第三章 PyTorch基礎：Autograde

出處: https://github.com/chenyuntc/pytorch-book

3.2 Autograd

- 用Tensor訓練網路很方便，但從上一小節最後的線性回歸例子來看，反向傳播過程需要手動實現。這對於像線性回歸等較為簡單的模型來說，還可以應付，但實際使用中經常出現非常複雜的網路結構，此時如果手動實現反向傳播，不僅費時費力，而且容易出錯，難以檢查。torch.autograd就是為方便用戶使用，而專門開發的一套自動求導引擎，它能夠根據輸入和前向傳播過程自動構建計算圖，並執行反向傳播。
- 計算圖(Computation Graph)是現代深度學習框架如PyTorch和TensorFlow等的核心，其為高效自動求導演算法——反向傳播(Back Propogation)提供了理論支援，瞭解計算圖在實際寫程式過程中會有極大的幫助。本節將涉及一些基礎的計算圖知識，但並不要求讀者事先對此有深入的瞭解。關於計算圖的基礎知識推薦閱讀Christopher Olah的文章
- Https://github.com/chenyuntc/pytorch-book/blob/master/chapter3-Tensor%E5%92%8Cautograd/Autograd.ipynb

3.2.1 Requires_grad

- PyTorch在autograd模組中實現了計算圖的相關功能，autograd中的核心資料結構是Variable。從v0.4版本起，Variable和Tensor合併。我們可以認為需要求導(requires_grad)的tensor即Variable. autograd記錄對tensor的操作記錄用來構建計算圖。
- Variable提供了大部分tensor支援的函數，但其不支援部分inplace函數，因這些函數會修改tensor自身，而在反向傳播中，variable需要緩存原來的tensor來計算反向傳播梯度。如果想要計算各個Variable的梯度，只需調用根節點variable的backward方法，autograd會自動沿著計算圖反向傳播，計算每一個葉子節點的梯度。

Variable.backward (gradient=None,retain_graph=None,create_graph=None)

主要有如下參數：

1. Grad_variables：形狀與variable一致，對於y.backward()，grad_variables相當於鏈式法則 中的 。grad_variables也可以是tensor或序列。
2. Retain_graph：反向傳播需要緩存一些中間結果，反向傳播之後，這些緩存就被清空，可通過指定這個參數不清空緩存，用來多次反向傳播。
3. Create_graph：對反向傳播過程再次構建計算圖，可通過backward of backward實現求高階導數。

http://colah.github.io/posts/2015-08-Backprop/

上述描述可能比較抽象，如果沒有看懂，不用著急，會在本節後半部分詳細介紹，下面先看幾個例子。

```
In [1]:
from __future__ import print_function
import torch as t
In [2]:
#在創建 tensor 的時候指定 requires_grad
a = t.randn(3,4, requires_grad=True)
# 或者
a = t.randn(3,4).requires_grad_()
# 或者
a = t.randn(3,4)
a.requires_grad=True
a
Out[2]:
tensor([[ 0.9289,  1.4844, -1.1878, -0.7120],
        [-0.3095,  1.2505,  0.9202,  0.9983],
        [-1.3756,  1.0462,  0.9083, -1.0232]])
In [3]:
b = t.zeros(3,4).requires_grad_()
b
Out[3]:
tensor([[ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.]])
In [4]:
# 也可寫成 c = a + b
c = a.add(b)
c
Out[4]:
tensor([[ 0.9289,  1.4844, -1.1878, -0.7120],
        [-0.3095,  1.2505,  0.9202,  0.9983],
        [-1.3756,  1.0462,  0.9083, -1.0232]])
In [5]:
d = c.sum()
d.backward() # 反向傳播
In [6]:
d # d 還是一個 requires_grad=True 的 tensor,對它的操作需要慎重
d.requires_grad
Out[6]:
True
```

```
In [7]:
a.grad
Out[7]:
tensor([[ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.]])
In [8]:
# 此處雖然沒有指定 c 需要求導，但 c 依賴於 a，而 a 需要求導，
# 因此 c 的 requires_grad 屬性會自動設為 True
a.requires_grad, b.requires_grad, c.requires_grad
Out[8]:
(True, True, True)
In [9]:
# 由用户創建的 variable 屬於葉子節點，對應的 grad_fn 是 None
a.is_leaf, b.is_leaf, c.is_leaf
Out[9]:
(True, True, False)
In [10]:
# c.grad 是 None, 因 c 不是葉子節點，它的梯度是用來計算 a 的梯度
# 所以雖然 c.requires_grad = True,但其梯度計算完之後即被釋放
c.grad is None
Out[10]:
True
```

計算下面這個函數的導函數：

來看看autograd的計算結果與手動求導計算結果的誤差。

```
In [11]:
def f(x):
    '''計算y'''
    y = x**2 * t.exp(x)
    return y
def gradf(x):
    '''手動求導函數'''
    dx = 2*x*t.exp(x) + x**2*t.exp(x)
    return dx
In [12]:
x = t.randn(3,4, requires_grad = True)
y = f(x)
y
Out[12]:
tensor([[ 0.0928,  0.1978,  0.6754,  0.8037],
        [ 0.9882,  0.3546,  0.2380,  0.0002],
        [ 0.2863,  0.0448,  0.1516,  2.9122]])
```

```
In [13]:
y.backward(t.ones(y.size())) # gradient 形狀與 y 一致
x.grad
Out[13]:
tensor([[-0.4146, -0.4610,  2.9016,  3.2831],
        [ 3.8102,  1.8614, -0.4536, -0.0244],
        [-0.4321,  0.5110, -0.4549,  8.6048]])
In [14]:
# autograd 的計算結果與利用公式手動計算的結果一致
gradf(x)
Out[14]:
tensor([[-0.4146, -0.4610,  2.9016,  3.2831],
        [ 3.8102,  1.8614, -0.4536, -0.0244],
        [-0.4321,  0.5110, -0.4549,  8.6048]])
```

3.2.2 計算圖

PyTorch中autograd的底層採用了計算圖,計算圖是一種特殊的有向無環圖(DAG),用於紀錄算子與變量之間的關係,一般用矩形表示算子,橢圓形表示變量.如表達式z=$wx+b$可以分解為$y = wx$和z= $y + b$,其計算圖如圖3-3所示,圖中MUL,ADD都是算子,w,x,b即變量.

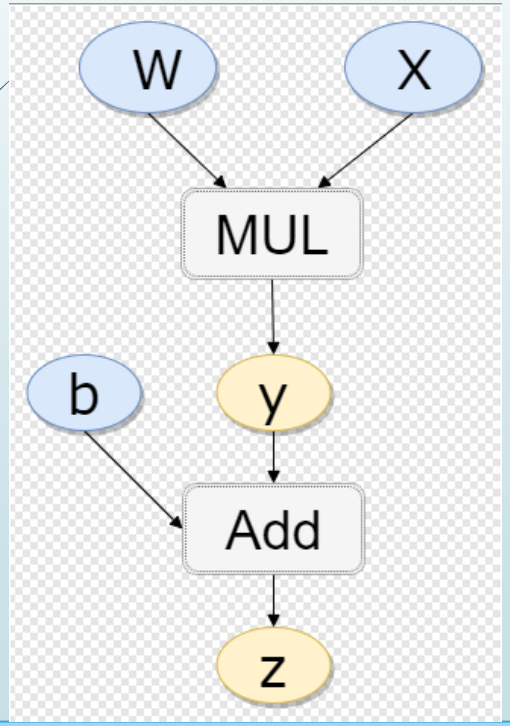


圖3-3 computation graph

如3-3有向無環圖中，x和b是葉子節點（leaf node），這些節點通常由用戶自己創建，不依賴於其他變數。z稱為根節點，是計算圖的最終目標。利用鏈式法則很容易求得各個葉子節點的梯度。

$$\partial\frac{z}{\partial}b=1,\partial\frac{z}{\partial}y=1\partial\frac{y}{\partial}w=x,\partial\frac{y}{\partial}x=w\partial\frac{z}{\partial}x=\partial\frac{z}{\partial}y\partial\frac{y}{\partial}x=1*w\partial\frac{z}{\partial}w=\partial\frac{z}{\partial}y\partial\frac{y}{\partial}w=1*x$$

有了計算圖，上述鏈式求導即可利用計算圖的反向傳播自動完成，其過程如圖3-4所示。

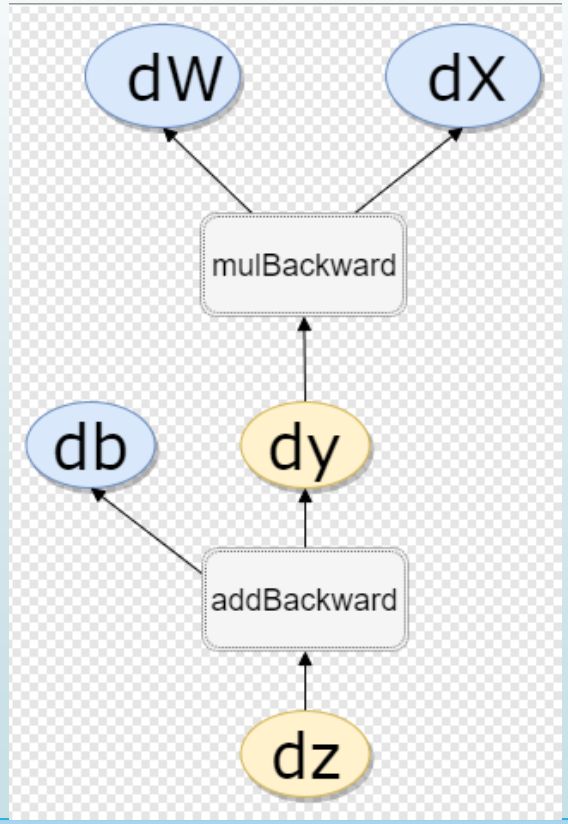


圖3-4 計算圖的反向傳播

- 在PyTorch實現中，autograd會隨著使用者的操作，記錄生成當前variable的所有操作，並由此建立一個有向無環圖。用戶每進行一個操作，相應的計算圖就會發生改變。更底層的實現中，圖中記錄了操作Function，每一個變數在圖中的位置可通過其grad_fn屬性在圖中的位置推測得到。在反向傳播過程中，autograd沿著這個圖從當前變數（根節點z）溯源，可以利用鏈式求導法則計算所有葉子節點的梯度。每一個前向傳播操作的函數都有與之對應的反向傳播函數用來計算輸入的各個variable的梯度，這些函數的函數名通常以Backward結尾。
- 下面結合代碼學習autograd的實現細節。

```
In [15]:
x = t.ones(1)
b = t.rand(1, requires_grad = True)
w = t.rand(1, requires_grad = True)
y = w * x # 等價於 y=w.mul(x)
z = y + b # 等價於 z=y.add(b)
In [16]:
x.requires_grad, b.requires_grad, w.requires_grad
Out[16]:
(False, True, True)
In [17]:
# 雖然未指定 y.requires_grad 為 True，但由於 y 依賴於需要求導的 w
# 故而 y.requires_grad 為 True
y.requires_grad
Out[17]:
True
In [18]:
x.is_leaf, w.is_leaf, b.is_leaf
Out[18]:
(True, True, True)
In [19]:
y.is_leaf, z.is_leaf
Out[19]:
(False, False)
In [20]:
# grad_fn 可以查看這個 variable 的反向傳播函數，
# z 是 add 函數的輸出，所以它的反向傳播函數是 AddBackward
z.grad_fn
Out[20]:
<AddBackward1 at 0x7f60b09c2630>
In [21]:
# next_functions 保存 grad_fn 的輸入，是一個 tuple，tuple 的元素也是 Function
# 第一個是 y，它是乘法(mul)的輸出，所以對應的反向傳播函數 y.grad_fn 是 MulBackward
# 第二個是 b，它是葉子節點，由用戶創建，grad_fn 為 None，但是有
z.grad_fn.next_functions
Out[21]:
((<MulBackward1 at 0x7f60b09c2278>, 0),
 (<AccumulateGrad at 0x7f60b09c2198>, 0))
In [22]:
# variable 的 grad_fn 對應著和圖中的 function 相對應
z.grad_fn.next_functions[0][0] == y.grad_fn
Out[22]:
True
In [23]:
# 第一個是 w，葉子節點，需要求導，梯度是累加的
# 第二個是 x，葉子節點，不需要求導，所以為 None
y.grad_fn.next_functions
Out[23]:
((<AccumulateGrad at 0x7f60b09c2898>, 0), (None, 0))
In [24]:
# 葉子節點的 grad_fn 是 None
w.grad_fn,x.grad_fn
Out[24]:
(None, None)
```

計算w的梯度的時候，需要用到x的數值($\partial \frac{y}{\partial} w = x$)這些數值在前向過程中會保存成buffer，在計算完梯度之後會自動清空。為了能夠多次反向傳播需要指定retain_graph來保留這些buffer。

```
In [25]:
# 使用 retain_graph 來保存 buffer
z.backward(retain_graph=True)
w.grad
Out[25]:
tensor([ 1.])
In [26]:
# 多次反向傳播，梯度累加，這也就是 w 中 AccumulateGrad 標識的含義
z.backward()
w.grad
Out[26]:
tensor([ 2.])
```

PyTorch使用的是動態圖，它的計算圖在每次前向傳播時都是從頭開始構建，所以它能夠使用Python控制語句（如for、if等）根據需求創建計算圖。這點在自然語言處理領域中很有用，它意味著你不需要事先構建所有可能用到的圖的路徑，圖在運行時才構建。

```
In [27]:
def abs(x):
    if x.data[0]>0: return x
    else: return -x
x = t.ones(1,requires_grad=True)
y = abs(x)
y.backward()
x.grad
Out[27]:
tensor([ 1.])
In [28]:
x = -1*t.ones(1)
x = x.requires_grad_()
y = abs(x)
y.backward()
print(x.grad)
tensor([-1.])
In [29]:
def f(x):
    result = 1
    for ii in x:
        if ii.item()>0: result=ii*result
    return result
x = t.arange(-2,4,requires_grad=True)
y = f(x) # y = x[3]*x[4]*x[5]
y.backward()
x.grad
Out[29]:
tensor([ 0.,  0.,  0.,  6.,  3.,  2.])
```

變數的requires_grad屬性預設為False，如果某一個節點requires_grad被設置為True，那麼所有依賴它的節點requires_grad都是True。這其實很好理解，對於x → y → z，x.requires_grad =True，當需要計算$\partial \frac{z}{\partial} x$時，根據鏈式法則，$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$，自然也需要求$\frac{\partial z}{\partial y}$，所以y.requires_grad會被自動標為True.

有些時候我們可能不希望autograd對tensor求導。認為求導需要緩存許多中間結構，增加額外的記憶體/顯存開銷，那麼我們可以關閉自動求導。對於不需要反向傳播的情景（如inference，即測試推理時），關閉自動求導可實現一定程度的速度提升，並節省約一半顯存，因其不需要分配空間計算梯度。

```
In [30]:
x = t.ones(1, requires_grad=True)
w = t.rand(1, requires_grad=True)
y = x * w
# y 依賴於 w，而 w.requires_grad = True
x.requires_grad, w.requires_grad, y.requires_grad
Out[30]:
(True, True, True)
```

```
In [31]:
with t.no_grad():
    x = t.ones(1)
    w = t.rand(1, requires_grad = True)
    y = x * w
# y 依賴於 w和 x，雖然 w.requires_grad = True，但是 y 的 requires_grad 依舊為 False
x.requires_grad, w.requires_grad, y.requires_grad
Out[31]:
(False, True, False)
In [32]:
t.no_grad??
In [33]:
t.set_grad_enabled(False)
x = t.ones(1)
w = t.rand(1, requires_grad = True)
y = x * w
# y 依賴於 w和 x，雖然 w.requires_grad = True，但是 y 的 requires_grad 依舊為 False
x.requires_grad, w.requires_grad, y.requires_grad
Out[33]:
(False, True, False)
In [34]:
# 恢復預設配置
t.set_grad_enabled(True)
Out[34]:
<torch.autograd.grad_mode.set_grad_enabled at 0x7f60b01730b8>
In [ ]:
```

如果我們想要修改tensor的數值，但是又不希望被autograd記錄，那麼我麼可以對tensor.data進行操作

```
In [35]:
a = t.ones(3,4,requires_grad=True)
b = t.ones(3,4,requires_grad=True)
c = a * b

a.data # 還是一個 tensor
Out[35]:
tensor([[ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.]])
```

```
In [36]:
a.data.requires_grad # 但是已經是獨立於計算圖之外
Out[36]:
False
In [37]:
d = a.data.sigmoid_() # sigmoid_ 是個 inplace 操作，會修改 a 自身的值
d.requires_grad
Out[37]:
False
In [38]:
a
Out[38]:
tensor([[ 0.7311,  0.7311,  0.7311,  0.7311],
        [ 0.7311,  0.7311,  0.7311,  0.7311],
        [ 0.7311,  0.7311,  0.7311,  0.7311]])
```

如果我們希望對tensor，但是又不希望被記錄, 可以使用tensor.data 或者tensor.detach()

```
In [39]:
a.requires_grad
Out[39]:
True
In [40]:
# 近似於 tensor=a.data, 但是如果 tensor 被修改，backward 可能會報錯
tensor = a.detach()
tensor.requires_grad
Out[40]:
False
In [41]:
# 統計 tensor 的一些指標，不希望被記錄
mean = tensor.mean()
std = tensor.std()
maximum = tensor.max()
In [42]:
tensor[0]=1
# 下面會報錯：  RuntimeError: one of the variables needed for gradient
#              computation has been modified by an inplace operation
#  因為 c=a*b, b 的梯度取決於 a，現在修改了 tensor，其實也就是修改了 a，梯度不再準確
# c.sum().backward()
```

在反向傳播過程中非葉子節點的導數計算完之後即被清空。若想查看這些變數的梯度，有兩種方法：

1. 使用autograd.grad函數
2. 使用hook

Autograd.grad 和 hook 方法都是很強大的工具，更詳細的用法參考官方api文檔，這裡舉例說明基礎的使用。推薦使用hook方法，但是在實際使用中應儘量避免修改grad的值。

```
In [43]:
x = t.ones(3, requires_grad=True)
w = t.rand(3, requires_grad=True)
y = x * w
# y 依賴於 w，而 w.requires_grad = True
z = y.sum()
x.requires_grad, w.requires_grad, y.requires_grad
Out[43]:
(True, True, True)
In [44]:
# 非葉子節點 grad 計算完之後自動清空，y.grad 是 None
z.backward()
(x.grad, w.grad, y.grad)
Out[44]:
(tensor([ 0.2709,  0.0473,  0.5052]), tensor([ 1.,  1.,  1.]), None)
In [45]:
# 第一種方法：使用 grad 獲取中間變數的梯度
x = t.ones(3, requires_grad=True)
w = t.rand(3, requires_grad=True)
y = x * w
z = y.sum()
# z 對 y 的梯度，隱式調用 backward()
t.autograd.grad(z, y)
Out[45]:
(tensor([ 1.,  1.,  1.]),)
In [46]:
# 第二種方法：使用 hook
# hook 是一個函數，輸入是梯度，不應該有返回值
def variable_hook(grad):
    print('y 的梯度：',grad)

x = t.ones(3, requires_grad=True)
w = t.rand(3, requires_grad=True)
y = x * w
# 註冊 hook
hook_handle = y.register_hook(variable_hook)
z = y.sum()
z.backward()

# 除非你每次都要用 hook，否則用完之後記得移除 hook
hook_handle.remove()
y 的梯度： tensor([ 1.,  1.,  1.])
```

最後再來看看variable中grad屬性和backward函數grad_variables參數的含義，這裡直接下結論：

- Variable x的梯度是目標函數$f(x)$對x的梯度，$\frac{df(x)}{dx}=(\frac{df(x)}{dx_0},\frac{df(x)}{dx_1},\cdots,\frac{df(x)}{dx_N})$，形狀和x一致。
- 對於y.backward(grad_variables)中的grad_variables相當於鏈式求導法則中的$\frac{\partial z}{\partial x}=\frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$中的$\frac{\partial z}{\partial y}$
- Z目標函數，一般是一個標量，故而$\frac{\partial z}{\partial y}$的形狀與variable y的形狀一致。z.backward()在一定程度上等價於y.backward(grad_y)。z.backward()省略了grad_variables參數，是因為z是一個標量，而$\frac{\partial z}{\partial z}=1$

```
In [47]:
x = t.arange(0,3, requires_grad=True)
y = x**2 + x*2
z = y.sum()
z.backward() # 從z 開始反向傳播
x.grad
Out[47]:
tensor([ 2.,  4.,  6.])
In [48]:
x = t.arange(0,3, requires_grad=True)
y = x**2 + x*2
z = y.sum()
y_gradient = t.Tensor([1,1,1]) # dz/dy
y.backward(y_gradient) #從y 開始反向傳播
x.grad
Out[48]:
tensor([ 2.,  4.,  6.])
```

另外值得注意的是，只有對variable的操作才能使用autograd，如果對variable的data直接進行操作，將無法使用反向傳播。除了對參數初始化，一般我們不會修改variable.data的值。

- 在PyTorch中計算圖的特點可總結如下：

1. Autograd根據使用者對variable的操作構建其計算圖。對變數的操作抽象為Function。
2. 對於那些不是任何函數(Function)的輸出，由用戶創建的節點稱為葉子節點，葉子節點的grad_fn為None。葉子節點中需要求導的variable，具有AccumulateGrad標識，因其梯度是累加的。
3. Variable預設是不需要求導的，即requires_grad屬性預設為False，如果某一個節點requires_grad被設置為True，那麼所有依賴它的節點requires_grad都為True。
4. Variable的volatile屬性預設為False，如果某一個variable的volatile屬性被設為True，那麼所有依賴它的節點volatile屬性都為True。volatile屬性為True的節點不會求導，volatile的優先順序比requires_grad高。
5. 多次反向傳播時，梯度是累加的。反向傳播的中間緩存會被清空，為進行多次反向傳播需指定retain_graph=True來保存這些緩存。
6. 非葉子節點的梯度計算完之後即被清空，可以使用autograd.grad或hook技術獲取非葉子節點的值。
7. Variable的grad與data形狀一致，應避免直接修改variable.data，因為對data的直接操作無法利用autograd進行反向傳播
8. 反向傳播函數backward的參數grad_variables可以看成鏈式求導的中間結果，如果是標量，可以省略，默認為1
9. PyTorch採用動態圖設計，可以很方便地查看中間層的輸出，動態的設計計算圖結構。

- 這些知識不懂大多數情況下也不會影響對pytorch的使用，但是掌握這些知識有助於更好的理解pytorch，並有效的避開很多

3.2.3 擴展Autograd

目前絕大多數函數都可以使用autograd實現反向求導，但如果需要自己寫一個複雜的函數，不支援自動反向求導怎麼辦? 寫一個Function，實現它的前向傳播和反向傳播代碼，Function對應於計算圖中的矩形， 它接收參數，計算並返回結果。下面給出一個例子。

```
class Mul(Function):

    @staticmethod
    def forward(ctx, w, x, b, x_requires_grad = True):
        ctx.x_requires_grad = x_requires_grad
        ctx.save_for_backward(w,x)
        output = w * x + b
        return output

    @staticmethod
    def backward(ctx, grad_output):
        w,x = ctx.saved_tensors
        grad_w = grad_output * x
        if ctx.x_requires_grad:
            grad_x = grad_output * w
        else:
            grad_x = None
        grad_b = grad_output * 1
        return grad_w, grad_x, grad_b, None
```

分析如下：

1. 自訂的Function需要繼承autograd.Function，沒有構造函數__init__，forward和backward函數都是靜態方法
2. Backward函數的輸出和forward函數的輸入一一對應，backward函數的輸入和forward函數的輸出一一對應
3. Backward函數的grad_output參數即t.autograd.backward中的grad_variables
4. 如果某一個輸入不需要求導，直接返回None，如forward中的輸入參數x_requires_grad顯然無法對它求導，直接返回None即可
5. 反向傳播可能需要利用前向傳播的某些中間結果，需要進行保存，否則前向傳播結束後這些物件即被釋放

Function的使用 利用Function.apply(variable)

```
In [49]:
from torch.autograd import Function
class MultiplyAdd(Function):

    @staticmethod
    def forward(ctx, w, x, b):
        ctx.save_for_backward(w,x)
        output = w * x + b
        return output

    @staticmethod
    def backward(ctx, grad_output):
        w,x = ctx.saved_tensors
        grad_w = grad_output * x
        grad_x = grad_output * w
        grad_b = grad_output * 1
        return grad_w, grad_x, grad_b
In [50]:
x = t.ones(1)
w = t.rand(1, requires_grad = True)
b = t.rand(1, requires_grad = True)
# 開始前向傳播
z=MultiplyAdd.apply(w, x, b)
# 開始反向傳播
z.backward()

# x不需要求導，中間過程還是會計算它的導數，但隨後被清空
x.grad, w.grad, b.grad
Out[50]:
(None, tensor([ 1.]), tensor([ 1.]))
```

```
In [51]:
x = t.ones(1)
w = t.rand(1, requires_grad = True)
b = t.rand(1, requires_grad = True)
#print('開始前向傳播')
z=MultiplyAdd.apply(w,x,b)
#print('開始反向傳播')

# 調用 MultiplyAdd.backward
# 輸出 grad_w, grad_x, grad_b
z.grad_fn.apply(t.ones(1))
Out[51]:
(tensor([ 1.]), tensor([ 0.1563]), tensor([ 1.]))
```

之所以forward函數的輸入是tensor，而backward函數的輸入是variable，是為了實現高階求導。backward函數的輸入輸出雖然是variable，但在實際使用時autograd.Function會將輸入variable提取為tensor，並將計算結果的tensor封裝成variable返回。在backward函數中，之所以也要對variable進行操作，是為了能夠計算梯度的梯度（backward of backward）。

下面舉例說明，有關torch.autograd.grad的更詳細使用請參照文檔。

```
In [52]:
x = t.tensor([5], requires_grad=True)
y = x ** 2
grad_x = t.autograd.grad(y, x, create_graph=True)
grad_x # dy/dx = 2 * x
Out[52]:
(tensor([ 10]),)
In [53]:
grad_grad_x = t.autograd.grad(grad_x[0],x)
grad_grad_x # 二階導數 d(2x)/dx = 2
Out[53]:
(tensor([ 2]),)
```

- 這種設計雖然能讓autograd具有高階求導功能，但其也限制了Tensor的使用，因autograd中反向傳播的函數只能利用當前已經有的Variable操作。這個設計是在0.2版本新加入的，為了更好的靈活性，也為了相容舊版本的代碼，PyTorch還提供了另外一種擴展autograd的方法。PyTorch提供了一個裝飾器@once_differentiable，能夠在backward函數中自動將輸入的variable提取成tensor，把計算結果的tensor自動封裝成variable。有了這個特性我們就能夠很方便的使用numpy/scipy中的函數，操作不再局限於variable所支持的操作。但是這種做法正如名字中所暗示的那樣只能求導一次，它打斷了反向傳播圖，不再支持高階求導。
- 上面所描述的都是新式Function，還有個legacy Function，可以帶有__init__方法，forward和backwad函數也不需要聲明為@staticmethod，但隨著版本更迭，此類Function將越來越少遇到，在此不做更多介紹。
- 此外在實現了自己的Function之後，還可以使用gradcheck函數來檢測實現是否正確。gradcheck通過數值逼近來計算梯度，可能具有一定的誤差，通過控制eps的大小可以控制容忍的誤差

下面舉例說明如何利用Function實現sigmoid Function。

```
In [54]:
class Sigmoid(Function):

    @staticmethod
    def forward(ctx, x):
        output = 1 / (1 + t.exp(-x))
        ctx.save_for_backward(output)
        return output

    @staticmethod
    def backward(ctx, grad_output):
        output,  = ctx.saved_tensors
        grad_x = output * (1 - output) * grad_output
        return grad_x

In [55]:
# 採用數值逼近方式檢驗計算梯度的公式對不對
test_input = t.randn(3,4, requires_grad=True)
t.autograd.gradcheck(Sigmoid.apply, (test_input,), eps=1e-3)
Out[55]:
True
```

```
In [56]:
def f_sigmoid(x):
    y = Sigmoid.apply(x)
    y.backward(t.ones(x.size()))

def f_naive(x):
    y =  1/(1 + t.exp(-x))
    y.backward(t.ones(x.size()))

def f_th(x):
    y = t.sigmoid(x)
    y.backward(t.ones(x.size()))

x=t.randn(100, 100, requires_grad=True)
%timeit -n 100 f_sigmoid(x)
%timeit -n 100 f_naive(x)
%timeit -n 100 f_th(x)
216 µs ± 42.5 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
285 µs ± 42.7 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
113 µs ± 46 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

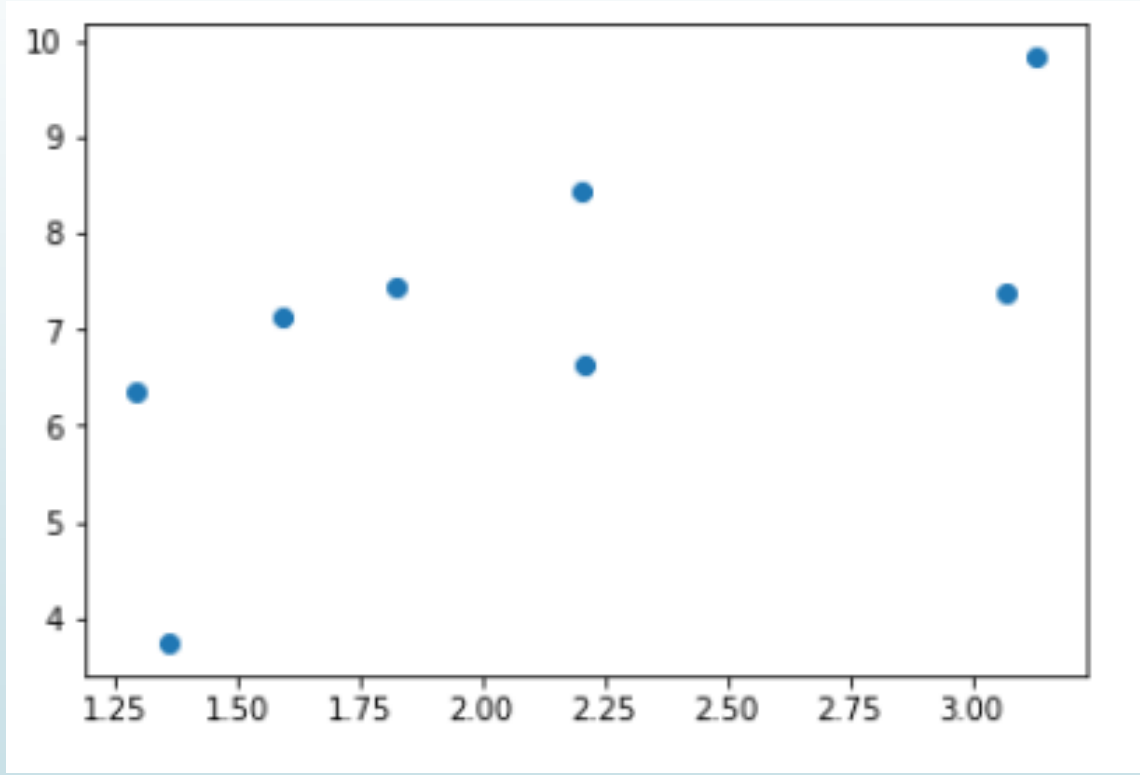
顯然f_sigmoid要比單純利用autograd加減和乘方操作實現的函數快不少，因為f_sigmoid的backward優化了反向傳播的過程。另外可以看出系統實現的built-in介面(t.sigmoid)更快。

3.2.4 小試牛刀: 用Variable實現線性回歸

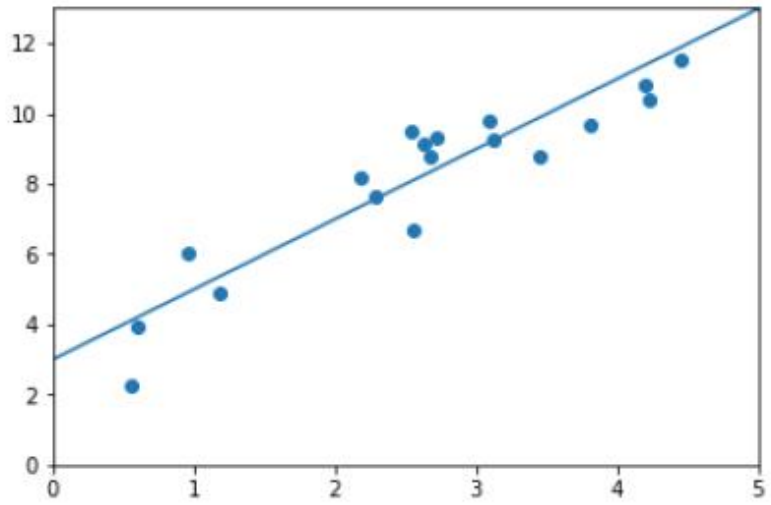
```
In [69]:
import torch as t
%matplotlib inline
from matplotlib import pyplot as plt
from IPython import display
import numpy as np
```

```
In [70]:
# 設置亂數種子，為了在不同人電腦上運行時下面的輸出一致
t.manual_seed(1000)

def get_fake_data(batch_size=8):
    ''' 產生亂數據：y = x*2 + 3，加上了一些雜訊'''
    x = t.rand(batch_size,1) * 5
    y = x * 2 + 3 + t.randn(batch_size, 1)
    return x, y
```

```
In [71]:
# 來看看產生 x-y 分佈是什麼樣的
x, y = get_fake_data()
plt.scatter(x.squeeze().numpy(), y.squeeze().numpy())
Out[71]:
<matplotlib.collections.PathCollection at 0x7f6099fa5668>
```

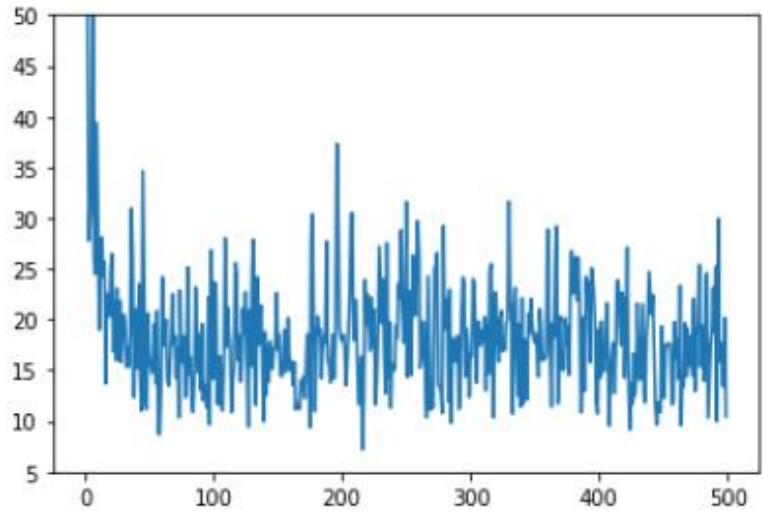


```
In [76]:
# 隨機初始化參數
w = t.rand(1,1, requires_grad=True)
b = t.zeros(1,1, requires_grad=True)
losses = np.zeros(500)

lr =0.005 # 學習率
for ii in range(500):
    x, y = get_fake_data(batch_size=32)

    # forward：計算 loss
    y_pred = x.mm(w) + b.expand_as(y)
    loss = 0.5 * (y_pred - y) ** 2
    loss = loss.sum()
    losses[ii] = loss.item()

    # backward：手動計算梯度
    loss.backward()

    # 更新參數
    w.data.sub_(lr * w.grad.data)
    b.data.sub_(lr * b.grad.data)

    # 梯度清零
    w.grad.data.zero_()
    b.grad.data.zero_()

    if ii%50 ==0:
        # 畫圖
        display.clear_output(wait=True)
        x = t.arange(0, 6).view(-1, 1)
        y = x.mm(w.data) + b.data.expand_as(x)
        plt.plot(x.numpy(), y.numpy()) # predicted

        x2, y2 = get_fake_data(batch_size=20)
        plt.scatter(x2.numpy(), y2.numpy()) # true data

        plt.xlim(0,5)
        plt.ylim(0,13)
        plt.show()
        plt.pause(0.5)

print(w.item(), b.item())
```




```
2.023955821990967 3.0775527954101562
```

```
In [80]:
plt.plot(losses)
plt.ylim(5,50)
Out[80]:
(5, 50)
```



用autograd實現的線性回歸最大的不同點就在於autograd不需要計算反向傳播，可以自動計算微分。這點不單是在深度學習，在許多機器學習的問題中都很有用。另外需要注意的是在每次反向傳播之前要記得先把梯度清零。

本章主要介紹了PyTorch中兩個基礎底層的資料結構：Tensor和autograd中的Variable。

- Tensor是一個類似Numpy陣列的高效多維數值運算資料結構，有著和Numpy相類似的介面，並提供簡單易用的GPU加速。
- Variable是autograd封裝了Tensor並提供自動求導技術的，具有和Tensor幾乎一樣的介面。
- Autograd是PyTorch的自動微分引擎，採用動態計算圖技術，能夠快速高效的計算導數。