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

Installed Packages

- TorchSharp, 0.96.7

```
#r "nuget: libtorch-cuda-11.3-linux-x64"
```

Installed Packages

- libtorch-cuda-11.3-linux-x64, 1.11.0.1

使用 cuda

当您的机器配备支持 CUDA 编程的 GPU 时，您可以使用“TorchSharp-cuda-windows”或“TorchSharp-cuda-linux”，具体取决于您的操作系统。 MacOS 没有 CUDA 发行版。

使用 CUDA，尤其是用于训练，可以显着提高性能，通常提高几个数量级。这可能是模型训练可行与不可行之间的区别。

注意：这些教程对功能的要求并不高，但要训练具有适度数据大小的真实视觉模型，您至少需要 8MB 的专用 GPU 内存。即使像 CIFAR10 这样简单的东西（在这个 repo 的示例解决方案中）也需要那么多内存才能不爆炸。 6MB，在支持 Nvidia 的笔记本电脑上的常见内存大小，是不够的。

安装正确的后端包后，使用 CUDA 非常简单。 TorchSharp 让您可以像在 CPU 上一样轻松地在 GPU 上创建和使用张量。以前，我们在创建张量时没有使用 'device' 参数，但它很容易使用。请注意，字符串表示看起来有所不同——它表示设备的“cuda:0”而不是“cpu”。

```
using TorchSharp;
using static TorchSharp.TensorExtensionMethods;
using static TorchSharp.torch.distributions;

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"));
```

```
torch.ones(3,4, device: torch.CUDA)
```

```
[3x4], type = Float32, device = cuda:0
 1 1 1 1
 1 1 1 1
 1 1 1 1
```

```
var a = torch.ones(3,4, device: torch.CUDA);
var b = torch.ones(3,4, device: torch.CUDA);
var c = torch.ones(3,4, device: torch.CUDA);

(a + b).print();
(a + c).print();
```

```
[3x4], type = Float32, device = cuda:0
 2 2 2 2
 2 2 2 2
 2 2 2 2

[3x4], type = Float32, device = cuda:0
 2 2 2 2
 2 2 2 2
 2 2 2 2
```

```
(a + c.cuda()).print();
c.print();

// or:

(a + c.to(device:torch.CUDA)).print();
c.print();
```

```
[3x4], type = Float32, device = cuda:0
 2 2 2 2
 2 2 2 2
 2 2 2 2

[3x4], type = Float32, device = cuda:0
 1 1 1 1
 1 1 1 1
 1 1 1 1

[3x4], type = Float32, device = cuda:0
 2 2 2 2
 2 2 2 2
```

```
 2 2 2 2

[3x4], type = Float32, device = cuda:0
 1 1 1 1
 1 1 1 1
 1 1 1 1
```

GPU 内存管理

在继续之前，重要的是讨论张量的显式内存管理。 因为 CUDA 没有任何虚拟内存机制，所以很容易耗尽 GPU 内存，除非仔细管理。

TorchSharp 张量最终被垃圾收集，当堆开始变满时触发。 但是，堆是所有 CPU 内存，并且只有托管运行时可以看到的内存。 张量的存储是在本机代码中分配的，因此不会增加触发 GC 的内存压力。 这对于 GPU 内存来说尤其不稳定。

因此，tensor 类实现了 IDisposable，这样就可以手动释放内存。

TorchSharp 算法会产生很多临时的，不再使用时需要释放。 考虑这个表达式：

```
(a + b) * (a + c.cuda())
```

```
[3x4], type = Float32, device = cuda:0
 4 4 4 4
 4 4 4 4
 4 4 4 4
```

```
var t0 = a + b;
var t1 = c.cuda();
var t2 = a + t1;
var t3 = t0 * t2;
```

```
using (torch.Tensor t0 = a + b, t1 = c.cuda(), t2 = a + t1 ) {

    var t3 = t0 * t2;
    t3.print();
}
```

```
[3x4], type = Float32, device = cuda:0
 4 4 4 4
 4 4 4 4
 4 4 4 4
```

处置范围

在 0.95.4 版本中，TorchSharp 引入了 DisposeScope 的概念，它系统地处理了 dispose 模式。 它引入了动态（运行时）作用域的概念，它控制作用域有效时创建的所有张量的提升时间。 词法范围，即保存变量的源代码位置等，对动态范围管理没有影响。

当任何 .NET 张量被创建时，它会被注册到当前的动态范围（如果有的话）。 一旦注册，张量将在释放范围时自动释放。 张量是否保存在范围外或范围内声明的变量中都没有关系。

尝试使用和不使用“使用”行运行下一个单元格（将其注释掉），并注意如果你没有它，张量计数如何增长，但当你这样做时保持不变。

```
Console.WriteLine(torch.Tensor.TotalCount);
using (var d = torch.NewDisposeScope())
{
    var t3 = (a + b) * (a + c.cuda());
    t3.print();
}
Console.WriteLine(torch.Tensor.TotalCount);
```

```
725
[3x4], type = Float32, device = cuda:0
 4 4 4 4
 4 4 4 4
 4 4 4 4

725
```

```
torch.Tensor t3;
using (var d = torch.NewDisposeScope())
{
    t3 = (a + b) * (a + c.cuda());
}
t3.print();
```

```
System.InvalidOperationException: Tensor invalid -- empty handle.


   at TorchSharp.torch.Tensor.get_Handle()


   at TorchSharp.torch.Tensor.get_device_type()
```

```
   at TorchSharp.torch.Tensor.ToString(TensorStringStyle style, String
fltFormat, Int32 width, CultureInfo cultureInfo, String newLine)


   at TorchSharp.TensorExtensionMethods.str(Tensor tensor,
TensorStringStyle style, String fltFormat, Int32 width, String newLine,
CultureInfo cultureInfo)


   at TorchSharp.TensorExtensionMethods.print(Tensor t, TensorStringStyle
style, String fltFormat, Int32 width, String newLine)


   at Submission#12.<<Initialize>>d__0.MoveNext()


--- End of stack trace from previous location ---


   at
Microsoft.CodeAnalysis.Scripting.ScriptExecutionState.RunSubmissionsAsync[T
Result](ImmutableArray`1 precedingExecutors, Func`2 currentExecutor,
StrongBox`1 exceptionHolderOpt, Func`2 catchExceptionOpt, CancellationToken
cancellationToken)
```

```
torch.Tensor t3;
using (var d = torch.NewDisposeScope())
{
    t3 = d.MoveToOuter((a + b) * (a + c.cuda()));
}
t3.print();
Console.WriteLine(t3.IsInvalid);
t3.Dispose();
Console.WriteLine(t3.IsInvalid);
```

```
[3x4], type = Float32, device = cuda:0
 4 4 4 4
 4 4 4 4
 4 4 4 4

False
True
```

```
torch.Tensor t3;
using (var d0 = torch.NewDisposeScope())
{
    using (var d1 = torch.NewDisposeScope())
```

```
    {
        t3 = d1.MoveToOuter((a + b) * (a + c.cuda()));
    }
    t3.print();
    Console.WriteLine(t3.IsInvalid);
}
Console.WriteLine(t3.IsInvalid);
```

```
[3x4], type = Float32, device = cuda:0
 4 4 4 4
 4 4 4 4
 4 4 4 4

False
True
```

在 GPU 上放置模型参数

要使用 GPU，必须将张量复制或移动到那里。 当你训练时，你的数据准备逻辑负责将数据传输到 GPU，但我们也需要那里的权重。 TorchSharp 通过在模块上定义一个 'to()' 方法来支持这一点，该方法可用于将模型依赖的权重移动（而不是复制）到 GPU（或返回到 CPU）。 我们还没有查看模型，但请记住这一点以备后用：

var model = ...; model.to(torch.CUDA);