single batch of images.

It is a good practice to compute the mean and standard deviation on all the training data in advance

Image Processing Datasets: CIFAR-10

- CIFAR-10 consists of 60,000 tiny 32 × 32 color (RGB) images, labeled with an integer corresponding to 1 of 10 classes:
- airplane (0), automobile (1), bird (2), cat (3), deer (4), dog (5), frog (6), horse (7), ship (8), and truck (9).
- Nowadays, CIFAR-10 is considered too simple for developing or validating new research, but it serves our learning purposes just fine.

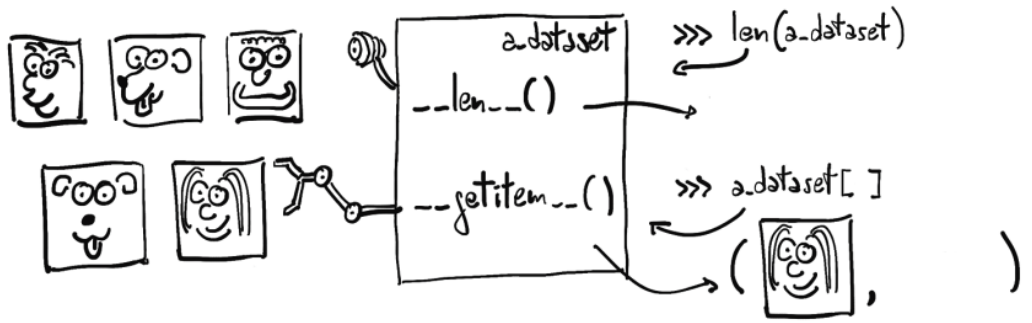
Downloading CIFAR-10

Instantiates a dataset for the training data;
TorchVision downloads the data if it is not present.

With train=False, this gets us a dataset for the validation data, again downloading as necessary.

```
# In[2]:
from torchvision import datasets
data_path = '../data-unversioned/p1ch7/'
cifar10 = datasets.CIFAR10(data_path, train=True, download=True)
cifar10_val = datasets.CIFAR10(data_path, train=False, download=True)
```

- The dataset is returned as a subclass of `torch.utils.data.Dataset`.
- It is an object that is required to implement two methods: `__len__` and `__getitem__`.
- `__len__` returns the number of items in the dataset
- `__getitem__` returns the item, consisting of a sample and its corresponding label (an integer index).



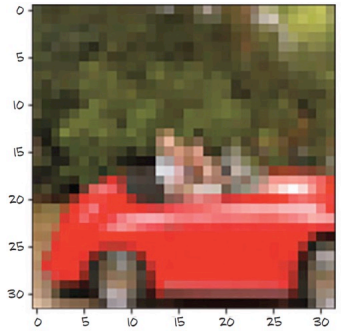
Showing Images

```
# In[5]:
len(cifar10)

# Out[5]:
50000
```

- We get a `PIL` (Python Imaging Library, the `PIL` package) image with our desired output—an integer with the value `1`, corresponding to “automobile”:

```
# In[6]:
img, label = cifar10[99]
img, label, class_names[label]

# Out[6]:
(<PIL.Image.Image image mode=RGB size=32x32 at 0x7FB383657390>,
 1,
 'automobile')
```

```
# In[7]:
plt.imshow(img)
plt.show()
```



- Since the dataset is equipped with the `__getitem__` method, we can use the standard indexing and lists to access individual items.

ToTensor

- We need a way to convert the PIL image to a PyTorch tensor before we can do anything with it.
- That's where `torchvision.transforms` comes in.
- This module defines a set of composable, function-like objects that can be passed as an argument to a `torchvision`.
- Among those transforms, we can spot `ToTensor`, which turns NumPy arrays and PIL images to tensors.
- It also takes care to lay out the dimensions of the output tensor as $C \times H \times W$ (channel, height, width)
- Once instantiated, it can be called like a function with the PIL image as the argument, returning a tensor as output:

```
# In[9]:
from torchvision import transforms

to_tensor = transforms.ToTensor()
img_t = to_tensor(img)
img_t.shape

# Out[9]:
torch.Size([3, 32, 32])
```

- The image has been turned into a 3 × 32 × 32 tensor and therefore a 3-channel (RGB) 32 × 32 image.
- Note that nothing has happened to `label`; it is still an integer.

ToTensor

- As we anticipated, we can pass the transform directly as an argument to `dataset.CIFAR10`:

```
# In[10]:
tensor_cifar10 = datasets.CIFAR10(data_path, train=True, download=False,
                                  transform=transforms.ToTensor())
```

- At this point, accessing an element of the dataset will return a tensor, rather than a PIL image:

```
# In[11]:
img_t, _ = tensor_cifar10[99]
type(img_t)

# Out[11]:
torch.Tensor

# In[12]:
img_t.shape, img_t.dtype

# Out[12]:
(torch.Size([3, 32, 32]), torch.float32)
```

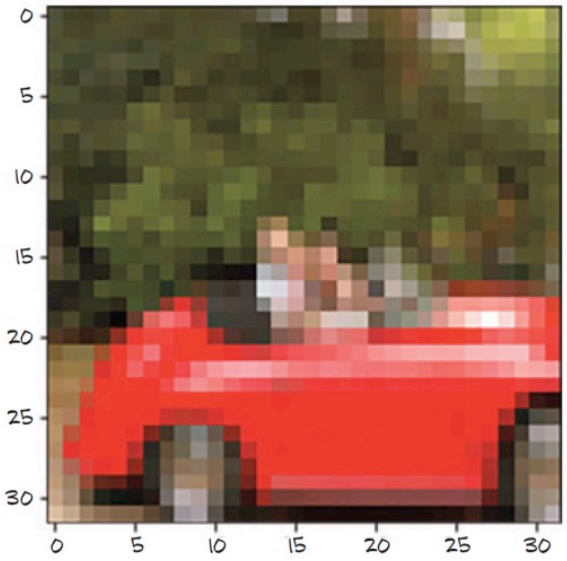
Whereas the values in the original PIL image ranged from 0 to 255 (8 bits per channel), the `ToTensor` transform turns the data into a 32-bit floating-point per channel, scaling the values down from 0.0 to 1.0.

ToTensor

- And let's verify that we're getting the same image out:

```
# In[14]:
plt.imshow(img_t.permute(1, 2, 0))
plt.show()

# Out[14]:
<Figure size 432x288 with 1 Axes>
```

Changes the order of the axes from C × H × W to H × W × C



- Note how we have to use `permute` to change the order of the axes from C × H × W to H × W × C to match what Matplotlib expects.

Normalizing Image Data

- It's good practice to normalize the dataset so that each channel has zero mean and unitary standard deviation.
- By choosing activation functions that are linear around 0 plus or minus 1 (or 2), keeping the data in the same range means it's more likely that neurons have nonzero gradients and, hence, will learn sooner.
- Also, normalizing each channel so that it has the same distribution will ensure that channel information can be mixed and updated through gradient descent using the same learning rate.
- In order to make it so that each channel has zero mean and unitary standard deviation, we can compute the mean value and the standard deviation of each channel across the dataset and apply the following transform: `v_n[c] = (v[c] - mean[c]) / stdev[c]`.

Normalizing Image Data

- Since the CIFAR-10 dataset is small, we'll be able to manipulate it entirely in memory.
- Let's stack all the tensors returned by the dataset along an extra dimension:

```
# In[15]:
imgs = torch.stack([img_t for img_t, _ in tensor_cifar10], dim=3)
imgs.shape

# Out[15]:
torch.Size([3, 32, 32, 50000])

# In[16]:
imgs.view(3, -1).mean(dim=1)

# Out[16]:
tensor([0.4915, 0.4823, 0.4468])

# In[17]:
imgs.view(3, -1).std(dim=1)

# Out[17]:
tensor([0.2470, 0.2435, 0.2616])
```

Recall that view(3, -1) keeps the three channels and merges all the remaining dimensions into one, figuring out the appropriate size. Here our 3 × 32 × 32 image is transformed into a 3 × 1,024 vector, and then the mean is taken over the 1,024 elements of each channel.

Normalizing Image Data

- With these numbers in our hands, we can initialize the `Normalize` transform

```
# In[18]:
transforms.Normalize((0.4915, 0.4823, 0.4468), (0.2470, 0.2435, 0.2616))

# Out[18]:
Normalize(mean=(0.4915, 0.4823, 0.4468), std=(0.247, 0.2435, 0.2616))
```

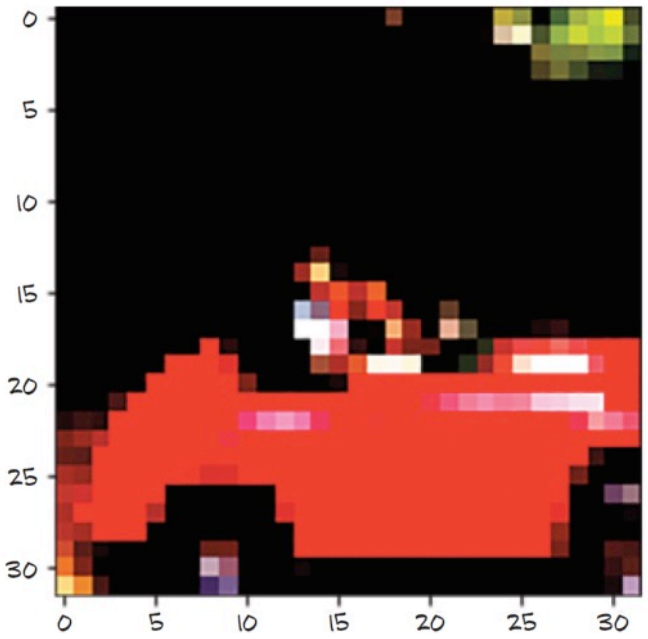
- and concatenate it after the `ToTensor` transform:

```
# In[19]:
transformed_cifar10 = datasets.CIFAR10(
    data_path, train=True, download=False,

transform=transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4915, 0.4823, 0.4468),
                         (0.2470, 0.2435, 0.2616))
]))
```

Normalizing Image Data

- Note that, at this point, plotting an image drawn from the dataset won't provide us with a faithful representation of the actual image:

```
# In[21]:
img_t, _ = transformed_cifar10[99]

plt.imshow(img_t.permute(1, 2, 0))
plt.show()
```



- Normalization has shifted the RGB levels outside the 0.0 to 1.0 range and changed the overall magnitudes of the channels.
- All of the data is still there; it's just that Matplotlib renders it as black.