```
#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

```
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"));
```

# 模型

虽然使用 Torch 进行数值计算是有用且高效的，但无论用 Python、C# 还是 F#，建模都是用 Torch 作为核心。

了解 LibTorch，Pytorch 和 TorchSharp 下的引擎是如何工作的很重要——它是一个动态模型引擎，这意味着该引擎可以跟踪从宿主语言完成的工作，跟踪它并进行能够执行反向传播。这与早期框架的工作方式非常不同，例如 Tensorflow v1.X——在这些框架中，模型图是由主机代码构建的，只有在构建图之后才能运行模型。

TF v1.X 的静态方法有一些与之相关的挑战，其中之一是当代码没有急切地运行时很难调试。静态方法的优点之一是外部化整个模型相对简单：它的权重和逻辑。

动态框架并非如此——权重和逻辑是紧密相连的，但只是松散的。为了在训练期间和之后执行模型，您需要模型的代码，以宿主语言表示。

# 模型类

通常，我们希望将模型的逻辑保留在它自己的类中。 这使得使用其他 TorchSharp 构造来组合和运行模型变得容易。 从概念上讲，这就是一个模型——一个张量 -> 张量函数，实现为一个具有“forward()”函数的类。 这个函数是放置模型逻辑的地方。 （如果 C# 支持 operator()，就像 C++ 那样，那我们就在这里使用它。就像 Python 一样。）

TorchSharp 使构建模型变得容易，因为您只需指定 forward 函数。 要进行反向传播，您还需要后向函数，该函数支持使用微积分的链式法则计算梯度。 在 Torch 中，只要前向函数仅依赖于 Torch API 进行计算，就会自动实现后向函数。

让我们从一个超级简单的模型开始，它只有一个线性层，它期望一个具有 1000 个输入元素的张量，并将产生一个具有 100 个元素的张量

```
using static TorchSharp.torch;

using static TorchSharp.torch.nn;
using static TorchSharp.torch.nn.functional;
```

```
private class Trivial : nn.Module
{
    public Trivial()
        : base(nameof(Trivial))
    {
        RegisterComponents();
    }

    public override Tensor forward(Tensor input)
    {
        return lin1.forward(input);
    }

    private nn.Module lin1 = nn.Linear(1000, 100);
}
```

```
var input = rand(1000);
var model = new Trivial();
model.forward(input)
```

```
[100], type = Float32, device = cpu
 0.58924 -0.3256 0.1649 0.22228 -0.15231 -0.12498 -0.32059 -0.21909 0.45464
-0.22725 -0.17543 ...
```

```
model.forward(rand(3,1000))
```

```
[3x100], type = Float32, device = cpu
 0.20076 0.086515 0.16434 -0.12702  0.21787  0.29725 -0.36543   -0.21218
```

```
0.14961 -0.10872 ...
 0.31423 -0.16598 0.49614 -0.26881 -0.19279  0.11501 -0.21428   -0.35448
0.55196 0.022209 ...
 0.23449 -0.37997 0.59868 -0.17393  0.14784 0.038413 -0.41799 -0.0074947
0.46324 -0.20228 ...
```

```
model.forward(rand(2,3,1000))
```

```
[2x3x100], type = Float32, device = cpu
[0,..,..] =
 0.54354 -0.050132 0.25674  -0.27538  -0.012125 0.12768  -0.4064 -0.22759
0.51197 -0.21235 -0.18049 ...
 0.36969  -0.23157 0.49817 -0.026331    0.13588  0.1565 -0.66135 -0.22795
0.56968 -0.13282 -0.39739 ...
 0.23221  -0.17621 0.12943   0.06133 -0.0010741 0.15685 -0.31468 -0.30573
0.70008 -0.21448 -0.27042 ...

[1,..,..] =
 0.12672 -0.23669 0.69381 -0.011724   0.35251 0.53432 -0.45561 -0.0073498
0.52984 -0.018081 ...
 0.39906 -0.10035  0.3624   -0.1719   0.37989 0.12773 -0.37688  -0.022094
0.32009  0.038945 ...
 0.26549 -0.35346 0.36921  -0.38581 -0.059927 0.44854 -0.31564   -0.24862
0.68627  -0.26292 ...
```

所有 TorchSharp 运算符都将接受批处理或非批处理输入数据，但并非所有运算符都将接受一个以上的先前维度。 线性在这方面并不是唯一的，但通常情况并非如此。

如果我们只想使用单个线性层，那么严格来说，模型类就不是必需的。 该类的真正价值，与大多数现实世界的场景相匹配，是在模型中组合多个运算符，并将其抽象（隐藏）于外部世界。 我们可以做的最简单的添加就是将 ReLU 添加到模型中。

由于 ReLU 和所有激活函数一样，不依赖于可训练的权重，因此无需创建模块并将其保存在字段中，我们可以将其作为函数调用。 ReLU 也有一个模块版本，它有它的用途，但在这种情况下，它是矫枉过正的。

```
private class Trivial : nn.Module
{
    public Trivial()
        : base(nameof(Trivial))
    {
        RegisterComponents();
    }

    public override Tensor forward(Tensor input)
    {
        using var x = lin1.forward(input);
        return nn.functional.relu(x);
```

```
    }

    private nn.Module lin1 = nn.Linear(1000, 100);
}
```

```
var input = rand(1000);
var model = new Trivial();
model.forward(input)
```

```
[100], type = Float32, device = cpu
 0 0.10657 0 0 0.011773 0 0.37909 0 0 0 0.10858 0 0 0.0031786 0 0.24584 0
1.1918 0 0 0.12788 ...
```

```
private class Trivial : nn.Module
{
    public Trivial()
        : base(nameof(Trivial))
    {
        RegisterComponents();
    }

    public override Tensor forward(Tensor input)
    {
        using var x = lin1.forward(input);
        using var y = nn.functional.relu(x);
        return lin2.forward(y);
    }

    private nn.Module lin1 = nn.Linear(1000, 100);
    private nn.Module lin2 = nn.Linear(100, 10);
}
```

```
var model = new Trivial();
model.forward(input)
```

```
[10], type = Float32, device = cpu
 0.11489 -0.11035 0.074485 0.005859 0.064043 0.21116 -0.13207 -0.049393
-0.055822 -0.29107
```

```
var dataBatch = rand(32,1000);  // Our pretend input data
var resultBatch = rand(32,10);  // Our pretend ground truth.
dataBatch.ToString()
```

```
[32x1000], type = Float32, device = cpu
```

```
var loss = nn.functional.mse_loss();
loss(model.forward(dataBatch), resultBatch).item<float>()
```

0.3450216

## 训练

训练包括重复计算损失，然后是梯度，然后在重新开始之前将梯度应用于模型权重。

```
var learning_rate = 0.001f;

// Compute the loss
var output = loss(model.forward(dataBatch), resultBatch);

// Clear the gradients before doing the back-propagation
model.zero_grad();

// Do back-progatation, which computes all the gradients.
output.backward();

// Adjust the weights using the gradients.
using (torch.no_grad()) {
    foreach (var param in model.parameters()) {
        var grad = param.grad();
        if (grad is not null) {
            var update = grad.mul(learning_rate);
            param.sub_(update);
        }
    }
}

loss(model.forward(dataBatch), resultBatch).item<float>()
```

0.34260148

## 优化器

上述训练逻辑的最后一步称为“优化”，它在数学意义上使用：我们正在最小化损失函数，即找到损失最小的输入。 请注意，我们不是针对输入数据将其最小化，而是针对权重进行最小化。

像上面那样显式地写出最小化步骤将变得非常乏味，并且存在比上面简单的优化方法（梯度下降）更复杂的优化方法。 这些被适当地称为“优化器”的类中捕获。 我们可以使用优化器并使用它来调整权重，而不是执行上述操作。

```
var learning_rate = 0.001f;

var optimizer = torch.optim.SGD(model.parameters(), learning_rate);

// Compute the loss
var output = loss(model.forward(dataBatch), resultBatch);

// Clear the gradients before doing the back-propagation
model.zero_grad();

// Do back-progatation, which computes all the gradients.
output.backward();

optimizer.step();

loss(model.forward(dataBatch), resultBatch).item<float>()
```

0.3378535

```
var learning_rate = 0.01f;

var optimizer = torch.optim.SGD(model.parameters(), learning_rate);

// Compute the loss
var output = loss(model.forward(dataBatch), resultBatch);

// Clear the gradients before doing the back-propagation
model.zero_grad();

// Do back-progatation, which computes all the gradients.
output.backward();

optimizer.step();

loss(model.forward(dataBatch), resultBatch).item<float>()
```

0.2763172

```
var optimizer = torch.optim.Adam(model.parameters());

// Compute the loss
var output = loss(model.forward(dataBatch), resultBatch);

// Clear the gradients before doing the back-propagation
```

```
model.zero_grad();

// Do back-progatation, which computes all the gradients.
output.backward();

optimizer.step();

loss(model.forward(dataBatch), resultBatch).item<float>()
```

0.12968187

```
var learning_rate = 0.01f;
model = new Trivial();

var optimizer = torch.optim.SGD(model.parameters(), learning_rate);

for (int i = 0; i < 1000; i++) {
    // Compute the loss
    using var output = loss(model.forward(dataBatch), resultBatch);

    // Clear the gradients before doing the back-propagation
    model.zero_grad();

    // Do back-progatation, which computes all the gradients.
    output.backward();

    optimizer.step();
}

loss(model.forward(dataBatch), resultBatch).item<float>()
```

0.030665454

## 准确性

损失对于优化很有用，但它相对粗略地表明了模型实际上有多好。 事实上，它在很大程度上是无用的。 我们真正关心的是它在做出预测时有多准确。 为此，我们需要计算正确预测的数量并除以批量大小。

为此，我们必须对如何解释模型的输出有所了解。 在这种情况下，每个预测都是一个十元素张量。 我们怎么读？ 让我们决定将模型解释为识别具有最大输出值的索引。 TorchSharp 有一个功能可以做到这一点，我们可以在模型训练后使用它来评估模型。

对于基本事实，以下是最大值：

```
var refMax = resultBatch.argmax(1);
refMax
```

```
[32], type = Int64, device = cpu
 9 7 9 3 9 6 0 2 8 5 2 0 5 8 7 0 8 9 4 9 8 5 1 1 0 3 6 7 5 2 6 9
```

```
var predMax = model.forward(dataBatch).argmax(1);
predMax
```

```
[32], type = Int64, device = cpu
 9 0 1 3 2 6 0 2 8 0 1 0 0 1 6 0 2 1 8 0 7 0 1 0 7 9 6 7 0 8 6 2
```

```
(refMax == predMax).sum() / predMax.numel()
```

```
[], type = Float32, device = cpu, value = 0.375
```

```
var x = rand(64,1000);  // Our pretend input data
var y = rand(64,10);    // Our pretend ground truth.

var yMax = y.argmax(1);
var pMax = model.forward(x).argmax(1);

(yMax == pMax).sum() / refMax.numel()
```

```
[], type = Float32, device = cpu, value = 0.1875
```