



第四章

PyTorch神經網路工具箱 NN

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

上一章中提到，使用autograd可實現深度學習模型，但其抽象程度較低，如果用其來實現深度學習模型，則需要編寫的代碼量極大。在這種情況下，torch.nn應運而生，其是專門為深度學習而設計的模組。torch.nn的核心資料結構是Module，它是一個抽象概念，既可以表示神經網路中的某個層（layer），也可以表示一個包含很多層的神經網路。在實際使用中，最常見的做法是繼承nn.Module，撰寫自己的網路/層。下面先來看看如何用nn.Module實現自己的全連接層。全連接層，又名仿射層，輸出y和輸入x滿足， $y = Wx + b$, W和b是可學習的參數。

可見，全連接層的實現非常簡單，其代碼量不超過10行，但需注意以下幾點：

1. 自訂層 Linear 必須繼承 `nn.Module`，並且在其構造函數中需調用 `nn.Module` 的構造函數，即 `super(Linear, self).__init__()` 或 `nn.Module.__init__(self)`，推薦使用第一種用法，儘管第二種寫法更直觀。
2. 在構造函數 `__init__` 中必須自己定義可學習的參數，並封裝成 `Parameter`，如在本例中我們把 `w` 和 `b` 封裝成 `parameter`。`parameter` 是一種特殊的 `Tensor`，但其預設需要求導（`requires_grad = True`），感興趣的讀者可以通過 `nn.Parameter??`，查看 `Parameter` 類的原始程式碼。
3. `Forward` 函數實現前向傳播過程，其輸入可以是一個或多個 `tensor`。
4. 無需寫反向傳播函數，`nn.Module` 能夠利用 `autograd` 自動實現反向傳播，這點比 `Function` 簡單許多。
5. 使用時，直觀上可將 `layer` 看成數學概念中的函數，調用 `layer(input)` 即可得到 `input` 對應的結果。它等價於 `layers.__call__(input)`，在 `__call__` 函數中，主要調用的是 `layer.forward(x)`，另外還對鉤子做了一些處理。所以在實際使用中應儘量使用 `layer(x)` 而不是使用 `layer.forward(x)`，關於鉤子技術將在下文講解。
6. `Module` 中的可學習參數可以通過 `named_parameters()` 或者 `parameters()` 返回反覆運算器，前者會給每個 `parameter` 都附上名字，使其更具有辨識度。

```
In [6]:
import torch as t
from torch import nn

In [7]:
class Linear(nn.Module): # 繼承 nn.Module
    def __init__(self, in_features, out_features):
        super(Linear, self).__init__() # 等價於 nn.Module.__init__(self)
        self.w = nn.Parameter(t.randn(in_features, out_features))
        self.b = nn.Parameter(t.randn(out_features))

    def forward(self, x):
        x = x.mm(self.w) # x.@(self.w)
        return x + self.b.expand_as(x)

In [8]:
layer = Linear(4, 3)
input = t.randn(2, 4)
output = layer(input)
output
Out[8]:
tensor([[ -2.4421, -0.2355,  0.5636],
        [-1.4299, -1.0916, -1.9299]])

In [9]:
for name, parameter in layer.named_parameters():
    print(name, parameter) # w and b
w Parameter containing:
tensor([[ 0.0481, -0.5937, -1.0651],
        [ 0.5725,  0.1317, -0.0920],
        [ 0.7409,  0.3163, -1.3940],
        [ 0.1598, -0.3302,  0.3646]])
b Parameter containing:
tensor([-1.5881, -0.8368, -0.5220])
```

可見利用Module實現的全連接層，比利用Function實現的更為簡單，因其不再需要寫反向傳播函數。

Module能夠自動檢測到自己的Parameter，並將其作為學習參數。除了parameter之外，Module還包含子Module，主Module能夠遞迴查找子Module中的parameter。下面再來看看稍微複雜一點的網路，多層感知機。

多層感知機的網路結構如圖4-1所示，它由兩個全連接層組成，採用sigmoid函數作為啟動函數，圖中沒有畫出。

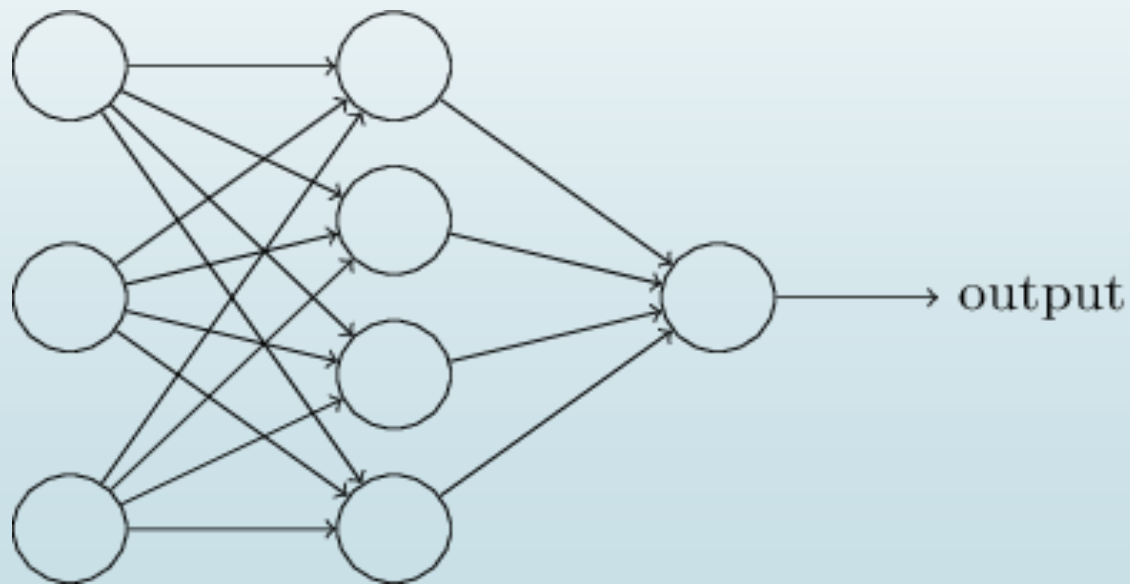


圖4-1 多層感知機的網路結構

```
In [10]:
class Perceptron(nn.Module):
    def __init__(self, in_features, hidden_features, out_features):
        nn.Module.__init__(self)
        self.layer1 = Linear(in_features, hidden_features) # 此處的Linear是前面自訂的全連接層
        self.layer2 = Linear(hidden_features, out_features)
    def forward(self, x):
        x = self.layer1(x)
        x = t.sigmoid(x)
        return self.layer2(x)

In [11]:
perceptron = Perceptron(3, 4, 1)
for name, param in perceptron.named_parameters():
    print(name, param.size())
layer1.w torch.Size([3, 4])
layer1.b torch.Size([4])
layer2.w torch.Size([4, 1])
layer2.b torch.Size([1])
```

可見，即使是稍複雜的多層感知機，其實現依舊很簡單。構造函數`__init__`中，可利用前面自訂的`Linear`層(module)，作為當前module對象的一個子module，它的可學習參數，也會成為當前module的可學習參數。

Module 中 Parameter 的命名規範：

1. 對於類似 `self.param_name = nn.Parameter(t.randn(3, 4))`，命名為 `param_name`
2. 對於子 Module 中的 parameter，會其名字之前加上當前 Module 的名字。如對於 `self.sub_module = SubModel()`，`SubModel` 中有個 parameter 的名字叫做 `param_name`，那麼二者拼接而成的 parameter name 就是 `sub_module.param_name`。

為方便使用者使用，PyTorch實現了神經網路中絕大多數的layer，這些layer都繼承於nn.Module，封裝了可學習參數parameter，並實現了forward函數，且很多都專門針對GPU運算進行了CuDNN優化，其速度和性能都十分優異。本書不準備對nn.Module中的所有層進行詳細介紹，具體內容讀者可參照官方文檔1或在IPython/Jupyter中使用nn.layer?來查看。閱讀文檔時應主要關注以下幾點：

1. 構造函數的參數，如nn.Linear(in_features, out_features, bias)，需關注這三個參數的作用。
2. 屬性、可學習參數和子module。如nn.Linear中有weight和bias兩個可學習參數，不包含子module。
3. 輸入輸出的形狀，如nn.linear的輸入形狀是(N, input_features)，輸出為(N, output_features)，N是batch_size。

這些自訂layer對輸入形狀都有假設：輸入的不是單個資料，而是一個batch。輸入只有一個資料，則必須調用tensor.unsqueeze(0) 或 tensor[None]將數據偽裝成batch_size=1的batch <http://pytorch.org/docs/nn.html>



4.1 常用神經網路層

4.1.1 圖像相關層

圖像相關層主要包括卷積層（Conv）、池化層（Pool）等，這些層在實際使用中可分為一維(1D)、二維(2D)、三維（3D），池化方式又分為平均池化（AvgPool）、最大值池化（MaxPool）、自我調整池化（AdaptiveAvgPool）等。而卷積層除了常用的前向卷積之外，還有逆卷積（TransposeConv）。下面舉例說明一些基礎的使用。

```
In [12]:
from PIL import Image
from torchvision.transforms import ToTensor, ToPILImage
to_tensor = ToTensor() # img -> tensor
to_pil = ToPILImage()
lena = Image.open('imgs/lena.png')
lena
Out[12]:
```



```
In [13]:
# 輸入是一個 batch，batch_size=1
input = to_tensor(lena).unsqueeze(0)

# 銳化卷積核
kernel = t.ones(3, 3)/-9.
kernel[1][1] = 1
conv = nn.Conv2d(1, 1, (3, 3), 1, bias=False)
conv.weight.data = kernel.view(1, 1, 3, 3)

out = conv(input)
to_pil(out.data.squeeze(0))
Out[13]:
```



除了上述的使用，圖像的卷積操作還有各種變體，具體可以參照此處動圖1介紹。

https://github.com/vdumoulin/conv_arithmetic/blob/master/README.md

池化層可以看作是一種特殊的卷積層，用來下採樣。
但池化層沒有可學習參數，其weight是固定的。

```
In [14]:  
pool = nn.AvgPool2d(2, 2)  
list(pool.parameters())  
Out[14]:  
[]  
In [15]:  
out = pool(input)  
to_pil(out.data.squeeze(0))  
Out[15]:
```





除了卷積層和池化層，深度學習中還將常用到以下幾個層：

1. Linear：全連接層。
2. BatchNorm：批規範化層，分為1D、2D和3D。除了標準的BatchNorm之外，還有在風格遷移中常用到的InstanceNorm層。
3. Dropout：dropout層，用來防止過擬合，同樣分為1D、2D和3D。

下面通過例子來說明它們的使用。

```
In [16]:
# 輸入 batch_size=2, 維度 3
input = t.randn(2, 3)
linear = nn.Linear(3, 4)
h = linear(input)
h
Out[16]:
tensor([[ 0.6993, -1.1460,  0.5710, -0.2496],
        [-0.1921,  0.8154, -0.3038,  0.1873]])

In [17]:
# 4 channel, 初始化標準差為 4, 均值為 0
bn = nn.BatchNorm1d(4)
bn.weight.data = t.ones(4) * 4
bn.bias.data = t.zeros(4)

bn_out = bn(h)
# 注意輸出的均值和方差
# 方差是標準差的平方, 計算無偏方差分母會減 1
# 使用 unbiased=False 分母不減 1
bn_out.mean(0), bn_out.var(0, unbiased=False)
Out[17]:
(tensor(1.00000e-07 *
      [ 1.1921,  0.0000,  0.0000,  0.0000]),
 tensor([ 15.9992, 15.9998, 15.9992, 15.9966]))

In [18]:
# 每個元素以 0.5 的概率捨棄
dropout = nn.Dropout(0.5)
o = dropout(bn_out)
o # 有一半左右的數變為 0
Out[18]:
tensor([[ 7.9998, -8.0000,  0.0000, -7.9992],
        [-0.0000,  8.0000, -7.9998,  7.9992]])
```

以上很多例子中都對module的屬性直接操作，其大多數是可學習參數，一般會隨著學習的進行而不斷改變。實際使用中除非需要使用特殊的初始化，應儘量不要直接修改這些參數。

4.1.2 啟動函數

PyTorch實現了常見的啟動函數，其具體的介面資訊可參見官方文檔¹，這些啟動函數可作為獨立的layer使用。這裡將介紹最常用的啟動函數ReLU，其數學運算式為：

$$\text{ReLU}(x) = \max(0, x)$$

<http://pytorch.org/docs/nn.html#non-linear-activations>

```
In [19]:
relu = nn.ReLU(inplace=True)
input = t.randn(2, 3)
print(input)
output = relu(input)
print(output) # 小於 0 的都被截斷為 0
# 等價於 input.clamp(min=0)
tensor([[ 1.2836,  2.0970, -0.0456],
        [ 1.5909, -1.3795,  0.5264]])
tensor([[ 1.2836,  2.0970,  0.0000],
        [ 1.5909,  0.0000,  0.5264]])
```

ReLU函數有個inplace參數，如果設為True，它會把輸出直接覆蓋到輸入中，這樣可以節省記憶體/顯存。之所以可以覆蓋是因為在計算ReLU的反向傳播時，只需根據輸出就能夠推算出反向傳播的梯度。但是只有少數的autograd操作支持inplace操作（如tensor.sigmoid_()），除非你明確地知道自己在做什麼，否則一般不要使用inplace操作。

- 在以上的例子中，基本上都是將每一層的輸出直接作為下一層的輸入，這種網路稱為前饋傳播網路（feedforward neural network）。
- 對於此類網路如果每次都寫複雜的forward函數會有些麻煩，在此就有兩種簡化方式，ModuleList和Sequential。其中Sequential是一個特殊的module，它包含幾個子Module，前向傳播時會將輸入一層接一層的傳遞下去。
- ModuleList也是一個特殊的module，可以包含幾個子module，可以像用list一樣使用它，但不能直接把輸入傳給ModuleList。
- 下面舉例說明

```

In [20]:
# Sequential 的三種寫法
net1 = nn.Sequential()
net1.add_module('conv', nn.Conv2d(3, 3, 3))
net1.add_module('batchnorm', nn.BatchNorm2d(3))

net1.add_module('activation_layer', nn.ReLU())

net2 = nn.Sequential(
    nn.Conv2d(3, 3, 3),
    nn.BatchNorm2d(3),
    nn.ReLU()
)

from collections import OrderedDict
net3= nn.Sequential(OrderedDict([
    ('conv1', nn.Conv2d(3, 3, 3)),
    ('bn1', nn.BatchNorm2d(3)),
    ('relu1', nn.ReLU())
]))

print('net1:', net1)
print('net2:', net2)
print('net3:', net3)
net1: Sequential(
  (conv): Conv2d(3, 3, kernel_size=(3, 3), stride=(1, 1))
  (batchnorm): BatchNorm2d(3, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (activation_layer): ReLU()
)
net2: Sequential(
  (0): Conv2d(3, 3, kernel_size=(3, 3), stride=(1, 1))
  (1): BatchNorm2d(3, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): ReLU()
)
net3: Sequential(
  (conv1): Conv2d(3, 3, kernel_size=(3, 3), stride=(1, 1))
  (bn1): BatchNorm2d(3, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu1): ReLU()
)

```

```

In [21]:
# 可根據名字或序號取出子 module
net1.conv, net2[0], net3.conv1
Out[21]:
(Conv2d(3, 3, kernel_size=(3, 3), stride=(1, 1)),
 Conv2d(3, 3, kernel_size=(3, 3), stride=(1, 1)),
 Conv2d(3, 3, kernel_size=(3, 3), stride=(1, 1)))
In [22]:
input = t.rand(1, 3, 4, 4)

output = net1(input)
output = net2(input)
output = net3(input)
output = net3.relu(net1.batchnorm(net1.conv(input)))
In [23]:
modellist = nn.ModuleList([nn.Linear(3, 4), nn.ReLU(), nn.Linear(4, 2)])
input = t.randn(1, 3)
for model in modellist:
    input = model(input)
# 下面會報錯,因為 modellist 沒有實現 forward 方法
# output = modellist(input)

```

為何不直接使用Python中自帶的list，而非要多此一舉呢？

這是因為ModuleList是Module的子類，當在Module中使用它的時候，就能自動識別為子module。下面舉例說明。

```
In [24]:
class MyModule(nn.Module):
    def __init__(self):
        super(MyModule, self).__init__()
        self.list = [nn.Linear(3, 4), nn.ReLU()]
        self.module_list = nn.ModuleList([nn.Conv2d(3, 3, 3), nn.ReLU()])
    def forward(self):
        pass
model = MyModule()
model
Out[24]:
MyModule(
  (module_list): ModuleList(
    (0): Conv2d(3, 3, kernel_size=(3, 3), stride=(1, 1))
    (1): ReLU()
  )
)
In [25]:
for name, param in model.named_parameters():
    print(name, param.size())
module_list.0.weight torch.Size([3, 3, 3, 3])
module_list.0.bias torch.Size([3])
```

可見，list中的子module並不能被主module所識別，而ModuleList中的子module能夠被主module所識別。這意味著如果用list保存子module，將無法調整其參數，因其未加入到主module的參數中。

除 ModuleList 之外還有 ParameterList，其是一個可以包含多個parameter的類list物件。在實際應用中，使用方式與ModuleList類似。如果在構造函數__init__中用到list、tuple、dict等物件時，一定要思考是否應該用ModuleList或ParameterList代替

4.1.3 迴圈神經網路層(RNN)

近些年隨著深度學習和自然語言處理的結合加深，RNN的使用也越來越多，關於RNN的基礎知識，推薦閱讀colah的文章1入門。PyTorch中實現了如今最常用的三種RNN：RNN（vanilla RNN）、LSTM和GRU。此外還有對應的三種RNNCell。

RNN和RNNCell層的區別在於前者一次能夠處理整個序列，而後者一次只處理序列中一個時間點的資料，前者封裝更完備更易於使用，後者更具靈活性。實際上RNN層的一種後端實現方式就是調用RNNCell來實現的。

<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

```
In [26]:
t.manual_seed(1000)
# 輸入：batch_size=3，序列長度都為 2，序列中每個元素占 4 維
input = t.randn(2, 3, 4)
# lstm 輸入向量 4 維，隱藏元 3，1 層
lstm = nn.LSTM(4, 3, 1)
# 初始狀態：1 層，batch_size=3，3 個隱藏元
h0 = t.randn(1, 3, 3)
c0 = t.randn(1, 3, 3)
out, hn = lstm(input, (h0, c0))
out
Out[26]:
tensor([[[[-0.3610, -0.1643,  0.1631],
          [-0.0613, -0.4937, -0.1642],
          [ 0.5080, -0.4175,  0.2502]],

          [[-0.0703, -0.0393, -0.0429],
          [ 0.2085, -0.3005, -0.2686],
          [ 0.1482, -0.4728,  0.1425]]]])
```

```
In [27]:
t.manual_seed(1000)
input = t.randn(2, 3, 4)
# 一個 LSTMCell 對應的層數只能是一層
lstm = nn.LSTMCell(4, 3)
hx = t.randn(3, 3)
cx = t.randn(3, 3)
out = []
for i_ in input:
    hx, cx=lstm(i_, (hx, cx))
    out.append(hx)
t.stack(out)
Out[27]:
tensor([[[[-0.3610, -0.1643,  0.1631],
          [-0.0613, -0.4937, -0.1642],
          [ 0.5080, -0.4175,  0.2502]],

          [[-0.0703, -0.0393, -0.0429],
          [ 0.2085, -0.3005, -0.2686],
          [ 0.1482, -0.4728,  0.1425]]]])
```

詞向量在自然語言中應用十分普及，PyTorch同樣提供了Embedding層。

```
In [29]:  
# 有 4 個詞，每個詞用 5 維的向量表示  
embedding = nn.Embedding(4, 5)  
# 可以用預訓練好的詞向量初始化 embedding  
embedding.weight.data = t.arange(0, 20).view(4, 5)  
In [30]:  
input = t.arange(3, 0, -1).long()  
output = embedding(input)  
output  
Out[30]:  
tensor([[ 15.,  16.,  17.,  18.,  19.],  
        [ 10.,  11.,  12.,  13.,  14.],  
        [  5.,   6.,   7.,   8.,   9.]])
```


4.1.4 損失函數

在深度學習中要用到各種各樣的損失函數（loss function），這些損失函數可看作是一種特殊的layer，PyTorch也將這些損失函數實現為nn.Module的子類。然而在實際使用中通常將這些loss function專門提取出來，和主模型互相獨立。詳細的loss使用請參照文檔1，這裡以分類中最常用的交叉熵損失CrossEntropyloss為例說明。

<http://pytorch.org/docs/nn.html#loss-functions>

```
In [31]:  
# batch_size=3，計算對應每個類別的分數（只有兩個類別）  
score = t.randn(3, 2)  
# 三個樣本分別屬於1，0，1類，label 必須是 LongTensor  
label = t.Tensor([1, 0, 1]).long()  
  
# loss 與普通的 layer 無差異  
criterion = nn.CrossEntropyLoss()  
loss = criterion(score, label)  
loss  
Out[31]:  
tensor(0.5944)
```


4.2 優化器

PyTorch 將深度學習中常用的優化方法全部封裝在 `torch.optim` 中，其設計十分靈活，能夠很方便的擴展成自訂的優化方法。

所有的優化方法都是繼承基類 `optim.Optimizer`，並實現了自己的優化步驟。下面就以最基本的優化方法——隨機梯度下降法（SGD）舉例說明。這裡需重點掌握：

1. 優化方法的基本使用方法
2. 如何對模型的不同部分設置不同的學習率
3. 如何調整學習率

```

In [32]:
# 首先定義一個 LeNet 網路
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 6, 5),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(6, 16, 5),
            nn.ReLU(),
            nn.MaxPool2d(2, 2)
        )
        self.classifier = nn.Sequential(
            nn.Linear(16 * 5 * 5, 120),
            nn.ReLU(),
            nn.Linear(120, 84),
            nn.ReLU(),
            nn.Linear(84, 10)
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(-1, 16 * 5 * 5)
        x = self.classifier(x)
        return x

net = Net()
In [33]:
from torch import optim
optimizer = optim.SGD(params=net.parameters(), lr=1)
optimizer.zero_grad() # 梯度清零，等價於 net.zero_grad()

input = t.randn(1, 3, 32, 32)
output = net(input)
output.backward(output) # fake backward

optimizer.step() # 執行優化

```

```

In [43]:
# 為不同子網路設置不同的學習率，在 finetune 中經常用到
# 如果對某個參數不指定學習率，就使用最外層的默認學習率
optimizer = optim.SGD([
    {'params': net.features.parameters(), # 學習率為 1e-5
    {'params': net.classifier.parameters(), 'lr': 1e-2}
], lr=1e-5)

optimizer
Out[43]:
SGD (
Parameter Group 0
  dampening: 0
  lr: 1e-05
  momentum: 0
  nesterov: False
  weight_decay: 0

Parameter Group 1
  dampening: 0
  lr: 0.01
  momentum: 0
  nesterov: False
  weight_decay: 0
)

```

```

In [44]:
# 只為兩個全連接層設置較大的學習率，其餘層的學習率較小
special_layers = nn.ModuleList([net.classifier[0], net.classifier[3]])
special_layers_params = list(map(id, special_layers.parameters()))
base_params = filter(lambda p: id(p) not in special_layers_params,
                      net.parameters())

optimizer = t.optim.SGD([
    {'params': base_params},
    {'params': special_layers.parameters(), 'lr': 0.01}
], lr=0.001)

optimizer
Out[44]:
SGD (
Parameter Group 0
  dampening: 0
  lr: 0.001
  momentum: 0
  nesterov: False
  weight_decay: 0

Parameter Group 1
  dampening: 0
  lr: 0.01
  momentum: 0
  nesterov: False
  weight_decay: 0
)

```

對於如何調整學習率，主要有兩種做法。一種是修改 `optimizer.param_groups` 中對應的學習率，另一種是更簡單也是較為推薦的做法——新建優化器，由於 `optimizer` 十分輕量級，構建開銷很小，故而可以構建新的 `optimizer`。但是後者對於使用動量的優化器（如 Adam），會丟失動量等狀態資訊，可能會造成損失函數的收斂出現震盪等情況。

```
In [48]:
# 方法 1: 調整學習率，新建一個 optimizer
old_lr = 0.1
optimizer1 = optim.SGD([
    {'params': net.features.parameters()},
    {'params': net.classifier.parameters(), 'lr': old_lr*0.1}
], lr=1e-5)

optimizer1
Out[48]:
SGD (
  Parameter Group 0
    dampening: 0
    lr: 1e-05
    momentum: 0
    nesterov: False
    weight_decay: 0

  Parameter Group 1
    dampening: 0
    lr: 0.010000000000000002
    momentum: 0
    nesterov: False
    weight_decay: 0
)
```

```
In [49]:
# 方法 2: 調整學習率，手動 decay，保存動量
for param_group in optimizer.param_groups:
    param_group['lr'] *= 0.1 # 學習率為之前的 0.1 倍
optimizer
Out[49]:
SGD (
  Parameter Group 0
    dampening: 0
    lr: 1.0000000000000002e-06
    momentum: 0
    nesterov: False
    weight_decay: 0

  Parameter Group 1
    dampening: 0
    lr: 0.0010000000000000002
    momentum: 0
    nesterov: False
    weight_decay: 0
)
```

4.3 NN.FUNCTIONAL

Nn中還有一個很常用的模組：`nn.functional`，`nn`中的大多數layer，在functional中都有一個與之相對應的函數。`nn.functional`中的函數和`nn.Module`的主要區別在於，用`nn.Module`實現的layers是一個特殊的類，都是由`class layer(nn.Module)`定義，會自動提取可學習的參數。而`nn.functional`中的函數更像是純函數，由`def function(input)`定義。下面舉例說明functional的使用，並指出二者的不同之處。

```
In [50]:
input = t.randn(2, 3)
model = nn.Linear(3, 4)
output1 = model(input)
output2 = nn.functional.linear(input, model.weight, model.bias)
output1 == output2
Out[50]:
tensor([[ 1,  1,  1,  1],
        [ 1,  1,  1,  1]], dtype=torch.uint8)

In [51]:
b = nn.functional.relu(input)
b2 = nn.ReLU()(input)
b == b2
Out[51]:
tensor([[ 1,  1,  1],
        [ 1,  1,  1]], dtype=torch.uint8)
```



應該什麼時候使用 `nn.Module`，什麼時候使用 `nn.functional` 呢？

如果模型有可學習的參數，最好用 `nn.Module`，否則既可以使用 `nn.functional` 也可以使用 `nn.Module`，二者在性能上沒有太大差異，具體的使用取決於個人的喜好。

如啟動函數（ReLU、sigmoid、tanh），池化（MaxPool）等層由於沒有可學習參數，則可以使用對應的 `functional` 函數代替，而對於卷積、全連接等具有可學習參數的網路建議使用 `nn.Module`。

另外雖然 `dropout` 操作也沒有可學習操作，但建議還是使用 `nn.Dropout` 而不是 `nn.functional.dropout`，因為 `dropout` 在訓練和測試兩個階段的行為有所差別，使用 `nn.Module` 物件能夠通過 `model.eval` 操作加以區分。

下面舉例說明，如何在模型中搭配使用nn.Module和nn.functional。

```
In [52]:
from torch.nn import functional as F
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = F.pool(F.relu(self.conv1(x)), 2)
        x = F.pool(F.relu(self.conv2(x)), 2)
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

對於不具備可學習參數的層（啟動層、池化層等），將它們用函數代替，這樣則可以不用放置在構造函數__init__中。對於有可學習參數的模組，也可以用functional來代替，只不過實現起來較為繁瑣，需要手動定義參數parameter，如前面實現自訂的全連接層，就可將weight和bias兩個參數單獨拿出來，在構造函數中初始化為parameter。

```
In [53]:
class MyLinear(nn.Module):
    def __init__(self):
        super(MyLinear, self).__init__()
        self.weight = nn.Parameter(t.randn(3, 4))
        self.bias = nn.Parameter(t.zeros(3))
    def forward(self):
        return F.linear(input, weight, bias)
```

關於nn.functional的設計初衷，以及它和nn.Module更多的比較說明，可參看論壇的討論和作者說明1。

<https://discuss.pytorch.org/search?q=nn.functional>

4.4 初始化策略

在深度學習中參數的初始化十分重要，良好的初始化能讓模型更快收斂，並達到更高水準，而糟糕的初始化則可能使得模型迅速癱瘓。

- PyTorch中`nn.Module`的模組參數都採取了較為合理的初始化策略，因此一般不用我們考慮，當然我們也可以用自訂初始化去代替系統的預設初始化。
- 而當我們在使用`Parameter`時，自訂初始化則尤為重要，因`t.Tensor()`返回的是記憶體中的亂數，很可能會有極大值，這在實際訓練網路中會造成溢出或者梯度消失。
- PyTorch中`nn.init`模組就是專門為初始化而設計，如果某種初始化策略`nn.init`不提供，用戶也可以自己直接初始化。

```
In [55]:
# 利用 nn.init 初始化
from torch.nn import init
linear = nn.Linear(3, 4)

t.manual_seed(1)
# 等價於 linear.weight.data.normal_(0, std)
init.xavier_normal_(linear.weight)
Out[55]:
Parameter containing:
tensor([[ 0.3535,  0.1427,  0.0330],
        [ 0.3321, -0.2416, -0.0888],
        [-0.8140,  0.2040, -0.5493],
        [-0.3010, -0.4769, -0.0311]])
```

```
In [56]:
# 直接初始化
import math
t.manual_seed(1)

# xavier 初始化的計算公式
std = math.sqrt(2)/math.sqrt(7.)
linear.weight.data.normal_(0, std)
Out[56]:
tensor([[ 0.3535,  0.1427,  0.0330],
        [ 0.3321, -0.2416, -0.0888],
        [-0.8140,  0.2040, -0.5493],
        [-0.3010, -0.4769, -0.0311]])

In [57]:
# 對模型的所有參數進行初始化
for name, params in net.named_parameters():
    if name.find('linear') != -1:
        # init linear
        params[0] # weight
        params[1] # bias
    elif name.find('conv') != -1:
        pass
    elif name.find('norm') != -1:
        pass
```

4.5 Nn.Module深入分析

如果想要更深入地理解nn.Module，究其原理是很有必要的。首先來看看nn.Module基類的構造函數：

```
def __init__(self):  
    self._parameters = OrderedDict()  
    self._modules = OrderedDict()  
    self._buffers = OrderedDict()  
    self._backward_hooks = OrderedDict()  
    self._forward_hooks = OrderedDict()  
    self.training = True
```

其中每個屬性的解釋如下：

- `_Parameters`：字典，保存使用者直接設置的parameter，`Self.param1 = nn.Parameter(t.randn(3, 3))`會被檢測到，在字典中加入一個key為'param'，value為對應parameter的item。而`self.submodule = nn.Linear(3, 4)`中的parameter則不會存於此。
- `_Modules`：子module，通過`self.submodel = nn.Linear(3, 4)`指定的子module會保存於此。
- `_Buffers`：緩存。如batchnorm使用momentum機制，每次前向傳播需用到上一次前向傳播的結果。
- `_Backward_hooks`與`_forward_hooks`：鉤子技術，用來提取中間變數，類似variable的hook。
- `Training`：BatchNorm與Dropout層在訓練階段和測試階段中採取的策略不同，通過判斷training值來決定前向傳播策略。

上述幾個屬性中，`_parameters`、`_modules`和`_buffers`這三個字典中的鍵值，都可以通過`self.key`方式獲得，效果等價於`self._parameters['key']`。

下面舉例說明。

```
In [58]:
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # 等價與 self.register_parameter('param1', nn.Parameter(t.randn(3, 3)))
        self.param1 = nn.Parameter(t.rand(3, 3))
        self.submodell = nn.Linear(3, 4)
    def forward(self, input):
        x = self.param1.mm(input)
        x = self.submodell(x)
        return x

net = Net()
net
Out[58]:
Net(
  (submodell): Linear(in_features=3, out_features=4, bias=True)
)

In [59]:
net._modules
Out[59]:
OrderedDict([('submodell', Linear(in_features=3, out_features=4, bias=True))])

In [60]:
net._parameters
Out[60]:
OrderedDict([('param1', Parameter containing:
      tensor([[ 0.3398,  0.5239,  0.7981],
               [ 0.7718,  0.0112,  0.8100],
               [ 0.6397,  0.9743,  0.8300]]))])

In [61]:
net.param1 # 等價於 net._parameters['param1']
Out[61]:
Parameter containing:
tensor([[ 0.3398,  0.5239,  0.7981],
         [ 0.7718,  0.0112,  0.8100],
         [ 0.6397,  0.9743,  0.8300]])
```

```
In [62]:
for name, param in net.named_parameters():
    print(name, param.size())
param1 torch.Size([3, 3])
submodell.weight torch.Size([4, 3])
submodell.bias torch.Size([4])

In [63]:
for name, submodel in net.named_modules():
    print(name, submodel)
Net(
  (submodell): Linear(in_features=3, out_features=4, bias=True)
)
submodell Linear(in_features=3, out_features=4, bias=True)

In [64]:
bn = nn.BatchNorm1d(2)
input = t.rand(3, 2)
output = bn(input)
bn._buffers
Out[64]:
OrderedDict([('running_mean', tensor(1.00000e-02 *
      [ 5.1362,  7.4864])),
             ('running_var', tensor([ 0.9116,  0.9068]))])
```

Nn.Module在實際使用中可能層層嵌套，一個module包含若干個子module，每一個子module又包含了更多的子module。為方便用戶訪問各個子module，nn.Module實現了很多方法，如函數children可以查看直接子module，函數module可以查看所有的子module（包括當前module）。與之相對應的還有函數named_children和named_modules，其能夠在返回module列表的同時返回它們的名字。

```
In [65]:
input = t.arange(0, 12).view(3, 4)
model = nn.Dropout()
# 在訓練階段，會有一半左右的數被隨機置為 0
model(input)
Out[65]:
tensor([[ 0.,  2.,  0.,  0.],
        [ 8.,  0., 12., 14.],
        [16.,  0.,  0., 22.]])

In [66]:
model.training = False
# 在測試階段，dropout 什麼都不做
model(input)
Out[66]:
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]])
```


對於 batchnorm 、 dropout 、instancenorm 等在訓練和測試階段行為差距巨大的層，如果在測試時不將其training值設為True，則可能會有很大影響，這在實際使用中要千萬注意。雖然可通過直接設置training屬性，來將子module設為train和eval模式，但這種方式較為繁瑣，因如果一個模型具有多個dropout層，就需要為每個dropout層指定training屬性。更為推薦的做法是調用model.train()函數，它會將當前module及其子module中的所有training屬性都設為True，相應的，model.eval()函數會把training屬性都設為False。

```
In [67]:
print(net.training, net.submodell.training)
net.eval()
net.training, net.submodell.training
True True
Out[67]:
(False, False)
In [68]:
list(net.named_modules())
Out[68]:
[('', Net(
  (submodell): Linear(in_features=3, out_features=4, bias=True)
)), ('submodell', Linear(in_features=3, out_features=4, bias=True))]
```


- Register_forward_hook與Register_backward_hook，這兩個函數的功能類似於variable函數的register_hook，可在module前向傳播或反向傳播時註冊鉤子。每次前向傳播執行結束後會執行鉤子函數（hook）。
- 前向傳播的鉤子函數具有如下形式：Hook(module, input, output) -> None，而反向傳播則具有如下形式：Hook(module, grad_input, grad_output) -> Tensor or None。
- 鉤子函數不應修改輸入和輸出，並且在使用後應及時刪除，以避免每次都運行鉤子增加運行負載。鉤子函數主要用在獲取某些中間結果的情景，如中間某一層的輸出或某一層的梯度。
- 這些結果本應寫在forward函數中，但如果在forward函數中專門加上這些處理，可能會使處理邏輯比較複雜，這時候使用鉤子技術就更合適一些。

- 下面考慮一種場景，有一個預訓練好的模型，需要提取模型的某一層（不是最後一層）的輸出作為特徵進行分類，但又不希望修改其原有的模型定義檔，這時就可以利用鉤子函數。
- 下面給出實現的偽代碼。

```
model = VGG()
features = t.Tensor()
def hook(module, input, output):
    '''把這層的輸出拷貝到 features 中'''
    features.copy_(output.data)

handle = model.layer8.register_forward_hook(hook)
_ = model(input)
# 用完 hook 後刪除
handle.remove()
```

Nn.Module物件在構造函數中的行為看起來有些怪異，如果想要真正掌握其原理，就需要看兩個魔法方法__getattr__和__setattr__。

在Python中有兩個常用的builtin方法getattr和setattr，getattr(obj, 'attr1')等價於obj.attr，如果getattr函數無法找到所需屬性，Python會轉而調用obj.__getattr__('attr1')方法，即getattr函數無法找到的交給__getattr__函數處理，沒有實現__getattr__或者__getattr__也無法處理的就會raise AttributeError。

setattr(obj, 'name', value)等價於obj.name=value，如果obj物件實現了__setattr__方法，setattr會直接調用obj.__setattr__('name', value)，否則調用builtin方法。

◆總結一下：

1. Result = obj.name會調用builtin函數getattr(obj, 'name')，如果該屬性找不到，會調用obj.__getattr__('name')
2. Obj.name = value會調用builtin函數setattr(obj, 'name', value)，如果obj物件實現了__setattr__方法，setattr會直接調用obj.__setattr__('name', value)

Nn.Module實現了自訂的__setattr__函數，當執行module.name=value時，會在__setattr__中判斷value是否為Parameter或nn.Module物件，如果是則將這些物件加到_parameters和_modules兩個字典中，而如果是其它類型的物件，如Variable、list、dict等，則調用默認的操作，將這個值保存在__dict__中。

```
In [69]:
module = nn.Module()
module.param = nn.Parameter(t.ones(2, 2))
module._parameters
Out[69]:
OrderedDict([('param', Parameter containing:
              tensor([ [ 1.,  1.],
                       [ 1.,  1.] ]))])

In [70]:
submodule1 = nn.Linear(2, 2)
submodule2 = nn.Linear(2, 2)
module_list = [submodule1, submodule2]
# 對於 list 物件，調用 buildin 函數，保存在__dict__中
module.submodules = module_list
print('_modules: ', module._modules)
print("__dict__['submodules']:", module.__dict__.get('submodules'))
_modules: OrderedDict()
__dict__['submodules']: [Linear(in_features=2, out_features=2, bias=True),
Linear(in_features=2, out_features=2, bias=True)]
In [71]:
module_list = nn.ModuleList(module_list)
module.submodules = module_list
print('ModuleList is instance of nn.Module: ', isinstance(module_list, nn.Module))
print('_modules: ', module._modules)
print("__dict__['submodules']:", module.__dict__.get('submodules'))
ModuleList is instance of nn.Module: True
_modules: OrderedDict([('submodules', ModuleList(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))])
__dict__['submodules']: None
```

因 `_modules` 和 `_parameters` 中的 item 未保存在 `__dict__` 中，所以默認的 `getattr` 方法無法獲取它，因而 `nn.Module` 實現了自訂的 `__getattr__` 方法，如果默認的 `getattr` 無法處理，就調用自訂的 `__getattr__` 方法，嘗試從 `_modules`、`_parameters` 和 `_buffers` 這三個字典中獲取。

```
In [74]:
getattr(module, 'training') # 等價於 module.training
# error
# module.__getattr__('training')
Out[74]:
True
In [75]:
module.attr1 = 2
getattr(module, 'attr1')
# 報錯
# module.__getattr__('attr1')
Out[75]:
2

In [76]:
# 即 module.param, 會調用 module.__getattr__('param')
getattr(module, 'param')
Out[76]:
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
```

在PyTorch中保存模型十分簡單，所有的Module物件都具有state_dict()函數，返回當前Module所有的狀態資料。

將這些狀態資料保存後，下次使用模型時即可利用model.load_state_dict()函數將狀態載入進來。優化器(optimizer)也有類似的機制，不過一般並不需要保存優化器的運行狀態。

```
In [77]:  
# 保存模型  
t.save(net.state_dict(), 'net.pth')  
  
# 載入已保存的模型  
net2 = Net()  
net2.load_state_dict(t.load('net.pth'))
```


實際上還有另外一種保存方法，但因其嚴重依賴模型定義方式及檔路徑結構等，很容易出問題，因而不建議使用。

```
In [56]:
t.save(net, 'net_all.pth')
net2 = t.load('net_all.pth')
net2
/usr/local/lib/python3.5/dist-packages/torch/serialization.py:158: UserWarning: Couldn't
retrieve source code for container of type Net. It won't be checked for correctness upon loading.
  "type " + obj.__name__ + ". It won't be checked "
Out[56]:
Net(
  (submodel1): Linear(in_features=3, out_features=4)
)
```


將Module放在GPU上運行也十分簡單，只需兩步：

1. `Model = model.cuda()`：將模型的所有參數轉存到GPU
2. `Input.cuda()`：將輸入資料也放置到GPU上

至於如何在多個GPU上平行計算，PyTorch也提供了兩個函數，可實現簡單高效的並行GPU計算

1. `Nn.parallel.data_parallel(module, inputs, device_ids=None, output_device=None, dim=0, module_kwargs=None)`
2. Class `torch.nn.DataParallel(module, device_ids=None, output_device=None, dim=0)`

可見二者的參數十分相似，通過`device_ids`參數可以指定在哪些GPU上進行優化，`Output_device`指定輸出到哪個GPU上。唯一的不同就在於前者直接利用多GPU平行計算得出結果，而後者則返回一個新的module，能夠自動在多GPU上進行並行加速。

```
# method 1
new_net = nn.DataParallel(net, device_ids=[0, 1])
output = new_net(input)

# method 2
output = nn.parallel.data_parallel(new_net, input, device_ids=[0, 1])
```

DataParallel並行的方式，是將輸入一個batch的資料均分成多份，分別送到對應的GPU進行計算，各個GPU得到的梯度累加。與Module相關的所有資料也都會以淺複製的方式複製多份，在此需要注意，在module中屬性應該是唯讀的。

4.6 Nn和Autograd的關係

Nn.Module利用的也是autograd技術，其主要工作是實現前向傳播。在forward函數中，nn.Module對輸入的tensor進行的各種操作，本質上都是用到了autograd技術。這裡需要對比autograd.Function和nn.Module之間的區別：

1. Autograd.Function利用了Tensor對autograd技術的擴展，為autograd實現了新的運算op，不僅要實現前向傳播還要手動實現反向傳播
2. Nn.Module利用了autograd技術，對nn的功能進行擴展，實現了深度學習中更多的層。只需實現前向傳播功能，Autograd即會自動實現反向傳播
3. Nn.functional是一些autograd操作的集合，是經過封裝的函數

作為兩大類擴充PyTorch介面的方法，我們在實際使用中應該如何選擇呢？如果某一個操作，在autograd中尚未支持，那麼只能實現Function介面對應的前向傳播和反向傳播。如果某些時候利用autograd介面比較複雜，則可以利用Function將多個操作聚合，實現優化，正如第三章所實現的Sigmoid一樣，比直接利用autograd低級別的操作要快。而如果只是想在深度學習中增加某一層，使用nn.Module進行封裝則更為簡單高效。

4.7 小試牛刀：搭建ResNet

Kaiming He的深度殘差網路（ResNet）^[7]在深度學習的發展中起到了很重要的作用，ResNet不僅一舉拿下了當年CV下多個比賽項目的冠軍，更重要的是這一結構解決了訓練極深網路時的梯度消失問題。

首先來看看ResNet的網路結構，這裡選取的是ResNet的一個變種：ResNet34。ResNet的網路結構如圖4-2所示，可見除了最開始的卷積池化和最後的池化全連接之外，網路中有很多結構相似的單元，這些重複單元的共同點就是有個跨層直連的shortcut。

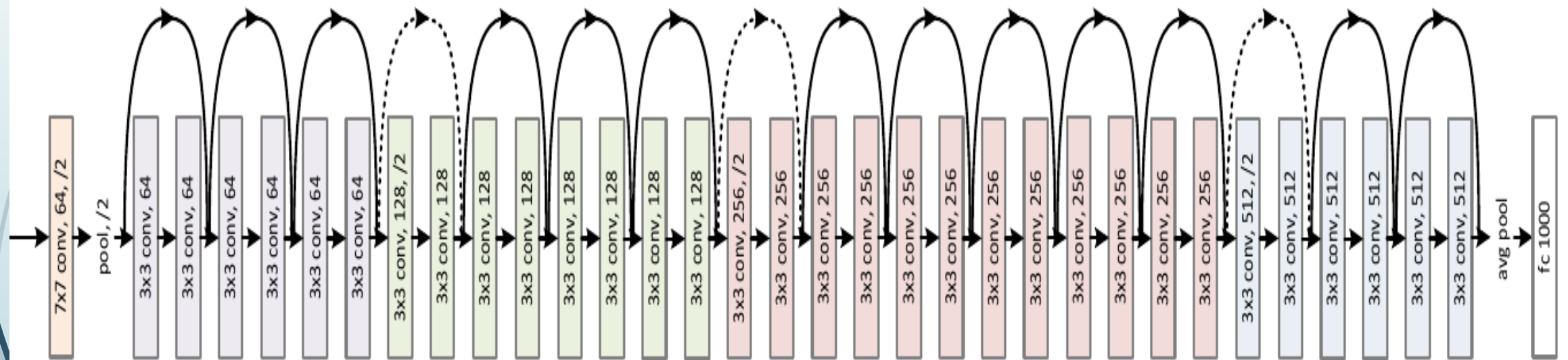


圖4-2 ResNet的網路結構

ResNet中將一個跨層直連的單元稱為Residual block，其結構如圖4-3所示，左邊部分是普通的卷積網路結構，右邊是直連，但如果輸入和輸出的通道數不一致，或其步長不為1，那麼就需要有一個專門的單元將二者轉成一致，使其可以相加。

另外我們可以發現Residual block的大小也是有規律的，在最開始的pool之後有連續的幾個一模一樣的Residual block單元，這些單元的通道數一樣，在這裡我們將這幾個擁有多個Residual block單元的結構稱之為layer，注意和之前講的layer區分開來，這裡的layer是幾個層的集合。

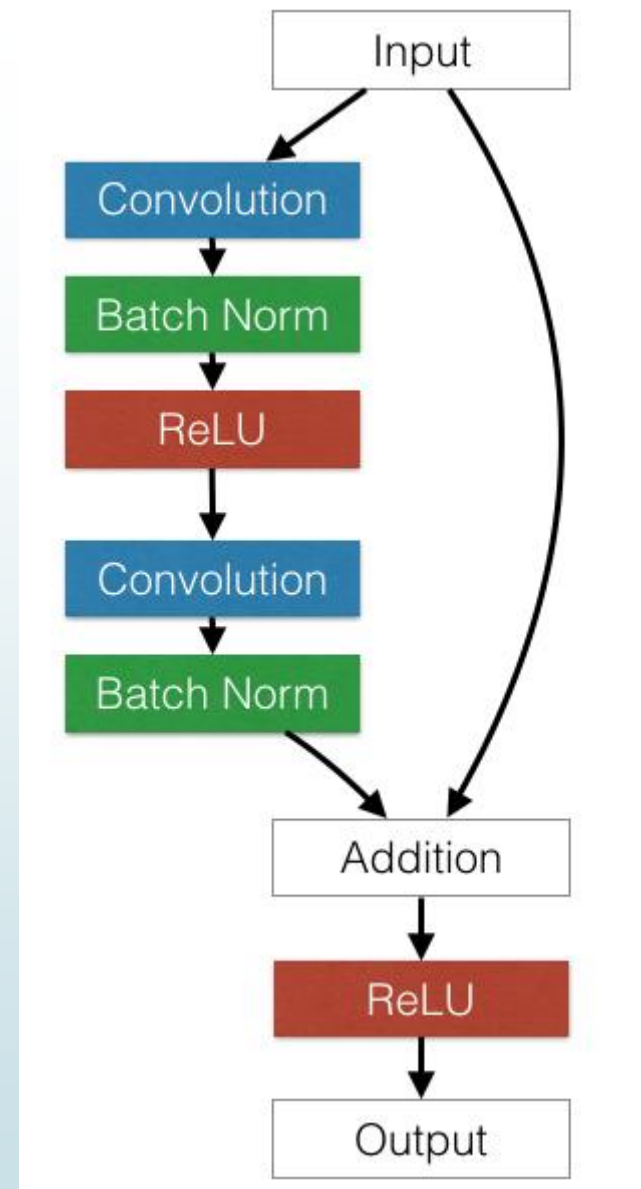



圖4-3 Residual block結構圖



考慮到Residual block和layer出現了多次，我們可以把它們實現為一個子Module或函數。這裡我們將Residual block實現為一個子module，而將layer實現為一個函數。

下面是實現代碼，規律總結如下：

1. 對於模型中的重複部分，實現為子module或用函數生成相應的modulemake_layer
2. Nn.Module和nn.Functional結合使用
3. 儘量使用nn.Sequential


```

In [78]:
from torch import nn
import torch as t
from torch.nn import functional as F
In [79]:
class ResidualBlock(nn.Module):
    ...

    實現子 module: Residual Block
    ...

    def __init__(self, inchannel, outchannel, stride=1, shortcut=None):
        super(ResidualBlock, self).__init__()
        self.left = nn.Sequential(
            nn.Conv2d(inchannel, outchannel, 3, stride, 1, bias=False),
            nn.BatchNorm2d(outchannel),
            nn.ReLU(inplace=True),
            nn.Conv2d(outchannel, outchannel, 3, 1, 1, bias=False),
            nn.BatchNorm2d(outchannel) )
        self.right = shortcut

    def forward(self, x):
        out = self.left(x)
        residual = x if self.right is None else self.right(x)
        out += residual
        return F.relu(out)

class ResNet(nn.Module):
    ...

    實現主 module: ResNet34
    ResNet34 包含多個 layer，每個 layer 又包含多個 residual block
    用子 module 來實現 residual block，用 _make_layer 函數來實現 layer
    ...

    def __init__(self, num_classes=1000):
        super(ResNet, self).__init__()
        # 前幾層圖像轉換
        self.pre = nn.Sequential(
            nn.Conv2d(3, 64, 7, 2, 3, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(3, 2, 1))

        # 重複的 layer，分別有 3, 4, 6, 3 個 residual block
        self.layer1 = self._make_layer( 64, 64, 3)
        self.layer2 = self._make_layer( 64, 128, 4, stride=2)

```

```

        self.layer3 = self._make_layer( 128, 256, 6, stride=2)
        self.layer4 = self._make_layer( 256, 512, 3, stride=2)

        # 分類用的全連接
        self.fc = nn.Linear(512, num_classes)

    def _make_layer(self, inchannel, outchannel, block_num, stride=1):
        ...

        構建 layer, 包含多個 residual block
        ...

        shortcut = nn.Sequential(
            nn.Conv2d(inchannel, outchannel, 1, stride, bias=False),
            nn.BatchNorm2d(outchannel))

        layers = []
        layers.append(ResidualBlock(inchannel, outchannel, stride, shortcut))

        for i in range(1, block_num):
            layers.append(ResidualBlock(outchannel, outchannel))
        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.pre(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = F.avg_pool2d(x, 7)
        x = x.view(x.size(0), -1)
        return self.fc(x)

In [80]:
model = ResNet()
input = t.randn(1, 3, 224, 224)
o = model(input)

```

感興趣的讀者可以嘗試實現Google的Inception網路結構或ResNet的其它變體，看看如何能夠簡潔明瞭地實現它，實現代碼儘量控制在80行以內（本例去掉空行和注釋總共不超過50行）。另外，與PyTorch配套的圖像工具包torchvision已經實現了深度學習中大多數經典的模型，其中就包括ResNet34，讀者可以通過下面兩行代碼使用：

```
from torchvision import models  
model = models.resnet34()
```

本例中ResNet34的實現就是參考了torchvision中的實現並做了簡化，感興趣的讀者可以閱讀相應的源碼，比較這裡的實現和torchvision中實現的不同。