半小时学会 PyTorch Hook

尹相楠
博士在读

取消关注

提到 hook，我首先想起的是动画《小飞侠》里滑稽的 captain hook，满满童年的回忆促使我 P 了张题图：虎克船长勾着 PyTorch 的 logo。同时想起的还有大名鼎鼎的胡克定律：Hooke's law（虽然不是一个 hook），当年上物理实验课，看着弹簧测力计下面的钩子，联想到胡克被牛顿爵士打压的悲惨一生，不由发出既生胡何生牛的唏嘘……然而本文将介绍的是 PyTorch 中的 hook。

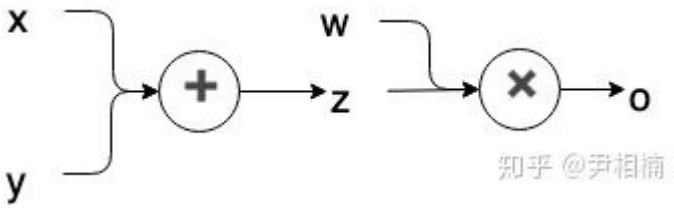
首先贴一段维基百科中对钩子的定义：

> 钩子编程（hooking），也称作“挂钩”，是计算机程序设计术语，指通过拦截软件模块间的函数调用、消息传递、事件传递来修改或扩展操作系统、应用程序或其他软件组件的行为的各种技术。处理被拦截的函数调用、事件、消息的代码，被称为钩子（hook）。

Hook 是 PyTorch 中一个十分有用的特性。利用它，我们可以**不必改变网络输入输出的结构，方便地获取、改变网络中间层变量的值和梯度**。这个功能被广泛用于可视化神经网络中间层的 feature、gradient，从而诊断神经网络中可能出现的问题，分析网络有效性。本文将结合代码，由浅入深地介绍 pytorch 中 hook 的用法。文章分为三部分：

1. **Hook for Tensors**：针对 Tensor 的 hook
2. **Hook for Modules**：针对例如 `nn.Conv2d` `nn.Linear` 等网络模块的 hook
3. **Guided Backpropagation**：利用 Hook 实现的一段神经网络可视化代码

Hook for Tensors



上面的计算图中，x y w 为叶子节点，而 z 为中间变量

在 PyTorch 的计算图（computation graph）中，只有叶子结点（leaf nodes）的变量会保留梯度。而所有中间变量的梯度只被用于反向传播，一旦完成反向传播，中间变量的梯度就将自动释放，从而节约内存。如下面这段代码所示：

```
import torch

x = torch.Tensor([0, 1, 2, 3]).requires_grad_()
y = torch.Tensor([4, 5, 6, 7]).requires_grad_()
w = torch.Tensor([1, 2, 3, 4]).requires_grad_()
z = x+y
# z.retain_grad()

o = w.matmul(z)
o.backward()
# o.retain_grad()

print('x.requires_grad:', x.requires_grad) # True
print('y.requires_grad:', y.requires_grad) # True
print('z.requires_grad:', z.requires_grad) # True
print('w.requires_grad:', w.requires_grad) # True
print('o.requires_grad:', o.requires_grad) # True


print('x.grad:', x.grad) # tensor([1., 2., 3., 4.])
print('y.grad:', y.grad) # tensor([1., 2., 3., 4.])
print('w.grad:', w.grad) # tensor([ 4.,  6.,  8., 10.])
print('z.grad:', z.grad) # None
print('o.grad:', o.grad) # None
```

由于 z 和 o 为中间变量（并非直接指定数值的变量，而是由别的变量计算得到的变量），它们虽然 `requires_grad` 的参数都是 True，但是反向传播后，它们的梯度并没有保存下来，而是直接删除了，因此是 None。如果想在反向传播之后保留它们的梯度，则需要特殊指定：把上面代码中的 `z.retain_grad()` 和 `o.retain_grad` 的注释去掉，可以得到它们对应的梯度，运行结果如下所示：

```
x.requires_grad: True
y.requires_grad: True
```

```
z.requires_grad: True
w.requires_grad: True
o.requires_grad: True
x.grad: tensor([1., 2., 3., 4.])
y.grad: tensor([1., 2., 3., 4.])
w.grad: tensor([ 4.,  6.,  8., 10.])
z.grad: tensor([1., 2., 3., 4.])
o.grad: tensor(1.)
```

但是，这种加 `retain_grad()` 的方案会增加内存占用，并不是个好办法，对此的一种替代方案，就是用 hook 保存中间变量的梯度。

对于中间变量 `z`，hook 的使用方式为：`z.register_hook(hook_fn)`，其中 `hook_fn` 为一个用户自定义的函数，其签名为：

```
hook_fn(grad) -> Tensor or None
```

它的输入为变量 `z` 的梯度，输出为一个 Tensor 或者是 None （None 一般用于直接打印梯度）。反向传播时，梯度传播到变量 z，再继续向前传播之前，将会传入 `hook_fn`。如果 `hook_fn` 的返回值是 None，那么梯度将不改变，继续向前传播，如果 `hook_fn` 的返回值是 Tensor 类型，则该 Tensor 将取代 z 原有的梯度，向前传播。

下面的示例代码中 `hook_fn` 不改变梯度值，仅仅是打印梯度：

```
import torch

x = torch.Tensor([0, 1, 2, 3]).requires_grad_()
y = torch.Tensor([4, 5, 6, 7]).requires_grad_()
w = torch.Tensor([1, 2, 3, 4]).requires_grad_()
z = x+y

# ==================
def hook_fn(grad):
    print(grad)

z.register_hook(hook_fn)
# ==================

o = w.matmul(z)

print('=====Start backprop=====')
o.backward()
print('=====End backprop=====')

print('x.grad:', x.grad)
```

```
print('y.grad:', y.grad)
print('w.grad:', w.grad)
print('z.grad:', z.grad)
```

运行结果如下：

```
=====Start backprop=====
tensor([1., 2., 3., 4.])
=====End backprop=====
x.grad: tensor([1., 2., 3., 4.])
y.grad: tensor([1., 2., 3., 4.])
w.grad: tensor([ 4.,  6.,  8., 10.])
z.grad: None
```

我们发现，`z` 绑定了 `hook_fn` 后，梯度反向传播时将会打印出 `o` 对 `z` 的偏导，和上文中 `z.retain_grad()` 方法得到的 z 的偏导一致。

接下来可以试一下，在 `hook_fn` 中改变梯度值，看看会有什么结果。

```
import torch

x = torch.Tensor([0, 1, 2, 3]).requires_grad_()
y = torch.Tensor([4, 5, 6, 7]).requires_grad_()
w = torch.Tensor([1, 2, 3, 4]).requires_grad_()
z = x + y


# ===================
def hook_fn(grad):
    g = 2 * grad
    print(g)
    return g


z.register_hook(hook_fn)
# ===================

o = w.matmul(z)

print('=====Start backprop=====')
o.backward()
print('=====End backprop=====')

print('x.grad:', x.grad)
print('y.grad:', y.grad)
```

```
print('w.grad:', w.grad)
print('z.grad:', z.grad)
```

运行结果如下：

```
=====Start backprop=====
tensor([2., 4., 6., 8.])
=====End backprop=====
x.grad: tensor([2., 4., 6., 8.])
y.grad: tensor([2., 4., 6., 8.])
w.grad: tensor([ 4.,  6.,  8., 10.])
z.grad: None
```

发现 `z` 的梯度变为两倍后，受其影响，`x` 和 `y` 的梯度也都变成了原来的两倍。

在实际代码中，为了方便，也可以用 lambda 表达式来代替函数，简写为如下形式：

```
import torch

x = torch.Tensor([0, 1, 2, 3]).requires_grad_()
y = torch.Tensor([4, 5, 6, 7]).requires_grad_()
w = torch.Tensor([1, 2, 3, 4]).requires_grad_()
z = x + y

# ===================
z.register_hook(lambda x: 2*x)
z.register_hook(lambda x: print(x))
# ===================

o = w.matmul(z)

print('=====Start backprop=====')
o.backward()
print('=====End backprop=====')

print('x.grad:', x.grad)
print('y.grad:', y.grad)
print('w.grad:', w.grad)
print('z.grad:', z.grad)
```

运行结果和上面的代码相同，我们发现一个变量可以绑定多个 `hook_fn`，反向传播时，它们按绑定顺序依次执行。例如上面的代码中，第一个绑定的 `hook_fn` 把 `z` 的梯度乘以2，第二个绑定的 `hook_fn` 打印 `z` 的梯度。因此反向传播时，也是按照这个顺序执行的，打印出来的 `z` 的梯度值，是其原本梯度值的两倍。

至此，针对对 Tensor 的 hook 就介绍完了。然而它的使用场景一般不多，最常用的 hook 是针对神经网络模块的。

Hook for Modules

网络模块 module 不像上一节中的 Tensor，拥有显式的变量名可以直接访问，而是被封装在神经网络中间。我们通常只能获得网络整体的输入和输出，对于夹在网络中间的模块，我们不但很难得知它输入/输出的梯度，甚至连它输入输出的数值都无法获得。除非设计网络时，在 forward 函数的返回值中包含中间 module 的输出，或者用很麻烦的办法，把网络按照 module 的名称拆分再组合，让中间层提取的 feature 暴露出来。

为了解决这个麻烦，PyTorch 设计了两种 hook：`register_forward_hook` 和 `register_backward_hook`，分别用来获取正/反向传播时，中间层模块输入和输出的 feature/gradient，大大降低了获取模型内部信息流的难度。

register forward hook

`register_forward_hook` 的作用是获取前向传播过程中，各个网络模块的输入和输出。对于模块 `module`，其使用方式为：`module.register_forward_hook(hook_fn)`。其中 `hook_fn` 的签名为：

```
hook_fn(module, input, output) -> None
```

它的输入变量分别为：模块，模块的输入，模块的输出，和对 Tensor 的 hook 不同，forward hook **不返回任何值**，也就是说**不能**用它来修改输入或者输出的值，但借助这个 hook，我们可以方便地用预训练的神经网络提取特征，而不用改变预训练网络的结构。下面提供一段示例代码：

```
import torch
from torch import nn

# 首先我们定义一个模型
class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.fc1 = nn.Linear(3, 4)
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(4, 1)
        self.initialize()

    # 为了方便验证，我们将指定特殊的weight和bias
    def initialize(self):
        with torch.no_grad():
            self.fc1.weight = torch.nn.Parameter(
                torch.Tensor([[1., 2., 3.],
```

```
                                [-4., -5., -6.],
                                [7., 8., 9.],
                                [-10., -11., -12.]]))

            self.fc1.bias = torch.nn.Parameter(torch.Tensor([1.0, 2.0, 3.0, 4.0]))
            self.fc2.weight = torch.nn.Parameter(torch.Tensor([[1.0, 2.0, 3.0, 4.0]]))
            self.fc2.bias = torch.nn.Parameter(torch.Tensor([1.0]))

    def forward(self, x):
        o = self.fc1(x)
        o = self.relu1(o)
        o = self.fc2(o)
        return o

# 全局变量，用于存储中间层的 feature
total_feat_out = []
total_feat_in = []

# 定义 forward hook function
def hook_fn_forward(module, input, output):
    print(module) # 用于区分模块
    print('input', input) # 首先打印出来
    print('output', output)
    total_feat_out.append(output) # 然后分别存入全局 list 中
    total_feat_in.append(input)


model = Model()

modules = model.named_children() #
for name, module in modules:
    module.register_forward_hook(hook_fn_forward)

# 注意下面代码中 x 的维度，对于linear module，输入一定是大于等于二维的
# （第一维是 batch size）。在 forward hook 中看不出来，但是 backward hook 中，
# 得到的梯度完全不对。
# 有一篇 hook 的教程就是这里出了错，作者还强行解释

x = torch.Tensor([[1.0, 1.0, 1.0]]).requires_grad_()
o = model(x)
o.backward()

print('==========Saved inputs and outputs==========')
for idx in range(len(total_feat_in)):
    print('input: ', total_feat_in[idx])
    print('output: ', total_feat_out[idx])
```

运行结果为：

```
Linear(in_features=3, out_features=4, bias=True)
input (tensor([[1., 1., 1.]], requires_grad=True),)
output tensor([[  7., -13.,  27., -29.]], grad_fn=<AddmmBackward>)
ReLU()
input (tensor([[  7., -13.,  27., -29.]], grad_fn=<AddmmBackward>),)
output tensor([[ 7.,  0., 27.,  0.]], grad_fn=<ThresholdBackward0>)
Linear(in_features=4, out_features=1, bias=True)
input (tensor([[ 7.,  0., 27.,  0.]], grad_fn=<ThresholdBackward0>),)
output tensor([[89.]], grad_fn=<AddmmBackward>)
==========Saved inputs and outputs==========
input:  (tensor([[1., 1., 1.]], requires_grad=True),)
output:  tensor([[  7., -13.,  27., -29.]], grad_fn=<AddmmBackward>)
input:  (tensor([[  7., -13.,  27., -29.]], grad_fn=<AddmmBackward>),)
output:  tensor([[ 7.,  0., 27.,  0.]], grad_fn=<ThresholdBackward0>)
input:  (tensor([[ 7.,  0., 27.,  0.]], grad_fn=<ThresholdBackward0>),)
output:  tensor([[89.]], grad_fn=<AddmmBackward>)
```

读者可以用笔验证一下，这里限于篇幅，就不做验证了。

register backward hook

和 `register_forward_hook` 相似， `register_backward_hook` 的作用是获取神经网络反向传播过程中，各个模块**输入端和输出端的梯度值**。对于模块 module，其使用方式为：`module.register_backward_hook(hook_fn)` 。其中 `hook_fn` 的函数签名为：

```
hook_fn(module, grad_input, grad_output) -> Tensor or None
```

它的输入变量分别为：模块，模块输入端的梯度，模块输出端的梯度。需要注意的是，这里的**输入端**和**输出端**，是站在前向传播的角度的，而不是反向传播的角度。例如线性模块： `o=W*x+b` ，其输入端为 W，x 和 b，输出端为 o。

如果模块有多个输入或者输出的话， `grad_input` 和 `grad_output` 可以是 tuple 类型。对于线性模块： `o=W*x+b` ，它的输入端包括了W、x 和 b 三部分，因此 `grad_input` 就是一个包含三个元素的 tuple。

这里注意和 forward hook 的不同：

1. 在 forward hook 中，input 是 x，而不包括 W 和 b。
2. 返回 Tensor 或者 None，backward hook 函数不能直接改变它的输入变量，但是可以返回新的 `grad_input` ，反向传播到它上一个模块。

Talk is cheap，下面看示例代码：

```python
import torch
from torch import nn


class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.fc1 = nn.Linear(3, 4)
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(4, 1)
        self.initialize()

    def initialize(self):
        with torch.no_grad():
            self.fc1.weight = torch.nn.Parameter(
                torch.Tensor([[1., 2., 3.],
                              [-4., -5., -6.],
                              [7., 8., 9.],
                              [-10., -11., -12.]]))

            self.fc1.bias = torch.nn.Parameter(torch.Tensor([1.0, 2.0, 3.0, 4.0]))
            self.fc2.weight = torch.nn.Parameter(torch.Tensor([[1.0, 2.0, 3.0, 4.0]]))
            self.fc2.bias = torch.nn.Parameter(torch.Tensor([1.0]))

    def forward(self, x):
        o = self.fc1(x)
        o = self.relu1(o)
        o = self.fc2(o)
        return o


total_grad_out = []
total_grad_in = []


def hook_fn_backward(module, grad_input, grad_output):
    print(module) # 为了区分模块
    # 为了符合反向传播的顺序，我们先打印 grad_output
    print('grad_output', grad_output)
    # 再打印 grad_input
    print('grad_input', grad_input)
    # 保存到全局变量
    total_grad_in.append(grad_input)
    total_grad_out.append(grad_output)


model = Model()
```

```
modules = model.named_children()
for name, module in modules:
    module.register_backward_hook(hook_fn_backward)

# 这里的 requires_grad 很重要，如果不加，backward hook
# 执行到第一层，对 x 的导数将为 None，某英文博客作者这里疏忽了
# 此外再强调一遍 x 的维度，一定不能写成 torch.Tensor([1.0, 1.0, 1.0]).requires_grad_()
# 否则 backward hook 会出问题。
x = torch.Tensor([[1.0, 1.0, 1.0]]).requires_grad_()
o = model(x)
o.backward()

print('==========Saved inputs and outputs==========')
for idx in range(len(total_grad_in)):
    print('grad output: ', total_grad_out[idx])
    print('grad input: ', total_grad_in[idx])
```

运行后的输出为：

```
Linear(in_features=4, out_features=1, bias=True)
grad_output (tensor([[1.]]),)
grad_input (tensor([1.]), tensor([[1., 2., 3., 4.]]), tensor([[ 7.],
        [ 0.],
        [27.],
        [ 0.]]))
ReLU()
grad_output (tensor([[1., 2., 3., 4.]]),)
grad_input (tensor([[1., 0., 3., 0.]]),)
Linear(in_features=3, out_features=4, bias=True)
grad_output (tensor([[1., 0., 3., 0.]]),)
grad_input (tensor([1., 0., 3., 0.]), tensor([[22., 26., 30.]]), tensor([[1., 0., 3.,
        [1., 0., 3., 0.],
        [1., 0., 3., 0.]]))
==========Saved inputs and outputs==========
grad output:  (tensor([[1.]]),)
grad input:  (tensor([1.]), tensor([[1., 2., 3., 4.]]), tensor([[ 7.],
        [ 0.],
        [27.],
        [ 0.]]))
grad output:  (tensor([[1., 2., 3., 4.]]),)
grad input:  (tensor([[1., 0., 3., 0.]]),)
grad output:  (tensor([[1., 0., 3., 0.]]),)
grad input:  (tensor([1., 0., 3., 0.]), tensor([[22., 26., 30.]]), tensor([[1., 0., 3
        [1., 0., 3., 0.],
        [1., 0., 3., 0.]]))
```

读者可以自己用笔算一遍，验证正确性。需要注意的是，对线性模块，其 `grad_input` 是一个三元组，排列顺序分别为：对 bias 的导数，对输入 x 的导数，对权重 W 的导数。

注意事项

`register_backward_hook` 只能操作简单模块，而不能操作包含多个子模块的复杂模块。 如果对复杂模块用了 backward hook，那么我们只能得到该模块最后一次简单操作的梯度信息。对于上面的代码稍作修改，不再遍历各个子模块，而是把 model 整体绑在一个 `hook_fn_backward` 上：

```
model = Model()
model.register_backward_hook(hook_fn_backward)
```

输出结果如下：

```
Model(
  (fc1): Linear(in_features=3, out_features=4, bias=True)
  (relu1): ReLU()
  (fc2): Linear(in_features=4, out_features=1, bias=True)
)
grad_output (tensor([[1.]]),)
grad_input (tensor([1.]), tensor([[1., 2., 3., 4.]]), tensor([[ 7.],
        [ 0.],
        [27.],
        [ 0.]]))
==========Saved inputs and outputs==========
grad output:  (tensor([[1.]]),)
grad input:  (tensor([1.]), tensor([[1., 2., 3., 4.]]), tensor([[ 7.],
        [ 0.],
        [27.],
        [ 0.]]))
```

我们发现，程序只输出了 fc2 的梯度信息。

除此之外，有人还总结（吐槽）了 backward hook 在全连接层和卷积层表现不一致的地方（Feedback about PyTorch register_backward_hook · Issue #12331 · pytorch/pytorch）

1. 形状
    1. 在卷积层中，weight 的梯度和 weight 的形状相同
    2. 在全连接层中，weight 的梯度的形状是 weight 形状的转秩（观察上文中代码的输出可以验证）
2. `grad_input` tuple 中各梯度的顺序
    1. 在卷积层中，bias 的梯度位于tuple 的末尾： `grad_input` = (对feature的导数，对权重 W 的导数，对 bias 的导数)
    2. 在全连接层中，bias 的梯度位于 tuple 的开头： `grad_input` =(对 bias 的导数，对 feature 的

导数，对 W 的导数)

3. 当 batchsize>1时，对 bias 的梯度处理不同
    1. 在卷积层，对 bias 的梯度为整个 batch 的数据在 bias 上的梯度之和：`grad_input` = (对feature的导数，对权重 W 的导数，对 bias 的导数)
    2. 在全连接层，对 bias 的梯度是分开的，bach 中每条数据，对应一个 bias 的梯度：`grad_input` = ((data1 对 bias 的导数，data2 对 bias 的导数 ...)，对 feature 的导数，对 W 的导数)

Guided Backpropagation

通过上文的介绍，我们已经掌握了PyTorch 中各种 hook 的使用方法。接下来，我们将用这个技术写一小段代码（从 kaggle 上扒的，稍作了一点修改），来可视化预训练的神经网络。

Guided Backpropagation 算法来自 ICLR 2015 的文章：Striving for Simplicity: The All Convolutional Net。其基本原理和大多数可视化算法类似：通过反向传播，计算需要可视化的输出或者feature map 对网络输入的梯度，归一化该梯度，作为图片显示出来。梯度大的部分，反映了输入图片该区域对目标输出的影响力较大，反之影响力小。借此，我们可以了解到神经网络作出的判断，到底是受图片中哪些区域所影响，或者目标 feature map 提取的是输入图片中哪些区域的特征。Guided Backpropagation 对反向传播过程中 ReLU 的部分做了微小的调整。

我们先回忆传统的反向传播算法：假如第 *l* 层为 ReLU，那么前向传播公式为：

$$f_i^{l+1} = relu(f_i^l) = max(f_i^l, 0) \tag{1}$$

当输入 ReLU 的值大于0时，其输出对输入的导数为 1，当输入 ReLU 的值小于等于 0 时，其输出对输入的导数为 0。根据链式法则，其反向传播公式如下：

$$R_i^l = (f_i^l > 0) \cdot R_i^{l+1} = (f_i^l > 0) \cdot \frac{\partial f_{out}}{\partial f_i^{l+1}} \tag{2}$$

即 ReLU 层反向传播时，只有输入大于 0 的位置，才会有梯度传回来，输入小于等于 0 的位置不再有梯度反传。

Guided Backpropagation 的创新在于，它反向传播时，只传播梯度大于零的部分，抛弃梯度小于零的部分。这很好理解，因为我们希望的是，找到输入图片中对目标输出有正面作用的区域，而不是对目标输出有负面作用的区域。其公式如下：

$$R_i^l = (f_i^l > 0) \cdot (R_i^{l+1} > 0) \cdot R_i^{l+1} \tag{3}$$

下面是代码部分：

```
import torch
from torch import nn
```

```
class Guided_backprop():
    def __init__(self, model):
        self.model = model
        self.image_reconstruction = None
        self.activation_maps = []
        self.model.eval()
        self.register_hooks()

    def register_hooks(self):
        def first_layer_hook_fn(module, grad_in, grad_out):
            # 在全局变量中保存输入图片的梯度，该梯度由第一层卷积层
            # 反向传播得到，因此该函数需绑定第一个 Conv2d Layer
            self.image_reconstruction = grad_in[0]

        def forward_hook_fn(module, input, output):
            # 在全局变量中保存 ReLU 层的前向传播输出
            # 用于将来做 guided backpropagation
            self.activation_maps.append(output)

        def backward_hook_fn(module, grad_in, grad_out):
            # ReLU 层反向传播时，用其正向传播的输出作为 guide
            # 反向传播和正向传播相反，先从后面传起
            grad = self.activation_maps.pop()
            # ReLU 正向传播的输出要么大于0，要么等于0，
            # 大于 0 的部分，梯度为1，
            # 等于0的部分，梯度还是 0
            grad[grad > 0] = 1

            # grad_out[0] 表示 feature 的梯度，只保留大于 0 的部分
            positive_grad_out = torch.clamp(grad_out[0], min=0.0)
            # 创建新的输入端梯度
            new_grad_in = positive_grad_out * grad

            # ReLU 不含 parameter，输入端梯度是一个只有一个元素的 tuple
            return (new_grad_in,)


        # 获取 module，这里只针对 alexnet，如果是别的，则需修改
        modules = list(self.model.features.named_children())

        # 遍历所有 module，对 ReLU 注册 forward hook 和 backward hook
        for name, module in modules:
            if isinstance(module, nn.ReLU):
                module.register_forward_hook(forward_hook_fn)
                module.register_backward_hook(backward_hook_fn)
```

```
        # 对第1层卷积层注册 hook
        first_layer = modules[0][1]
        first_layer.register_backward_hook(first_layer_hook_fn)

    def visualize(self, input_image, target_class):
        # 获取输出，之前注册的 forward hook 开始起作用
        model_output = self.model(input_image)
        self.model.zero_grad()
        pred_class = model_output.argmax().item()

        # 生成目标类 one-hot 向量，作为反向传播的起点
        grad_target_map = torch.zeros(model_output.shape,
                                      dtype=torch.float)
        if target_class is not None:
            grad_target_map[0][target_class] = 1
        else:
            grad_target_map[0][pred_class] = 1

        # 反向传播，之前注册的 backward hook 开始起作用
        model_output.backward(grad_target_map)
        # 得到 target class 对输入图片的梯度，转换成图片格式
        result = self.image_reconstruction.data[0].permute(1,2,0)
        return result.numpy()

def normalize(I):
    # 归一化梯度map，先归一化到 mean=0 std=1
    norm = (I-I.mean())/I.std()
    # 把 std 重置为 0.1，让梯度map中的数值尽可能接近 0
    norm = norm * 0.1
    # 均值加 0.5，保证大部分的梯度值为正
    norm = norm + 0.5
    # 把 0，1 以外的梯度值分别设置为 0 和 1
    norm = norm.clip(0, 1)
    return norm




if __name__=='__main__':
    from torchvision import models, transforms
    from PIL import Image
    import matplotlib.pyplot as plt

    image_path = './cat.png'
    I = Image.open(image_path).convert('RGB')
    means = [0.485, 0.456, 0.406]
    stds = [0.229, 0.224, 0.225]
    size = 224
```

```
    transform = transforms.Compose([
        transforms.Resize(size),
        transforms.CenterCrop(size),
        transforms.ToTensor(),
        transforms.Normalize(means, stds)
    ])

    tensor = transform(I).unsqueeze(0).requires_grad_()

    model = models.alexnet(pretrained=True)

    guided_bp = Guided_backprop(model)
    result = guided_bp.visualize(tensor, None)

    result = normalize(result)
    plt.imshow(result)
    plt.show()


    print('END')
```

程序中用到的图为：

运行结果为：

从图中可以看出，小猫的脑袋部分，尤其是眼睛、鼻子、嘴巴和耳朵的梯度很大，而背景等部分，梯度很小，正是这些部分让神经网络认出该图片为小猫的。

Guided Backpropagation 的缺点是对 target class 不敏感，设置不同的 target class，最终可能得到的 gradient map 差别不大。基于此，有 Grad-CAM (Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization) 等更高级的可视化方法，限于篇幅不做介绍。

总结

本文介绍了 PyTorch 中的 hook 技术， 从针对 Tensor 的 hook，到针对 Module 的 hook，最终详细解读了利用 hook 技术可视化神经网络的代码。感谢大家的阅读，还望各位不吝批评指教。

编辑于 2019-08-04

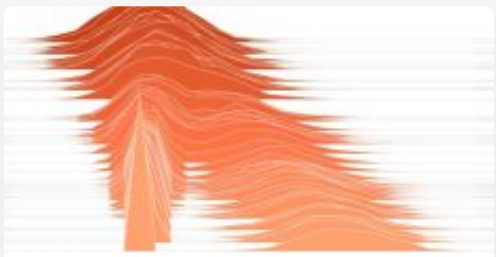
深度学习（Deep Learning） 计算机视觉

文章被以下专栏收录

SIGAI
专注于AI技术研究与机器学习框架研发，让AI所见即所得

已关注

推荐阅读



Pytorch训练可视化 (TensorboardX)

Teddy... 发表于那些搬砖的...

这个故事是
作的时候

阿树，让你
西，你给我

陈总的办公

15 条评论 切换为时间排序

写下你的评论...