1. 用例子学习 PyTorch

> 译者：bat67
>
> 最新版会在译者仓库首先同步。

作者：Justin Johnson

这个教程通过自洽的示例介绍了PyTorch的基本概念。

PyTorch主要是提供了两个核心的功能特性：

- 一个类似于numpy的n维张量，但是可以在GPU上运行
- 搭建和训练神经网络时的自动微分/求导机制

我们将使用全连接的ReLU网络作为运行示例。该网络将有一个单一的隐藏层，并将使用梯度下降训练，通过最小化网络输出和真正结果的欧几里得距离，来拟合随机生成的数据。

1.1. 目录

1.2. 张量

1.2.1. 热身：NumPy

在介绍PyTorch之前，我们将首先使用NumPy实现网络。

NumPy提供了一个n维数组对象和许多用于操作这些数组的函数。NumPy是用于科学计算的通用框架；它对计算图、深度学习和梯度一无所知。然而，我们可以很容易地使用NumPy，手动实现网络的前向和反向传播，来拟合随机数据：

```
# 可运行代码见本文件夹中的 two_layer_net_numpy.py
import numpy as np

# N是批大小；D_in是输入维度
# H是隐藏层维度；D_out是输出维度
N, D_in, H, D_out = 64, 1000, 100, 10

# 产生随机输入和输出数据
x = np.random.randn(N, D_in)
y = np.random.randn(N, D_out)

# 随机初始化权重
```

```
w1 = np.random.randn(D_in, H)
w2 = np.random.randn(H, D_out)

learning_rate = 1e-6
for t in range(500):
    # 前向传播：计算预测值y
    h = x.dot(w1)
    h_relu = np.maximum(h, 0)
    y_pred = h_relu.dot(w2)

    # 计算并显示loss（损失）
    loss = np.square(y_pred - y).sum()
    print(t, loss)

    # 反向传播，计算w1、w2对loss的梯度
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.T.dot(grad_y_pred)
    grad_h_relu = grad_y_pred.dot(w2.T)
    grad_h = grad_h_relu.copy()
    grad_h[h < 0] = 0
    grad_w1 = x.T.dot(grad_h)

    # 更新权重
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

1.2.2. PyTorch：张量

NumPy是一个很棒的框架，但是它不支持GPU以加速运算。现代深度神经网络，GPU常常提供50倍以上的加速，所以NumPy不能满足当代深度学习的需求。

我们先介绍PyTorch最基础的概念：**张量（Tensor）**。逻辑上，PyTorch的tensor和NumPy array是一样的：tensor是一个n维数组，PyTorch提供了很多函数操作这些tensor。任何希望使用NumPy执行的计算也可以使用PyTorch的tensor来完成；可以认为它们是科学计算的通用工具。

和NumPy不同的是，PyTorch可以利用GPU加速。要在GPU上运行PyTorch张量，在构造张量使用`device`参数把tensor建立在GPU上。

这里我们利用PyTorch的tensor在随机数据上训练一个两层的网络。和前面NumPy的例子类似，我们使用PyTorch的tensor，手动在网络中实现前向传播和反向传播：

```
# 可运行代码见本文件夹中的 two_layer_net_tensor.py
import torch

```

```
device = torch.device('cuda' if torch.cuda.is_available()
else 'cpu')

# N是批大小； D_in 是输入维度；
# H 是隐藏层维度； D_out 是输出维度
N, D_in, H, D_out = 64, 1000, 100, 10

# 产生随机输入和输出数据
x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)

# 随机初始化权重
w1 = torch.randn(D_in, H, device=device)
w2 = torch.randn(H, D_out, device=device)

learning_rate = 1e-6
for t in range(500):
    # 前向传播：计算预测值y
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)

    # 计算并输出loss；loss是存储在PyTorch的tensor中的标量，维度是()
（零维标量）；
    # 我们使用loss.item()得到tensor中的纯python数值。
    loss = (y_pred - y).pow(2).sum()
    print(t, loss.item())

    # 反向传播，计算w1、w2对loss的梯度
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    # 使用梯度下降更新权重
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

1.3. 自动求导

1.3.1. PyTorch：张量和自动求导

在上面的例子里，需要我们手动实现神经网络的前向和后向传播。对于简单的两层网络，手动实现前向、后向传播不是什么难事，但是对于大型的复杂网络就比较麻烦了。

庆幸的是，我们可以使用自动微分来自动完成神经网络中反向传播的计算。PyTorch中**autograd**包提供的正是这个功能。当使用autograd时，网络前向传播将定义一个**计算图**；图中的节点是tensor，边是函数，这些函数是输出tensor到输入tensor的映射。这张计算图使得在网络中反向传播时梯度的计算十分简单。

这听起来复杂，但是实际操作很简单。如果我们想计算某些的tensor的梯度，我们只需要在建立这个tensor时加入这么一句：`requires_grad=True`。这个tensor上的任何PyTorch的操作都将构造一个计算图，从而允许我们稍后在图中执行反向传播。如果这个tensor`x`的`requires_grad=True`，那么反向传播之后`x.grad`将会是另一个张量，其为`x`关于某个标量值的梯度。

有时可能希望防止PyTorch在`requires_grad=True`的张量执行某些操作时构建计算图；例如，在训练神经网络时，我们通常不希望通过权重更新步骤进行反向传播。在这种情况下，我们可以使用`torch.no_grad()`上下文管理器来防止构造计算图。

下面我们使用PyTorch的Tensors和autograd来实现我们的两层的神经网络；我们不再需要手动执行网络的反向传播：

```
# 可运行代码见本文件夹中的 two_layer_net_autograd.py
import torch

device = torch.device('cuda' if torch.cuda.is_available()
else 'cpu')

# N是批大小；D_in是输入维度；
# H是隐藏层维度；D_out是输出维度
N, D_in, H, D_out = 64, 1000, 100, 10

# 产生随机输入和输出数据
x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)

# 产生随机权重tensor，将requires_grad设置为True意味着我们希望在反向传播时候计算这些值的梯度
w1 = torch.randn(D_in, H, device=device, requires_grad=True)
w2 = torch.randn(H, D_out, device=device, requires_grad=True)

learning_rate = 1e-6
for t in range(500):

    # 前向传播：使用tensor的操作计算预测值y。
```

```
    # 由于w1和w2有requires_grad=True，涉及这些张量的操作将让PyTorch构建计算图，
    # 从而允许自动计算梯度。由于我们不再手工实现反向传播，所以不需要保留中间值的引用。
    y_pred = x.mm(w1).clamp(min=0).mm(w2)

    # 计算并输出loss，loss是一个形状为()的张量，loss.item()是这个张量对应的python数值
    loss = (y_pred - y).pow(2).sum()
    print(t, loss.item())

    # 使用autograd计算反向传播。这个调用将计算loss对所有requires_grad=True的tensor的梯度。
    # 这次调用后，w1.grad和w2.grad将分别是loss对w1和w2的梯度张量。
    loss.backward()


    # 使用梯度下降更新权重。对于这一步，我们只想对w1和w2的值进行原地改变；不想为更新阶段构建计算图，
    # 所以我们使用torch.no_grad()上下文管理器防止PyTorch为更新构建计算图
    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad

        # 反向传播之后手动置零梯度
        w1.grad.zero_()
        w2.grad.zero_()
```

1.3.2. PyTorch：定义新的自动求导函数

在底层，每一个原始的自动求导运算实际上是两个在Tensor上运行的函数。其中，**forward**函数计算从输入Tensors获得的输出Tensors。而**backward**函数接收输出Tensors对于某个标量值的梯度，并且计算输入Tensors相对于该相同标量值的梯度。

在PyTorch中，我们可以很容易地通过定义`torch.autograd.Function`的子类并实现`forward`和`backward`函数，来定义自己的自动求导运算。之后我们就可以使用这个新的自动梯度运算符了。然后，我们可以通过构造一个实例并像调用函数一样，传入包含输入数据的tensor调用它，这样来使用新的自动求导运算。

这个例子中，我们自定义一个自动求导函数来展示ReLU的非线性。并用它实现我们的两层网络：

```
# 可运行代码见本文件夹中的 two_layer_net_custom_function.py
```

```
import torch

class MyReLU(torch.autograd.Function):
    """
    我们可以通过建立torch.autograd的子类来实现我们自定义的autograd函数，
    并完成张量的正向和反向传播。
    """
    @staticmethod
    def forward(ctx, x):
        """
        在正向传播中，我们接收到一个上下文对象和一个包含输入的张量；
        我们必须返回一个包含输出的张量，
        并且我们可以使用上下文对象来缓存对象，以便在反向传播中使用。
        """
        ctx.save_for_backward(x)
        return x.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_output):
        """
        在反向传播中，我们接收到上下文对象和一个张量，
        其包含了相对于正向传播过程中产生的输出的损失的梯度。
        我们可以从上下文对象中检索缓存的数据，
        并且必须计算并返回与正向传播的输入相关的损失的梯度。
        """
        x, = ctx.saved_tensors
        grad_x = grad_output.clone()
        grad_x[x < 0] = 0
        return grad_x


device = torch.device('cuda' if torch.cuda.is_available()
else 'cpu')

# N是批大小； D_in 是输入维度；
# H 是隐藏层维度； D_out 是输出维度
N, D_in, H, D_out = 64, 1000, 100, 10

# 产生输入和输出的随机张量
x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)

# 产生随机权重的张量
w1 = torch.randn(D_in, H, device=device, requires_grad=True)
```

```
w2 = torch.randn(H, D_out, device=device, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    # 正向传播：使用张量上的操作来计算输出值y；
    # 我们通过调用 MyReLU.apply 函数来使用自定义的ReLU
    y_pred = MyReLU.apply(x.mm(w1)).mm(w2)

    # 计算并输出loss
    loss = (y_pred - y).pow(2).sum()
    print(t, loss.item())

    # 使用autograd计算反向传播过程。
    loss.backward()

    with torch.no_grad():
        # 用梯度下降更新权重
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad

        # 在反向传播之后手动清零梯度
        w1.grad.zero_()
        w2.grad.zero_()

```

1.3.3. TensorFlow：静态图

PyTorch自动求导看起来非常像TensorFlow：这两个框架中，我们都定义计算图，使用自动微分来计算梯度。两者最大的不同就是TensorFlow的计算图是**静态的**，而PyTorch使用**动态的**计算图。

在TensorFlow中，我们定义计算图一次，然后重复执行这个相同的图，可能会提供不同的输入数据。而在PyTorch中，每一个前向通道定义一个新的计算图。

静态图的好处在于你可以预先对图进行优化。例如，一个框架可能要融合一些图的运算来提升效率，或者产生一个策略来将图分布到多个GPU或机器上。如果重复使用相同的图，那么在重复运行同一个图时，，前期潜在的代价高昂的预先优化的消耗就会被分摊开。

静态图和动态图的一个区别是控制流。对于一些模型，我们希望对每个数据点执行不同的计算。例如，一个递归神经网络可能对于每个数据点执行不同的时间步数，这个展开（unrolling）可以作为一个循环来实现。对于一个静态图，循环结构要作为图的一部分。因此，TensorFlow提供了运算符（例如`tf.scan`）来把循环嵌入到图当中。对于动态图来说，情况更加简单：既然我们为每个例子即时创建图，我们可以使用普通的命令式控制流来为每个输入执行不同的计算。

为了与上面的PyTorch自动梯度实例做对比，我们使用TensorFlow来拟合一个简单的2层网络：

```
# 可运行代码见本文件夹中的 tf_two_layer_net.py
import tensorflow as tf
import numpy as np

# 首先我们建立计算图（computational graph）

# N是批大小；D是输入维度；
# H是隐藏层维度；D_out是输出维度。
N, D_in, H, D_out = 64, 1000, 100, 10

# 为输入和目标数据创建placeholder；
# 当执行计算图时，他们将会被真实的数据填充
x = tf.placeholder(tf.float32, shape=(None, D_in))
y = tf.placeholder(tf.float32, shape=(None, D_out))

# 为权重创建Variable并用随机数据初始化
# TensorFlow的Variable在执行计算图时不会改变
w1 = tf.Variable(tf.random_normal((D_in, H)))
w2 = tf.Variable(tf.random_normal((H, D_out)))

# 前向传播：使用TensorFlow的张量运算计算预测值y。
# 注意这段代码实际上不执行任何数值运算；
# 它只是建立了我们稍后将执行的计算图。
h = tf.matmul(x, w1)
h_relu = tf.maximum(h, tf.zeros(1))
y_pred = tf.matmul(h_relu, w2)

# 使用TensorFlow的张量运算损失（loss）
loss = tf.reduce_sum((y - y_pred) ** 2.0)

# 计算loss对于w1和w2的导数
grad_w1, grad_w2 = tf.gradients(loss, [w1, w2])

# 使用梯度下降更新权重。为了实际更新权重，我们需要在执行计算图时计算new_w1和new_w2。
# 注意，在TensorFlow中，更新权重值的行为是计算图的一部分；
# 但在PyTorch中，这发生在计算图形之外。
learning_rate = 1e-6
new_w1 = w1.assign(w1 - learning_rate * grad_w1)
new_w2 = w2.assign(w2 - learning_rate * grad_w2)

```

```
# 现在我们搭建好了计算图，所以我们开始一个TensorFlow的会话（session）来实际执行计算图。
with tf.Session() as sess:

    # 运行一次计算图来初始化Variable w1和w2
    sess.run(tf.global_variables_initializer())

    # 创建numpy数组来存储输入x和目标y的实际数据
    x_value = np.random.randn(N, D_in)
    y_value = np.random.randn(N, D_out)

    for _ in range(500):
        # 多次运行计算图。每次执行时，我们都用feed_dict参数，
        # 将x_value绑定到x，将y_value绑定到y，
        # 每次执行图形时我们都要计算损失、new_w1和new_w2；
        # 这些张量的值以numpy数组的形式返回。
        loss_value, _, _ = sess.run([loss, new_w1, new_w2],
                                    feed_dict={x: x_value, y: y_value})
        print(loss_value)
```

1.4. nn模块

1.4.1. PyTorch：nn

计算图和autograd是十分强大的工具，可以定义复杂的操作并自动求导；然而对于大规模的网络，autograd太过于底层。

在构建神经网络时，我们经常考虑将计算安排成**层**，其中一些具有**可学习的参数**，它们将在学习过程中进行优化。

TensorFlow里，有类似Keras，TensorFlow-Slim和TFLearn这种封装了底层计算图的高度抽象的接口，这使得构建网络十分方便。

在PyTorch中，包`nn`完成了同样的功能。`nn`包中定义一组大致等价于层的**模块**。一个模块接受输入的tesnor，计算输出的tensor，而且还保存了一些内部状态比如需要学习的tensor的参数等。`nn`包中也定义了一组损失函数（loss functions），用来训练神经网络。

这个例子中，我们用`nn`包实现两层的网络：

```
# 可运行代码见本文件夹中的 two_layer_net_nn.py
import torch

```

```
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

# N是批大小；D是输入维度
# H是隐藏层维度；D_out是输出维度
N, D_in, H, D_out = 64, 1000, 100, 10

# 产生输入和输出随机张量
x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)


# 使用nn包将我们的模型定义为一系列的层。
# nn.Sequential是包含其他模块的模块，并按顺序应用这些模块来产生其输出。
# 每个线性模块使用线性函数从输入计算输出，并保存其内部的权重和偏差张量。
# 在构造模型之后，我们使用.to()方法将其移动到所需的设备。
model = torch.nn.Sequential(
            torch.nn.Linear(D_in, H),
            torch.nn.ReLU(),
            torch.nn.Linear(H, D_out),
        ).to(device)


# nn包还包含常用的损失函数的定义；
# 在这种情况下，我们将使用平均平方误差(MSE)作为我们的损失函数。
# 设置reduction='sum'，表示我们计算的是平方误差的“和”，而不是平均值；
# 这是为了与前面我们手工计算损失的例子保持一致，
# 但是在实践中，通过设置reduction='elementwise_mean'来使用均方误差作为损失更为常见。
loss_fn = torch.nn.MSELoss(reduction='sum')

learning_rate = 1e-4
for t in range(500):

    # 前向传播：通过向模型传入x计算预测的y。
    # 模块对象重载了__call__运算符，所以可以像函数那样调用它们。
    # 这么做相当于向模块传入了一个张量，然后它返回了一个输出张量。
    y_pred = model(x)

    # 计算并打印损失。我们传递包含y的预测值和真实值的张量，损失函数返回包含损失的张量。
    loss = loss_fn(y_pred, y)
    print(t, loss.item())

    # 反向传播之前清零梯度
```

```
    model.zero_grad()

    # 反向传播：计算模型的损失对所有可学习参数的导数（梯度）。
    # 在内部，每个模块的参数存储在requires_grad=True的张量中，
    # 因此这个调用将计算模型中所有可学习参数的梯度。
    loss.backward()

    # 使用梯度下降更新权重。
    # 每个参数都是张量，所以我们可以像我们以前那样可以得到它的数值和梯度
    with torch.no_grad():
        for param in model.parameters():
            param.data -= learning_rate * param.grad
```

1.4.2. PyTorch: optim

到目前为止，我们已经通过手动改变包含可学习参数的张量来更新模型的权重。对于随机梯度下降(SGD/stochastic gradient descent)等简单的优化算法来说，这不是一个很大的负担，但在实践中，我们经常使用AdaGrad、RMSProp、Adam等更复杂的优化器来训练神经网络。

```
# 可运行代码见本文件夹中的 two_layer_net_optim.py
import torch

# N是批大小；D是输入维度
# H是隐藏层维度；D_out是输出维度
N, D_in, H, D_out = 64, 1000, 100, 10

# 产生随机输入和输出张量
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# 使用nn包定义模型和损失函数
model = torch.nn.Sequential(
          torch.nn.Linear(D_in, H),
          torch.nn.ReLU(),
          torch.nn.Linear(H, D_out),
        )
loss_fn = torch.nn.MSELoss(reduction='sum')

# 使用optim包定义优化器（Optimizer）。Optimizer将会为我们更新模型的权重。
# 这里我们使用Adam优化方法；optim包还包含了许多别的优化算法。
# Adam构造函数的第一个参数告诉优化器应该更新哪些张量。
learning_rate = 1e-4
```

```
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

for t in range(500):

    # 前向传播：通过像模型输入x计算预测的y
    y_pred = model(x)

    # 计算并打印loss
    loss = loss_fn(y_pred, y)
    print(t, loss.item())

    # 在反向传播之前，使用optimizer将它要更新的所有张量的梯度清零(这些张量是模型可学习的权重)
    optimizer.zero_grad()

    # 反向传播：根据模型的参数计算loss的梯度
    loss.backward()

    # 调用Optimizer的step函数使它所有参数更新
    optimizer.step()
```

1.4.3. PyTorch：自定义nn模块

有时候需要指定比现有模块序列更复杂的模型；对于这些情况，可以通过继承`nn.Module`并定义`forward`函数，这个`forward`函数可以使用其他模块或者其他的自动求导运算来接收输入tensor，产生输出tensor。

在这个例子中，我们用自定义Module的子类构建两层网络：

```
# 可运行代码见本文件夹中的 two_layer_net_module.py
import torch

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        """
        在构造函数中，我们实例化了两个nn.Linear模块，并将它们作为成员变量。
        """
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        """
```

```
        在前向传播的函数中，我们接收一个输入的张量，也必须返回一个输出张量。
        我们可以使用构造函数中定义的模块以及张量上的任意的（可微分的）操作。
        """
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

# N是批大小； D_in 是输入维度；
# H 是隐藏层维度； D_out 是输出维度
N, D_in, H, D_out = 64, 1000, 100, 10

# 产生输入和输出的随机张量
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# 通过实例化上面定义的类来构建我们的模型。
model = TwoLayerNet(D_in, H, D_out)

# 构造损失函数和优化器。
# SGD构造函数中对model.parameters()的调用，
# 将包含模型的一部分，即两个nn.Linear模块的可学习参数。
loss_fn = torch.nn.MSELoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    # 前向传播：通过向模型传递x计算预测值y
    y_pred = model(x)

    #计算并输出loss
    loss = loss_fn(y_pred, y)
    print(t, loss.item())

    # 清零梯度，反向传播，更新权重
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

```

1.4.4. PyTorch：控制流和权重共享

作为动态图和权重共享的一个例子，我们实现了一个非常奇怪的模型：一个全连接的ReLU网络，在每一次前向传播时，它的隐藏层的层数为随机1到4之间的数，这样可以多次重用相同的权重来计算。

因为这个模型可以使用普通的Python流控制来实现循环，并且我们可以通过在定义转发时多次重用同一个模块来实现最内层之间的权重共享。

我们利用Mudule的子类很容易实现这个模型：

```
# 可运行代码见本文件夹中的 dynamic_net.py
import random
import torch

class DynamicNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        """
        在构造函数中，我们构造了三个nn.Linear实例，它们将在前向传播时被使用。
        """
        super(DynamicNet, self).__init__()
        self.input_linear = torch.nn.Linear(D_in, H)
        self.middle_linear = torch.nn.Linear(H, H)
        self.output_linear = torch.nn.Linear(H, D_out)

    def forward(self, x):
        """
        对于模型的前向传播，我们随机选择0、1、2、3，
        并重用了多次计算隐藏层的middle_linear模块。
        由于每个前向传播构建一个动态计算图，
        我们可以在定义模型的前向传播时使用常规Python控制流运算符，如循环或条件语句。
        在这里，我们还看到，在定义计算图形时多次重用同一个模块是完全安全的。
        这是Lua Torch的一大改进，因为Lua Torch中每个模块只能使用一次。
        """
        h_relu = self.input_linear(x).clamp(min=0)
        for _ in range(random.randint(0, 3)):
            h_relu = self.middle_linear(h_relu).clamp(min=0)
        y_pred = self.output_linear(h_relu)
        return y_pred


# N是批大小；D是输入维度
# H是隐藏层维度；D_out是输出维度
N, D_in, H, D_out = 64, 1000, 100, 10

# 产生输入和输出随机张量
x = torch.randn(N, D_in)
```

```
y = torch.randn(N, D_out)

# 实例化上面定义的类来构造我们的模型
model = DynamicNet(D_in, H, D_out)

# 构造我们的损失函数（loss function）和优化器（Optimizer）。
# 用平凡的随机梯度下降训练这个奇怪的模型是困难的，所以我们使用了momentum方法。
criterion = torch.nn.MSELoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4, momentum=0.9)
for t in range(500):

    # 前向传播：通过向模型传入x计算预测的y。
    y_pred = model(x)

    # 计算并打印损失
    loss = criterion(y_pred, y)
    print(t, loss.item())

    # 清零梯度，反向传播，更新权重
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

1.5. Examples

You can browse the above examples here.

1.5.1. Tensors

Warm-up: numpy

PyTorch: Tensors

1.5.2. Autograd

PyTorch: Tensors and autograd

PyTorch: Defining New autograd Functions

TensorFlow: Static Graphs

1.5.3. `nn` module

PyTorch: nn

PyTorch: optim

PyTorch: Custom nn Modules

PyTorch: Control Flow + Weight Sharing