Ch2　PyTorch 基礎練習

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

大綱

- 本節將先介紹一些PyTorch的基礎知識，使得讀者能夠對PyTorch有一個大致的瞭解，並能夠用PyTorch搭建一個簡單的神經網路。部分內容讀者可能暫時不太理解，可先不予以深究，本書的第3章和第4章將會對此進行深入講解。
- 本節內容參考了PyTorch官方教程1並做了相應的增刪修改，使得內容更貼合新版本的PyTorch介面，同時也更適合新手快速入門。另外本書需要讀者先掌握基礎的Numpy使用，其他相關知識推薦讀者參考CS231n的教程2。
- http://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html
- http://cs231n.github.io/python-numpy-tutorial/

Tensor

- Tensor是PyTorch中重要的資料結構，可認為是一個高維陣列。它可以是一個數（標量）、一維陣列（向量）、二維陣列（矩陣）以及更高維的陣列。Tensor和Numpy的Ndarrays類似，但Tensor可以使用GPU進行加速。Tensor的使用和Numpy及Matlab的介面十分相似，下面通過幾個例子來看看Tensor的基本使用。

```
In [1]:
from __future__ import print_function
import torch as t
t.__version__
Out[1]:
'0.4.0'
In [2]:
# 構建 5x3 矩陣，只是分配了空間，未初始化
x = t.Tensor(5, 3)

x = t.Tensor([[1,2],[3,4]])
x
Out[2]:
tensor([[ 1.,  2.],
        [ 3.,  4.]])
In [3]:
# 使用[0,1]均匀分佈隨機初始化二維陣列
x = t.rand(5, 3)
x
Out[3]:
tensor([[ 0.8052,  0.7188,  0.0332],
        [ 0.6054,  0.8955,  0.8972],
        [ 0.1107,  0.3319,  0.0336],
        [ 0.2394,  0.5188,  0.2201],
        [ 0.9730,  0.9370,  0.5677]])
In [4]:
print(x.size()) # 查看 x 的形狀
x.size()[1], x.size(1) # 查看列的個數，兩種寫法等價
torch.Size([5, 3])
Out[4]:
(3, 3)
torch.Size 是 tuple 物件的子類，因此它支持 tuple 的所有操作，如 x.size()[0]
```

```
In [5]:
y = t.rand(5, 3)
# 加法的第一種寫法
x + y
Out[5]:
tensor([[ 0.9639,  0.8763,  0.2834],
        [ 1.3785,  1.5090,  1.3919],
        [ 0.7139,  0.6348,  0.8439],
        [ 0.7022,  1.5079,  0.4776],
        [ 1.7892,  1.6383,  0.7774]])

In [6]:
# 加法的第二種寫法
t.add(x, y)
Out[6]:
tensor([[ 0.9639,  0.8763,  0.2834],
        [ 1.3785,  1.5090,  1.3919],
        [ 0.7139,  0.6348,  0.8439],
        [ 0.7022,  1.5079,  0.4776],
        [ 1.7892,  1.6383,  0.7774]])
In [7]:
# 加法的第三種寫法：指定加法結果的輸出目標為 result
result = t.Tensor(5, 3) # 預先分配空間
t.add(x, y, out=result) # 輸入到 result
result
Out[7]:
tensor([[ 0.9639,  0.8763,  0.2834],
        [ 1.3785,  1.5090,  1.3919],
        [ 0.7139,  0.6348,  0.8439],
        [ 0.7022,  1.5079,  0.4776],
        [ 1.7892,  1.6383,  0.7774]])
```

```
In [8]:
print('最初 y')
print(y)

print('第一種加法，y 的結果')
y.add(x) # 普通加法，不改變 y 的內容
print(y)

print('第二種加法，y 的結果')
y.add_(x) # inplace 加法，y 變了
print(y)
最初 y
tensor([[ 0.1587,  0.1575,  0.2501],
        [ 0.7732,  0.6135,  0.4947],
        [ 0.6033,  0.3029,  0.8103],
        [ 0.4628,  0.9891,  0.2575],
        [ 0.8163,  0.7013,  0.2097]])
第一種加法，y 的結果
tensor([[ 0.1587,  0.1575,  0.2501],
        [ 0.7732,  0.6135,  0.4947],
        [ 0.6033,  0.3029,  0.8103],
        [ 0.4628,  0.9891,  0.2575],
        [ 0.8163,  0.7013,  0.2097]])
第二種加法，y 的結果
tensor([[ 0.9639,  0.8763,  0.2834],
        [ 1.3785,  1.5090,  1.3919],
        [ 0.7139,  0.6348,  0.8439],
        [ 0.7022,  1.5079,  0.4776],
        [ 1.7892,  1.6383,  0.7774]])
```

- 注意，函數名後面帶底線_ 的函數會修改Tensor本身。
- 例如，x.add_(y)和x.t_()會改變 x，但x.add(y)和x.t()返回一個新的Tensor， 而x不變。

```
In [9]:
# Tensor 的選取操作與 Numpy 類似
x[:, 1]
Out[9]:
tensor([ 0.7188,  0.8955,  0.3319,  0.5188,  0.9370])
```

Tensor還支援很多操作，包括數學運算、線性代數、選擇、切片等等，其介面設計與Numpy極為相似。更詳細的使用方法，會在第三章系統講解。

Tensor和Numpy的陣列之間的交互操作非常容易且快速。對於Tensor不支援的操作，可以先轉為Numpy陣列處理，之後再轉回Tensor。

```
In [10]:
a = t.ones(5) #  新建一個全 1 的 Tensor
a
Out[10]:
tensor([ 1.,  1.,  1.,  1.,  1.])
In [11]:
b = a.numpy() # Tensor -> Numpy
b
Out[11]:
array([1., 1., 1., 1., 1.], dtype=float32)
In [12]:
import numpy as np
a = np.ones(5)
b = t.from_numpy(a) # Numpy->Tensor
print(a)
print(b)
[1. 1. 1. 1. 1.]
tensor([ 1.,  1.,  1.,  1.,  1.], dtype=torch.float64)
```

Tensor和Numpy物件共用記憶體，所以他們之間的轉換很快，而且幾乎不會消耗什麼資源。但這也意味著，如果其中一個變了，另外一個也會隨之改變。

```
In [13]:
b.add_(1) # 以`_`結尾的函數會修改自身
print(a)
print(b) # Tensor 和 Numpy 共用記憶體
[2. 2. 2. 2. 2.]
tensor([ 2.,  2.,  2.,  2.,  2.], dtype=torch.float64)
```

如果你想獲取某一個元素的值，可以使用scalar.item。直接tensor[idx]得到的還是一個tensor: 一個0-dim 的tensor，一般稱為scalar.

此外在pytorch中還有一個和np.array 很類似的介面: torch.tensor, 二者的使用十分類似。

```
In [14]:
scalar = b[0]
scalar
Out[14]:
tensor(2., dtype=torch.float64)
In [15]:
scalar.size() #0-dim

Out[15]:
torch.Size([])
In [16]:
scalar.item() # 使用 scalar.item()能從中取出 python 物件的數值
Out[16]:
2.0
In [17]:
tensor = t.tensor([2]) # 注意和 scalar 的區別
tensor,scalar
Out[17]:
(tensor([ 2]), tensor(2., dtype=torch.float64))
In [18]:
tensor.size(),scalar.size()
Out[18]:
(torch.Size([1]), torch.Size([]))
In [19]:
# 只有一個元素的 tensor 也可以調用`tensor.item()`
tensor.item(), scalar.item()
Out[19]:
(2, 2.0)
```

```
In [20]:
tensor = t.tensor([3,4]) # 新建一個包含 3，4 兩個元素的 tensor
In [21]:
scalar = t.tensor(3)
scalar
Out[21]:
tensor(3)
In [22]:
old_tensor = tensor
new_tensor = t.tensor(old_tensor)
new_tensor[0] = 1111
old_tensor, new_tensor
Out[22]:
(tensor([ 3,    4]), tensor([ 1111,        4]))
```

需要注意的是，t.tensor()總是會進行資料拷貝，新tensor和原來的資料不再共用記憶體。所以如果你想共用記憶體的話，建議使用 torch.from_numpy() 或者 tensor.detach()來新建一個tensor,二者共用記憶體。

```
In [23]:
new_tensor = old_tensor.detach()
new_tensor[0] = 1111
old_tensor, new_tensor
Out[23]:
(tensor([ 1111,        4]), tensor([ 1111,        4]))
```

Tensor 可通過.cuda 方法轉為 GPU 的 Tensor，從而享受 GPU 帶來的加速運算。

```
In [24]:
# 在不支持 CUDA 的機器下，下一步還是在 CPU 上運行
device = t.device("cuda:0" if t.cuda.is_available() else "cpu")
x = x.to(device)
y = y.to(device)
z = x+y
```

此外，還可以使用tensor.cuda() 的方式將tensor拷貝到gpu上，但是這種方式不太推薦。

此處可能發現GPU運算的速度並未提升太多，這是因為x和y太小且運算也較為簡單，而且將資料從記憶體轉移到顯存還需要花費額外的開銷。GPU的優勢需在大規模資料和複雜運算下才能體現出來。

Autograd: 自動微分

深度學習的演算法本質上是通過反向傳播求導數，而PyTorch的autograd模組則實現了此功能。在Tensor上的所有操作，Aautograd都能為它們自動提供微分，避免了手動計算導數的複雜過程。

Variable的資料結構

- 從0.4起, Variable 正式合併入Tensor, Variable本來實現的自動微分功能，Tensor就能支持。讀者還是可以使用Variable(tensor), 但是這個操作其實什麼都沒做。建議讀者以後直接使用tensor.
- 要想使得Tensor使用Autograd功能，只需要設置tensor.requries_grad=True.

```
In [25]:
# 為 tensor 設置 requires_grad 標識，代表著需要求導數
# pytorch 會自動調用 autograd 記錄操作
x = t.ones(2, 2, requires_grad=True)

# 上一步等價於
# x = t.ones(2,2)
# x.requires_grad = True

x
Out[25]:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
In [26]:
y = x.sum()
y
Out[26]:
tensor(4.)
In [27]:
y.grad_fn
Out[27]:
<SumBackward0 at 0x7ffaa589a780>
In [28]:
y.backward() # 反向傳播,計算梯度
In [29]:
# y = x.sum() = (x[0][0] + x[0][1] + x[1][0] + x[1][1])
# 每個值的梯度都為 1
x.grad
Out[29]:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
```

注意：Grad在反向傳播過程中是累加的(Accumulated)，這意味著每一次運行反向傳播，梯度都會累加之前的梯度，所以反向傳播之前需把梯度清零。

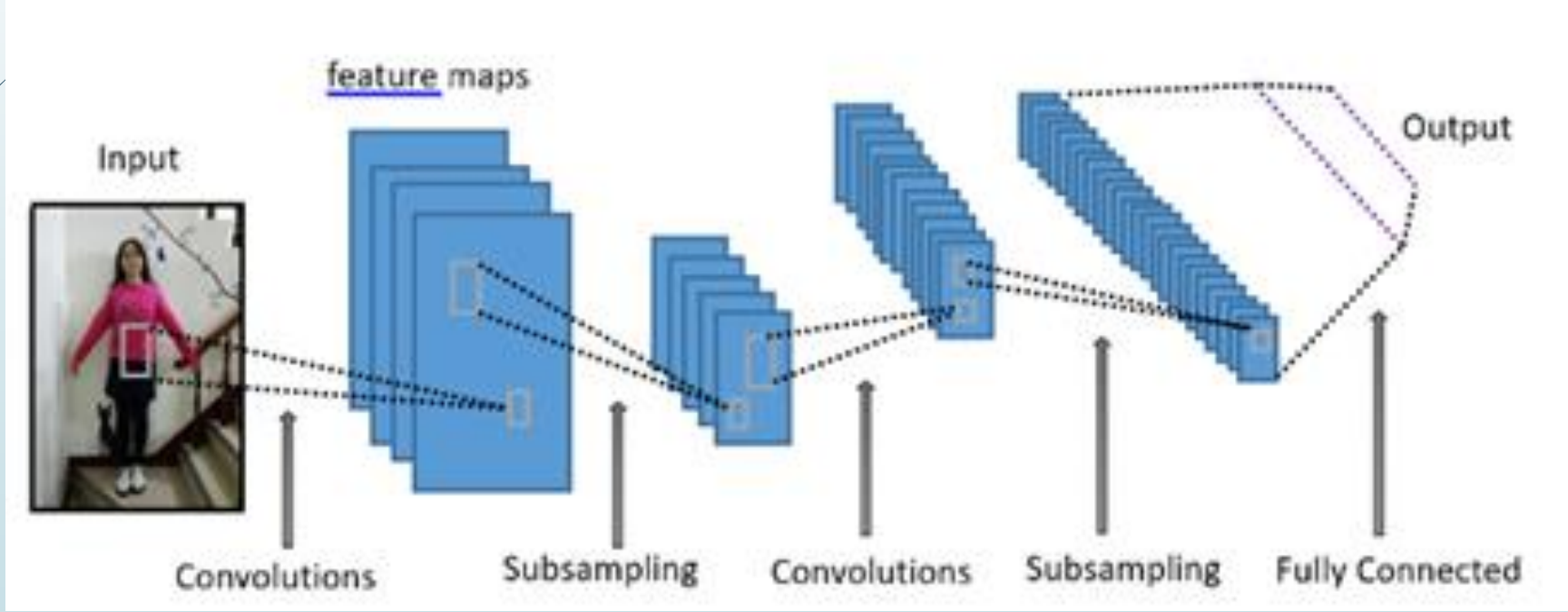
```
In [30]:
y.backward()
x.grad
Out[30]:
tensor([[ 2.,  2.],
        [ 2.,  2.]])

y.backward()
x.grad
Out[31]:
tensor([[ 3.,  3.],
        [ 3.,  3.]])
In [32]:
# 以底線結束的函數是 inplace 操作，會修改自身的值，就像 add_
x.grad.data.zero_()
Out[32]:
tensor([[ 0.,  0.],
        [ 0.,  0.]])
In [33]:
y.backward()
x.grad
Out[33]:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
```

神經網路

- Autograd實現了反向傳播功能，但是直接用來寫深度學習的代碼在很多情況下還是稍顯複雜，torch.nn是專門為神經網路設計的模組化介面。nn構建於 Autograd之上，可用來定義和運行神經網路。nn.Module是nn中最重要的類，可把它看成是一個網路的封裝，包含網路各層定義以及forward方法，調用forward(input)方法，可返回前向傳播的結果。下面就以最早的卷積神經網路：LeNet為例，來看看如何用nn.Module實現。

LeNet網路結構

- 這是一個基礎的前向傳播(feed-forward)網路: 接收輸入，經過層層傳遞運算，得到輸出。



定義網路

定義網路時，需要繼承nn.Module，並實現它的forward方法，把網路中具有可學習參數的層放在構造函數__init__中。如果某一層(如ReLU)不具有可學習的參數，則既可以放在構造函數中，也可以不放，但建議不放在其中，而在forward中使用nn.functional代替。

```
In [34]:
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        # nn.Module 子類的函數必須在構造函數中執行父類的構造函數
        #  下式等價於 nn.Module.__init__(self)
        super(Net, self).__init__()

        # 卷積層 '1'表示輸入圖片為單通道, '6'表示輸出通道數，'5'表示卷積核為 5*5
        self.conv1 = nn.Conv2d(1, 6, 5)
        # 卷積層
        self.conv2 = nn.Conv2d(6, 16, 5)
        # 仿射層/全連接層，y = Wx + b
        self.fc1   = nn.Linear(16*5*5, 120)
        self.fc2   = nn.Linear(120, 84)
        self.fc3   = nn.Linear(84, 10)

    def forward(self, x):
        # 卷積 -> 啟動 -> 池化
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        # reshape，  '-1'表示自我調整
        x = x.view(x.size()[0], -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

net = Net()
print(net)
Net(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

只要在nn.Module的子類中定義了forward函數，backward函數就會自動被實現(利用autograd)。在forward 函數中可使用任何tensor支援的函數，還可以使用if、for迴圈、print、log等Python語法，寫法和標準的Python寫法一致。

網路的可學習參數通過net.parameters()返回，net.named_parameters可同時返回可學習的參數及名稱。

```
In [35]:
params = list(net.parameters())
print(len(params))
10
In [36]:
for name,parameters in net.named_parameters():
        print(name,':',parameters.size())
conv1.weight : torch.Size([6, 1, 5, 5])
conv1.bias : torch.Size([6])
conv2.weight : torch.Size([16, 6, 5, 5])
conv2.bias : torch.Size([16])
fc1.weight : torch.Size([120, 400])
fc1.bias : torch.Size([120])
fc2.weight : torch.Size([84, 120])
fc2.bias : torch.Size([84])
fc3.weight : torch.Size([10, 84])
fc3.bias : torch.Size([10])
forward 函數的輸入和輸出都是 Tensor。

In [37]:
input = t.randn(1, 1, 32, 32)
out = net(input)
out.size()
Out[37]:
torch.Size([1, 10])
In [38]:
net.zero_grad() # 所有參數的梯度清零
out.backward(t.ones(1,10)) # 反向傳播
```

需要注意的是，torch.nn只支持mini-batches，不支持一次只輸入一個樣本，即一次必須是一個batch。但如果只想輸入一個樣本，則用input.unsqueeze(0)將batch_size設為 1。例如 nn.Conv2d 輸入必須是4維的，形如$nSamples \times nChannels \times Height \times Width$。可將nSample設為1，即$1 \times nChannels \times Height \times Width$。

損失函數

nn實現了神經網路中大多數的損失函數，例如nn.MSELoss用來計算均方誤差，nn.CrossEntropyLoss用來計算交叉熵損失。

```
In [39]:
output = net(input)
target = t.arange(0,10).view(1,10)
criterion = nn.MSELoss()
loss = criterion(output, target)
loss # loss 是個 scalar
Out[39]:
tensor(28.5625)
```

如果對 loss 進行反向傳播溯源(使用 gradfn 屬性)，可看到它的計算圖如下：

```
input -> conv2d -> relu -> maxpool2d -> conv2d -> relu -> maxpool2d
      -> view -> linear -> relu -> linear -> relu -> linear
      -> MSELoss
      -> loss
```

當調用loss.backward()時，該圖會動態生成並自動微分，也即會自動計算圖中參數(Parameter)的導數。

```
In [40]:
# 運行.backward，觀察調用之前和調用之後的 grad
net.zero_grad() # 把 net 中所有可學習參數的梯度清零
print('反向傳播之前  conv1.bias 的梯度')
print(net.conv1.bias.grad)
loss.backward()
print('反向傳播之後  conv1.bias 的梯度')
print(net.conv1.bias.grad)
反向傳播之前  conv1.bias 的梯度
tensor([ 0.,  0.,  0.,  0.,  0.,  0.])
反向傳播之後  conv1.bias 的梯度
tensor(1.00000e-02 *
       [-2.2701, -1.8687,  4.3639, -2.9262, -4.3535, -5.9584])
```

優化器

- 在反向傳播計算完所有參數的梯度後，還需要使用優化方法來更新網路的權重和參數，例如隨機梯度下降法(SGD)的更新策略如下：

```
weight = weight - learning_rate * gradient
```

手動實現如下：

```
learning_rate = 0.01
for f in net.parameters():
    f.data.sub_(f.grad.data * learning_rate)# inplace  減法
```

torch.optim 中實現了深度學習中絕大多數的優化方法，例如 RMSProp、Adam、SGD 等，更便於使用，因此大多數時候並不需要手動寫上述代碼。

```
In [41]:
import torch.optim as optim
#新建一個優化器，指定要調整的參數和學習率
optimizer = optim.SGD(net.parameters(), lr = 0.01)

# 在訓練過程中
# 先梯度清零(與 net.zero_grad()效果一樣)
optimizer.zero_grad()

# 計算損失
output = net(input)
loss = criterion(output, target)

#反向傳播
loss.backward()

#更新參數
optimizer.step()

In [42]:
import torchvision as tv
import torchvision.transforms as transforms
from torchvision.transforms import ToPILImage
show = ToPILImage() # 可以把 Tensor 轉成 Image，方便視覺化
```

數據載入與預處理

- 在深度學習中資料載入及預處理是非常複雜繁瑣的，但PyTorch提供了一些可極大簡化和加快資料處理流程的工具。同時，對於常用的資料集，PyTorch也提供了封裝好的接口供用戶快速調用，這些資料集主要保存在torchvison中。
- Torchvision實現了常用的圖像資料載入功能，例如Imagenet、CIFAR10、MNIST等，以及常用的資料轉換操作，這極大地方便了資料載入，並且代碼具有再使用性。

小試牛刀：CIFAR-10分類

- 下面我們來嘗試實現對CIFAR-10資料集的分類，步驟如下:

1. 使用Torchvision載入並預處理CIFAR-10資料集
2. 定義網路
3. 定義損失函數和優化器
4. 訓練網路並更新網路參數
5. 測試網路

CIFAR-10數據載入及預處理

CIFAR-101是一個常用的彩色圖片資料集，它有10個類別: ‘airplane', ‘automobile', ‘bird', ‘cat', ‘deer', ‘dog', ‘frog', ‘horse', ‘ship', ‘truck'。每張圖片都是$3\times32\times32$，也即3-通道彩色圖片，解析度為32×32

```
In [42]:
import torchvision as tv
import torchvision.transforms as transforms
from torchvision.transforms import ToPILImage
show = ToPILImage() # 可以把 Tensor 轉成 Image，方便视觉化
```

In [43]:

```
# 第一次運行程式 torchvision 會自動下載 CIFAR-10 資料集，
# 大約 100M，需花費一定的時間，
# 如果已經下載有 CIFAR-10，可通過 root 參數指定

# 定義對資料的預處理
transform = transforms.Compose([
            transforms.ToTensor(), # 轉為 Tensor
            transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)), # 歸一化
                                 ])

# 訓練集
trainset = tv.datasets.CIFAR10(
                           root='/home/cy/tmp/data/',
                           train=True,
                           download=True,
                           transform=transform)

trainloader = t.utils.data.DataLoader(
                           trainset,
                           batch_size=4,
                           shuffle=True,
                           num_workers=2)

# 測試集
testset = tv.datasets.CIFAR10(
                           '/home/cy/tmp/data/',
                           train=False,
                           download=True,
                           transform=transform)

testloader = t.utils.data.DataLoader(
                           testset,
                           batch_size=4,
                           shuffle=False,
                           num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
                 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
Files already downloaded and verified
Files already downloaded and verified
Dataset 物件是一個資料集，可以按下標訪問，返回形如(data, label)的資料。
```

In [44]:

```
(data, label) = trainset[100]
print(classes[label])

# (data + 1) / 2 是為了還原被歸一化的資料
show((data + 1) / 2).resize((100, 100))
ship
```

Out[44]:

In [45]:

```
dataiter = iter(trainloader)
images, labels = dataiter.next() # 返回 4 張圖片及標籤
print(' '.join('%11s'%classes[labels[j]] for j in range(4)))
show(tv.utils.make_grid((images+1)/2)).resize((400,100))
```

```
        bird        deer         cat        bird
```

Out[45]:

Dataloader是一個可反覆運算的物件，它將dataset返回的每一條資料拼接成一個batch，並提供多執行緒加速優化和資料打亂等操作。當程式對dataset的所有資料遍歷完一遍之後，相應的對Dataloader也完成了一次反覆運算。

定義網路

拷貝上面的LeNet網路，修改self.conv1第一個參數為3通道，因CIFAR-10是3通道彩色圖。

```
In [46]:
import torch.nn as nn
import torch.nn.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.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(x.size()[0], -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()
print(net)
Net(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

定義損失函數和優化器(loss和optimizer)

```
In [47]:
from torch import optim
criterion = nn.CrossEntropyLoss() #  交叉熵損失函數
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

訓練網路

所有網路的訓練流程都是類似的，不斷地執行如下流程：

1. 輸入資料
2. 前向傳播+反向傳播
3. 更新參數

```
In [48]:
t.set_num_threads(8)
for epoch in range(2):

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):

        # 輸入資料
        inputs, labels = data

        # 梯度清零
        optimizer.zero_grad()

        # forward + backward
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()

        # 更新參數
        optimizer.step()

        # 列印 log 資訊
        # loss 是一個 scalar,需要使用 loss.item()來獲取數值，不能使用 loss[0]
        running_loss += loss.item()
        if i % 2000 == 1999: # 每 2000 個 batch 列印一下訓練狀態
            print('[%d, %5d] loss: %.3f' \
                  % (epoch+1, i+1, running_loss / 2000))
            running_loss = 0.0
print('Finished Training')
[1,  2000] loss: 2.251
[1,  4000] loss: 1.955
[1,  6000] loss: 1.722
[1,  8000] loss: 1.590
[1, 10000] loss: 1.518
[1, 12000] loss: 1.482
[2,  2000] loss: 1.416
[2,  4000] loss: 1.385
[2,  6000] loss: 1.365
[2,  8000] loss: 1.341
[2, 10000] loss: 1.328
[2, 12000] loss: 1.307
Finished Training
此處僅訓練了 2 個 epoch（遍歷完一遍資料集稱為一個 epoch），
```

來看看網路有沒有效果。將測試圖片輸入到網路中，計算它的Label，然後與實際的Label進行比較。

In [49]:

```
dataiter = iter(testloader)
images, labels = dataiter.next() # 一個 batch 返回 4 張圖片
print('實際的 label: ', ' '.join(\
                    '%08s'%classes[labels[j]] for j in range(4)))
show(tv.utils.make_grid(images / 2 - 0.5)).resize((400,100))
```

實際的 label: cat ship ship plane

接著計算網路預測的 label：

In [50]:

```
# 計算圖片在每個類別上的分數
outputs = net(images)
# 得分最高的那個類
_, predicted = t.max(outputs.data, 1)

print('預測結果: ', ' '.join('%5s'\
                % classes[predicted[j]] for j in range(4)))
```

預測結果: frog car ship ship

已經可以看出效果，準確率 50%，但這只是一部分的圖片，再來看看在整個測試集上的效果。

In [51]:

```
correct = 0 # 預測正確的圖片數
total = 0 # 總共的圖片數


# 由於測試的時候不需要求導，可以暫時關閉 autograd，提高速度，節約記憶體
with t.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = t.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum()

print('10000 張測試集中的準確率為: %d %%' % (100 * correct / total))
```

10000 張測試集中的準確率為: 53 %

訓練的準確率遠比隨機猜測(準確率 10%)好，證明網路確實學到了東西。

在 GPU 訓練

就像之前把 Tensor 從 CPU 轉到 GPU 一樣，模型也可以類似地從 CPU 轉到 GPU。

In [52]:

```
device = t.device("cuda:0" if t.cuda.is_available() else "cpu")

net.to(device)
images = images.to(device)
labels = labels.to(device)
output = net(images)
loss= criterion(output,labels)

loss
```

Out[52]:

```
tensor(1.3732)
```

總結一下

本節主要包含以下內容

1. Tensor: 類似Numpy陣列的資料結構，與Numpy介面類別似，可方便地互相轉換。
2. Autograd/: 為tensor提供自動求導功能。
3. nn: 專門為神經網路設計的介面，提供了很多有用的功能(神經網路層，損失函數，優化器等)。
4. 神經網路訓練: 以CIFAR-10分類為例演示了神經網路的訓練流程，包括資料載入、網路搭建、訓練及測試。
5. 通過本節的學習，相信可以體會出PyTorch具有介面簡單、使用靈活等特點。從下一章開始，將深入系統地講解PyTorch的各部分知識。