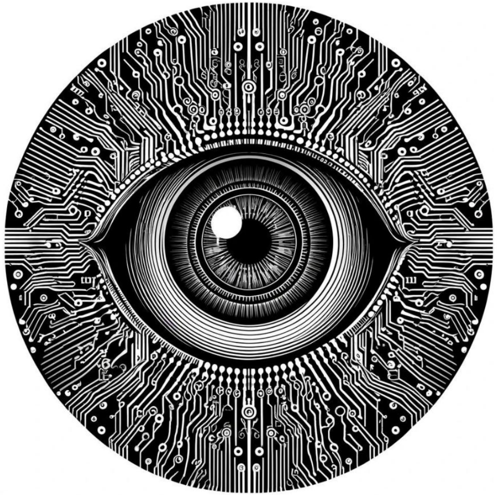
Image Tensors in PyTorch



Antonio Rueda-Toicen

KI Service Zentrum
by Hasso-Plattner-Institut

SPONSORED BY THE
Federal Ministry
of Education
and Research

Learning goals

- Gain familiarity with `torch` tensors
- Understand the need for data type transformation and scaling
- Understand how to move `torch` tensors between devices

Images as tensors

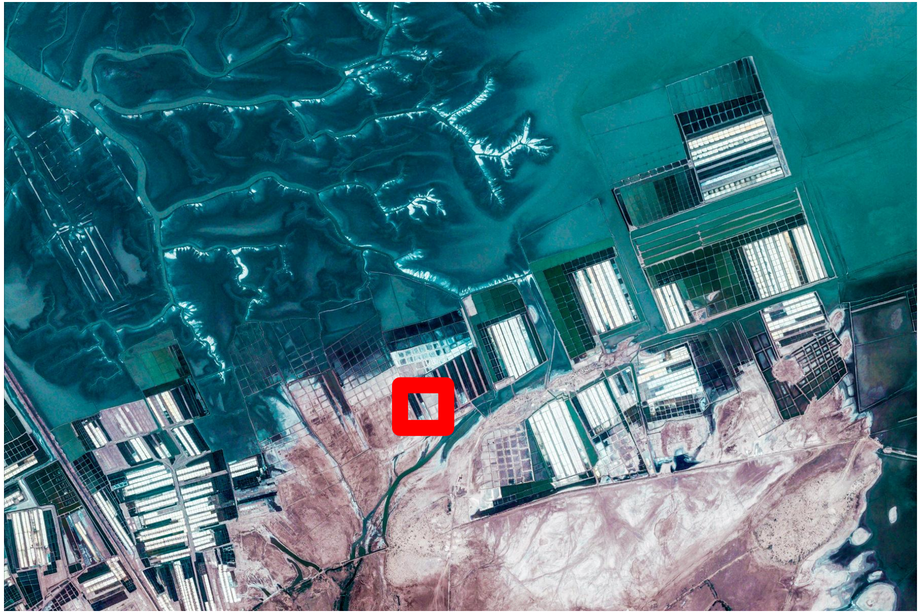


Image from Google Earth

What a human sees

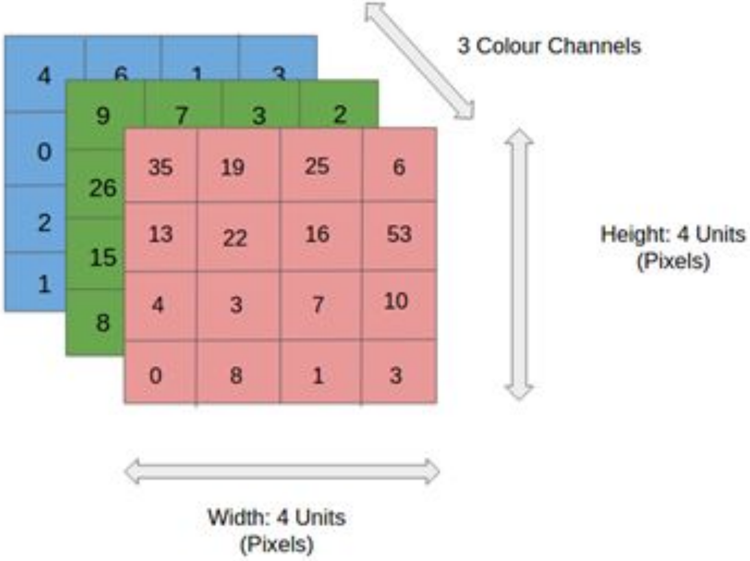
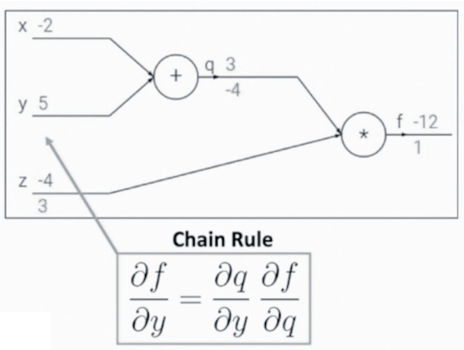


Image source

What the computer 'sees'

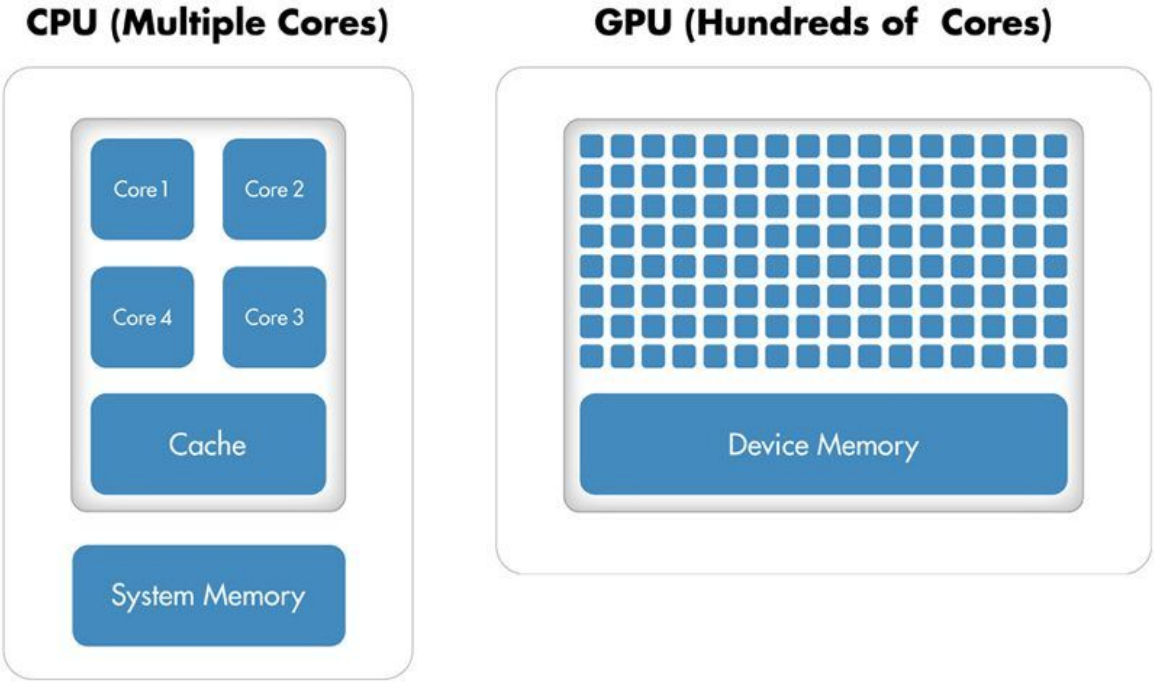
Why do we use PyTorch tensors?

1. Can run on the Graphics Processing Unit (GPU) or Tensor Processing Unit (TPU) speeding up training and inference

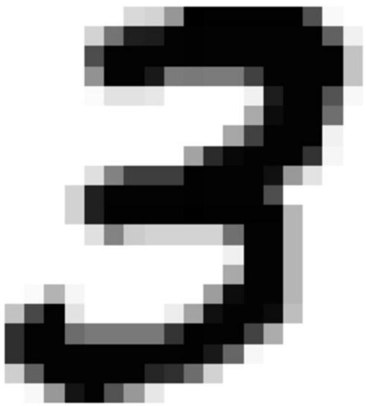


2. Do automatic gradient computation (autograd), enabling us to update the weights of models with automatically computed derivatives of the loss with respect to the weights

CPUs vs GPUs

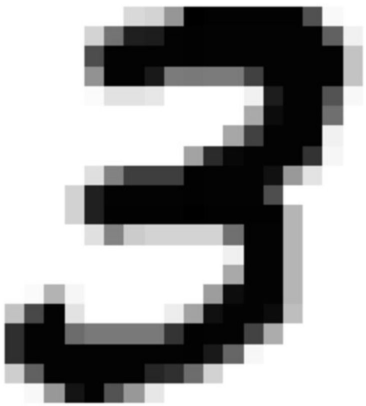

CPU (Multiple Cores)
Core 1
Core 2
Core 4
Core 3
Cache
System Memory
GPU (Hundreds of Cores)
Device Memory

Moving tensors to the GPU



```
import torch.nn as nn
# A model with a single linear layer
model = nn.Linear(in_features = 28 * 28,
                  out_features = 10)

device = 'cuda' if torch.cuda.is_available() else 'cpu'
# Move the model to the GPU
model = model.to(device)
torch_tensor_gray.to(device)

# Get prediction scores aka 'logits' (in GPU)
scores = model(torch_tensor_gray)
```

Moving tensors to the CPU for inspection/visualization



```
# Get prediction scores aka 'logits' (in GPU)

scores = model(torch_tensor_gray)


# Fails, scores are in the GPU

plt.show(scores)


# Moves a copy of the tensor the cpu
# scores.cpu() is acceptable too
plt.show(scores.to('cpu'))
```

Speedup on matrix multiplication

```
Running matrix multiplication benchmarks...

Device: Tesla P100-PCIE-16GB
+---------------+-----------------+-----------------+-----------+
| Matrix Size   | CPU Time (s)    | GPU Time (s)    | Speedup   |
+===============+=================+=================+===========+
| 32x32         | 0.0000 ± 0.0000 | 0.0000 ± 0.0000 | 0.3x      |
+---------------+-----------------+-----------------+-----------+
| 2048x2048     | 0.0588 ± 0.0024 | 0.0026 ± 0.0001 | 23.0x     |
+---------------+-----------------+-----------------+-----------+
| 8192x8192     | 3.8277 ± 0.1194 | 0.1285 ± 0.0022 | 29.8x     |
+---------------+-----------------+-----------------+-----------+
```

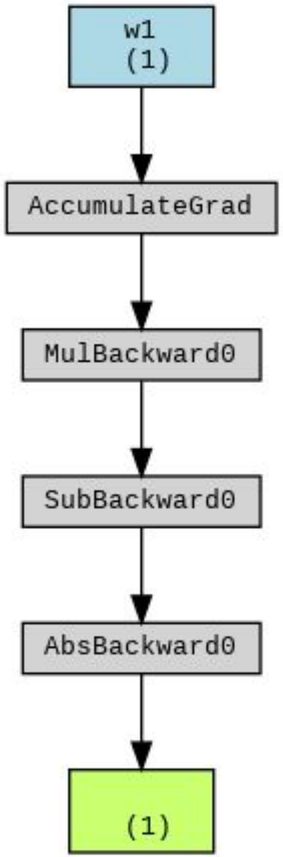
Quick benchmark on Kaggle

Autograd to compute derivatives

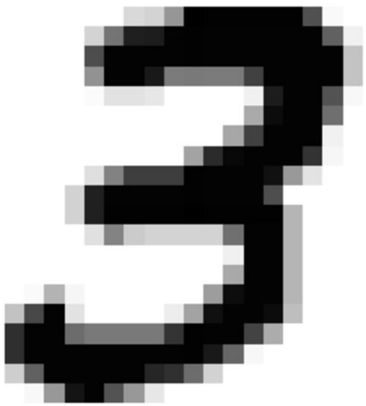
```python
import torch

# Create weight tensor with requires_grad=True
w1 = torch.tensor([4.0], requires_grad=True)

# Input features and ground truth do not require gradients (do not change)
x1 = torch.tensor([5.0], requires_grad=False)
y = torch.tensor([6.0], requires_grad=False)

# Let's create a simple computation graph
L = torch.abs(w1 * x1 - y)

# Will show 14.0, requires_grad = True
print(f"l = {z.item()}, requires_grad = {L.requires_grad}")

# Compute gradients
L.backward()

# Access the gradient
print(f"dl/dw1 = {w1.grad}")  # Will be 5.0
```

$$L = |w_1 x_1 - y|$$

$$\frac{\partial L}{\partial w_1} = \begin{cases} x_1 & \text{if } w_1 x_1 - y \geq 0, \\ -x_1 & \text{if } w_1 x_1 - y < 0. \end{cases}$$



The dimensions of a `torch` image tensor



`torch` image tensors follow the format: `[N, C, H, W]` where:

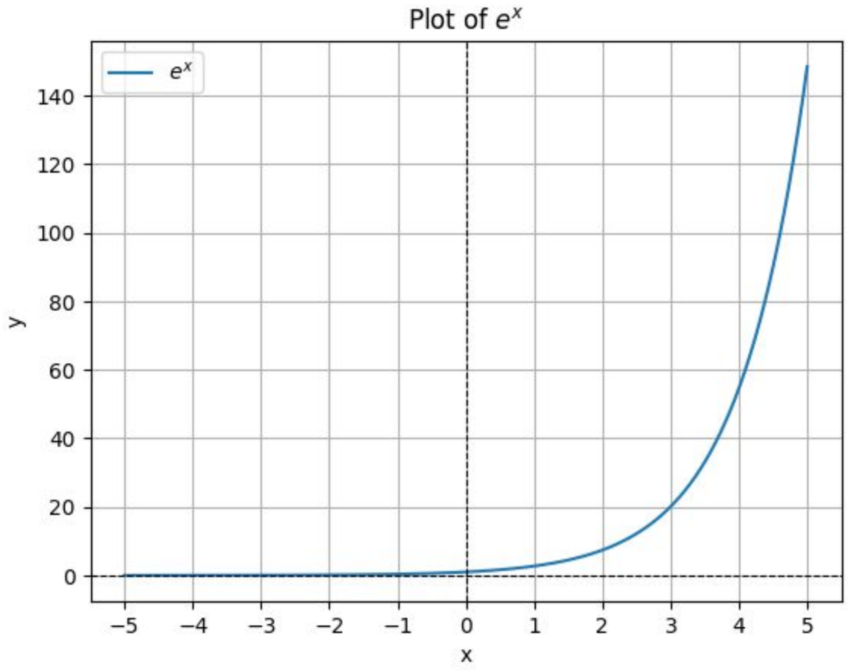
- `N` = batch size (number of images)
- `C` = channels (e.g., 1 for grayscale, 3 for RGB)
- `H` = height in pixels
- `W` = width in pixels

```
# Tensors shown to neural networks include the batch size

print(torch_tensor_gray.unsqueeze(0).shape)

# prints torch.Size([1, 1, 28, 28])

print(torch_tensor_gray.dim())

# prints 3, the “rank” (# of dimensions)
```

PyTorch (aka `torch`) tensors

```
# ToImage() converts NumPy arrays with H, W, C format to torch's C, W, H
transform = transforms.ToImage()
image_tensor = transform(np_array)

# Will print 3, 1200, 1800
print(image_tensor.shape)

# Will print 1200, 1800, 3
print(image_tensor.permute(1, 2, 0).shape)

# Error, matplotlib expects NumPy format
plt.imshow(image_tensor)

# Channel order needs to permuted to use matplotlib, as it expects NumPy’s channel order
plt.imshow(image_tensor.permute(1, 2, 0))
```

Numerical stability




```
original_values = torch.tensor([255, 204, 170], dtype=torch.uint8)

torch.exp(original_values)
```

$$\mathrm{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \quad \text{for } i = 1, \ldots, n$$

Numerical stability example

```python
# Let's simulate a single pixel intensity in RGB
original_values = torch.tensor([255, 204, 170], dtype=torch.uint8)
print(f"Original values: {original_values}")

# Convert to float and scale
float_values = original_values.float() / 255
print(f"Scaled values: {float_values}")

# Example of multiplication stability
print(f"Original exp: {torch.exp(original_values)}")  # Very large! inf!
print(f"Scaled exp: {torch.exp(float_values)}")  # Stay small, bitte
```

Numerical stability

```
Original values: tensor([255, 204, 170], dtype=torch.uint8)
Scaled values: tensor([1.0000, 0.8000, 0.6667])
Original exp: tensor([inf, inf, inf])
Scaled exp: tensor([2.7183, 2.2255, 1.9477])
```

`torch` tensors with rescaled floating point values

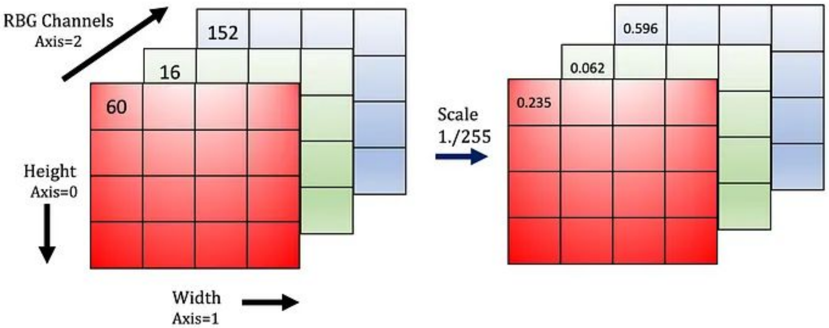


Image from source

1.0

0.5

0.0

```python
# We convert the input to float and rescale it to
# 0,1 to avoid overflow and get stable operations
transform = transforms.Compose([
      transforms.ToImage(),
      transforms.ToDType(torch.float32,  scale=True)])

# This transform would also work with a NumPy array
image_tensor = transform(pil_image)

# Prints (1.0, 0.0)
print(image_tensor.max(), image_tensor.min())
```

Summary

Tensors in PyTorch can run on the GPU and do automatic gradient computation

- We use `tensor_name.to(device)` to move tensors between CPU and GPU memory
- We use `result.backward()` to compute gradients on tensors with `requires_grad = True`

Common pitfall: different standards of tensor dimensions in NumPy vs PyTorch

- We use the B, C, H, W order in PyTorch; H, W, C in NumPy. The batch dimension corresponds to the number of tensors that we process at once on the device (GPU or CPU). The `toImage()` transform allows us to put the channel dimension on its proper position

Conversion to float and scaling tensors helps against numerical errors

- Operations like `torch.exp()` overflow without scaling

SPONSORED BY THE
Federal Ministry of Education and Research

References

torch.Tensor.to

- https://pytorch.org/docs/stable/generated/torch.Tensor.to.html

ToImage

- https://pytorch.org/vision/0.19/generated/torchvision.transforms.v2.ToImage.html

SPONSORED BY THE

Federal Ministry
of Education
and Research