三.张量运算

有超过 100 种张量运算，包括算术、线性代数、矩阵操作（转置、索引、切片）。 用于采样和查看此处的全面描述。

这些操作中的每一个都可以在 GPU 上运行（通常以比 CPU 更高的速度）。

CPU 最多有 16 个内核。 核心是执行实际计算的单元。 每个核心按顺序处理任务（一次一个任务）。 GPU有1000个核心。 GPU 内核在并行处理中处理计算。 任务在不同的核心之间进行划分和处理。 在大多数情况下，这使得 GPU 比 CPU 更快。 GPU 处理大数据的性能优于处理小数据。 GPU 通常用于图形或神经网络的高强度计算（我们将在后面的神经网络单元中看到更多相关内容）。 PyTorch 可以使用 Nvidia CUDA 库来利用他们的 GPU 卡。

```
#r "nuget: TorchSharp"
```

Installed Packages

- TorchSharp, 0.96.7

```
using TorchSharp;
using static TorchSharp.TensorExtensionMethods;
using Microsoft.DotNet.Interactive.Formatting;

var style = TensorStringStyle.Julia;

Formatter.SetPreferredMimeTypesFor(typeof(torch.Tensor), "text/plain");
Formatter.Register<torch.Tensor>((torch.Tensor x) => x.ToString(style,
newLine: "\n"));
```

基本数值

算术是 TorchSharp 的全部内容，功能丰富。 不过，这都是关于张量算术的——这就是 GPU 加速有意义的地方。

```
var a = torch.ones(3,4);
var b = torch.zeros(3,4);
var c = torch.tensor(5);
a * c + b
```

```
[3x4], type = Float32, device = cpu
 5 5 5 5
 5 5 5 5
 5 5 5 5
```

```
a.mul_(c).add_(b)
```

```
[3x4], type = Float32, device = cpu
 5 5 5 5
 5 5 5 5
 5 5 5 5
```

在此之后，'a' 不再持有，因为它已被覆盖。 如果始终如一地使用，使用就地运算符对性能的影响是显着的，但重要的是要知道您正在覆盖什么，而不是过度使用就地运算符。 将其视为性能优化。

```
a
```

```
[3x4], type = Float32, device = cpu
 5 5 5 5
 5 5 5 5
 5 5 5 5
```

传播

在上面的简单示例中，您看到“c”是从单个值定义的。 如果我们看一下，我们可以看到它是一个单例张量。 也就是说，它没有形状。

```
c.shape
```

(empty)

```
c
```

```
[], type = Int32, device = cpu, value = 5
```

```
a = torch.ones(3,4);
(a + torch.ones(4)).print();
a + torch.ones(1,4)
```

```
[3x4], type = Float32, device = cpu
 2 2 2 2
 2 2 2 2
 2 2 2 2



[3x4], type = Float32, device = cpu
 2 2 2 2
 2 2 2 2
 2 2 2 2
```

数值库

可用的数值运算符的集合太大而无法在此处进行介绍，但只要说所有常见的嫌疑人都可用就足够了。 大多数操作都是基于元素的，即运算符应用于操作数的每个元素，可能涉及广播。

一个值得注意且非常重要的例外是矩阵乘法，它是将向量点积推广到矩阵。 '*' 运算符表示逐元素乘法，而矩阵乘法由 'mm' 方法执行：

```
a = torch.full(4,4, 17);
b = torch.full(4,4, 12);

(a * b).print();
(a.mm(b)).str()
```

```
[4x4], type = Int64, device = cpu
 204 204 204 204
 204 204 204 204
 204 204 204 204
 204 204 204 204



[4x4], type = Int64, device = cpu
 816 816 816 816
 816 816 816 816
 816 816 816 816
 816 816 816 816
```

```
var x = torch.rand(5);
var y = torch.rand(5);
```

```
(x * torch.log(y)).print();
x.xlogy(y)
```

```
[5], type = Float32, device = cpu
 -1.1874 -0.77134 -0.038077 -0.12703 -0.084931



[5], type = Float32, device = cpu
 -1.1874 -0.77134 -0.038077 -0.12703 -0.084931
```

随机数和分布

TorchSharp 中有一组丰富的随机数生成 API。 我们已经看到了最容易使用的那些：randn()、rand() 和 randint()。 正态分布和均匀分布是许多其他随机数特征的基础。

注意 randint() 会生成整数，默认类型是 64 位整数。 randperm() 也是如此。

```
torch.rand(10).print();
torch.randn(10).print();
torch.randint(100,10).print();
torch.randperm(25).print();
```

```
[10], type = Float32, device = cpu
 0.24964 0.51392 0.36439 0.15735 0.95061 0.8357 0.80364 0.87166 0.12194
0.046405

[10], type = Float32, device = cpu
 -2.3601 -0.77014 -1.2273 -0.35986 -0.20497 -0.94875 -0.36751 -1.3559
-0.99768 -0.1024

[10], type = Int64, device = cpu
 91 7 82 30 6 47 38 40 82 27

[25], type = Int64, device = cpu
 14 0 11 3 1 8 15 10 2 7 9 24 18 21 4 17 23 20 22 5 16 12 13 6 19
```

设置种子

与大多数随机数库一样，TorchSharp 允许您设置用于随机数生成的种子。 您应该看到第一个系列和最后一个系列相同，而中间的系列不同。

TorchSharp 的一个特点是，在使用 CPU 和 GPU 时，使用相同的初始种子不会导致相同的数字序列。 你不能通过在 GPU 上运行来重现你在 CPU 上的结果。

```
torch.random.manual_seed(4711);
torch.rand(10).print();
torch.random.manual_seed(17);
torch.rand(10).print();
torch.random.manual_seed(4711);
torch.rand(10).print();
```

```
[10], type = Float32, device = cpu
 0.69071 0.94377 0.033924 0.28365 0.10061 0.89436 0.21124 0.16128 0.59802
0.43391

[10], type = Float32, device = cpu
 0.43424 0.53511 0.83021 0.12386 0.029321 0.5494 0.38249 0.54626 0.46828
0.017153

[10], type = Float32, device = cpu
 0.69071 0.94377 0.033924 0.28365 0.10061 0.89436 0.21124 0.16128 0.59802
0.43391
```