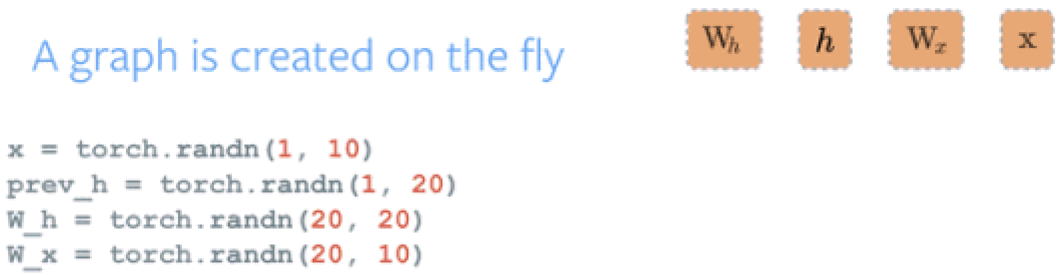
第二章 张量

Tensor，中文叫张量，逻辑上是一个多维数组，类似NumPy的ndarrays，０维对应标量，１维对应向量，２维对应矩阵，其优势在于：

１．并行加速：　Tensor实现了许多并行算法，可用多核CPU和CUDA加速。众所周知，编写高效并行算法，极具挑战，Tensor简化了这项工作。特别地，Tensor建立在ATen库上，源码用C/C++和CUDA实现，效率有保证。

２．自动微分：　深度学习算法的基础是反向传播求导，从0.4版开始，Tensor可设置requries_grad来支持自动微分。



３．数据共享：　Tensor含头部信息和数据存储区。头部保存形状(size)、步长(stride)、数据类型(type)等，数据则保存为一块内存(显存)。一般头部占内存少，数据占内存大，相关数据会尽可能共享存储，是否共享，可用函数 `id()` 验证。

本章尽量用实例来讲解Tensor(全面文档请参阅)。开始之间，让我们导入最重要的模块torch:

```
import torch
print("Torch Version: ", torch.__version__)
```

```
Torch Version:  1.0.0a0+17c6d16
```

2.1.1 表格排序

Torch支持排序，函数原型如下：

sort(input, dim=None, descending=False, out=None) -> (Tensor, LongTensor)

能对输入的张量按给定的维度排序。下面是一个例子：

```
x = torch.rand(3,4)
print(x)
torch.sort(x, dim = 0)
```

```
tensor([[0.6674, 0.6190, 0.7941, 0.9073],
        [0.9862, 0.3700, 0.6737, 0.3410],
        [0.7595, 0.7885, 0.9475, 0.3307]])
```

```
(tensor([[0.6674, 0.3700, 0.6737, 0.3307],
         [0.7595, 0.6190, 0.7941, 0.3410],
         [0.9862, 0.7885, 0.9475, 0.9073]]), tensor([[0, 1, 1, 2],
         [2, 0, 0, 1],
         [1, 2, 2, 0]]))
```

上例每列从小到大，排序正确，但它打乱了每行的联系，在大多数情况下，我们希望保持行的一致性，例如只按第一列排序，步骤如下：

1. 选取第一列排序，得到排序索引；
2. 根据排序索引，按行挑选数据。

```
def sort_table(table, dim, no):
    """sort_table(table, dim, sort_index) --> Tensor ."""

    assert(table.dim() == 2)

    # narrow(input, dimension, start, length) -> Tensor
    n = table.narrow((dim + 1) % 2, no, 1).view(-1)

    # sort(input, dim=None, descending=False, out=None) -> (Tensor, LongTensor)
    _, index = n.sort()

    # index_select(input, dim, index, out=None) -> Tensor
    r = torch.index_select(table, dim, index)

    return r

def test_sort_table():
    x = torch.rand(3, 5)
    r = sort_table(x, 0, 0)
    print("Random Data:")
    print(x)
    print("Sorted Data:")
    print(r)

test_sort_table()
```

```
Random Data:
tensor([[0.2727, 0.6275, 0.5062, 0.4976, 0.3659],
        [0.7815, 0.5055, 0.5457, 0.9314, 0.7082],
        [0.5755, 0.6025, 0.9368, 0.1088, 0.7734]])
Sorted Data:
tensor([[0.2727, 0.6275, 0.5062, 0.4976, 0.3659],
        [0.5755, 0.6025, 0.9368, 0.1088, 0.7734],
        [0.7815, 0.5055, 0.5457, 0.9314, 0.7082]])
```

2.1.2 图像积分

积分图像在Viola的人脸实时识别中发挥了巨大威力，它是一张图像，图像某点的颜色定义为：原始图像原点到该点矩形区内的各点颜色和。通过积分图像，原始图像任意矩形区的颜色之和就可以通过“加减”操作在常数时间内完成，它是快速Box滤波算法的关键。本质上，它就是一个累加矩阵，因此，我们可以使用Tensor实现。

```
def integrate_image():
    a = torch.arange(25).float().view(5, 5)
    print(a)
    b = a.cumsum(dim=0).cumsum(dim=1)
    print(b)

integrate_image()
```

```
tensor([[ 0.,  1.,  2.,  3.,  4.],
        [ 5.,  6.,  7.,  8.,  9.],
        [10., 11., 12., 13., 14.],
        [15., 16., 17., 18., 19.],
        [20., 21., 22., 23., 24.]])
tensor([[  0.,   1.,   3.,   6.,  10.],
        [  5.,  12.,  21.,  32.,  45.],
        [ 15.,  33.,  54.,  78., 105.],
        [ 30.,  64., 102., 144., 190.],
        [ 50., 105., 165., 230., 300.]])
```

2.1.3 线性回归

数学原理

给定数据集$D=\{(x_1,y_1),(x_2,y_2,\ldots,(x_m,y_m)\}$，线性回归希望找到函数f(x)，满足$f(x_i)=wx_i+b$而且$f(x_i)$能够和$y_i$尽可能接近。

如何才能学到参数w和b呢？需要确定如何衡量f(x)与y之差，一般通过损失函数（Loss Funciton)来衡量：

$Loss=\sum_{i=1}^{m}(f(x_i)-y_i)^2$。

这就是著名的均方误差。我们要做的就是找到w^*和b^*，使得：

$(w^*,b^*)=argmin_{w,b}\sum_{i=1}^{m}(f(x_i)-y_i)^2$

$$= argmin_{w,b} \sum_{i=1}^{m} (y_i - wx_i - b)^2$$

均方误差直观，有好的几何意义，对应欧式距离。现在要求解损失函数的最小值，方法是求它的偏导数，让偏导等于0来估计参数，即：

$$\frac{\partial Loss_{(w,b)}}{\partial w} = 2(w \sum_{i=1}^{m} x_i^2 - \sum_{i=1}^{m} (y_i - b)x_i) = 0$$

$$\frac{\partial Loss_{(w,b)}}{\partial b} = 2(mb - \sum_{i=1}^{m} (y_i - wx_i)) = 0$$

求解以上两式，我们就可以得到最优解。幸运的是，对线性函数，我们有最优解析解，不幸的是，对大多数非线性函数，只有数值近似解，下面给出一个用Torch求解的例子。

实际例子

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader

num_inputs = 2
num_outpus = 1
num_examples = 512


true_w = torch.randn(num_inputs, num_outpus)
true_b = torch.randn(num_outpus)

def f(x):
    """f(x) = X*w + b == > Y"""
    return x.mm(true_w) + true_b.item()


def make_feat(x):
    """Builds a matrix with columns [x^4, x^3, x^2, x^1]."""
    x = x.unsqueeze(1)
    return torch.cat([x ** i for i in range(num_inputs, 0, -1)], 1)


def poly_desc(W, b):
    """Creates a string description of a polynomial."""
    result = 'y = '
    for i, w in enumerate(W):
        result += '{:+.4f}x^{} '.format(w, len(W) - i)
    result += '{:+.4f}'.format(b[0])
    return result


def make_data(nums):
    """
    # x = torch.randn(num_examples, num_inputs)
    # y = torch.randn(num_examples, num_outpus)
    """
    x = make_feat(torch.randn(nums))
    y = f(x)
```

```python
    return x, y

# 1.创建数据
x, y = make_data(num_examples)
dataset = TensorDataset(x, y)
trainloader = DataLoader(dataset, batch_size=32, shuffle=True)

# 2.定义模型
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc = nn.Linear(num_inputs, num_outpus)

    def forward(self, x):
        x = self.fc(x)
        return x

net = Net()

# 3. 损失函数
criterion = nn.MSELoss()

# 4. 优化方法
optimizer = optim.SGD(net.parameters(), lr=1e-2)

# 5. 训练模型
net.train()
epochs = 100

for epoch in range(epochs):
    total_loss = 0

    for data in trainloader:
        x, y = data

        # # zero the parameter gradients
        optimizer.zero_grad()

        # forward
        outputs = net(x)
        loss = criterion(outputs, y)

        # backward
        loss.backward()
        optimizer.step()

        total_loss = total_loss + loss.item()

    average_loss = total_loss/num_examples
    print("Epoch %d, average_loss loss: %f" %(epoch, average_loss))
    if average_loss < 0.0001:
        break
```

```
# 6. 验证模型
print('Loss: {:.6f} after {} epoch'.format(loss, epoch))
print('==> Learned function:\t' + poly_desc(net.fc.weight.view(-1), net.fc.bias))
print('==> Actual  function:\t' + poly_desc(true_w.view(-1), true_b))
```

```
Epoch 0, average_loss loss: 0.017373
Epoch 1, average_loss loss: 0.003713
Epoch 2, average_loss loss: 0.001729
Epoch 3, average_loss loss: 0.001107
Epoch 4, average_loss loss: 0.000770
Epoch 5, average_loss loss: 0.000541
Epoch 6, average_loss loss: 0.000379
Epoch 7, average_loss loss: 0.000267
Epoch 8, average_loss loss: 0.000188
Epoch 9, average_loss loss: 0.000132
Epoch 10, average_loss loss: 0.000093
Loss: 0.001778 after 10 epoch
==> Learned function:   y = +0.9773x^2 -0.8348x^1 +0.1609
==> Actual  function:   y = +0.9508x^2 -0.8457x^1 +0.2215
```

这个例子非常简单，但也遵循深度学习的基本套路：创建数据，定义模型，选定损失函数和优化方法，训练模型，验证模型。

注意由**num_inputs**控制多项式的次数，把这个多项式看成核函数，就是机器学习中的核方法。

2.1.4 图像滤波

我们知道，二维卷积就是滤波，一个自然的问题是：**PyTorch**可以代替行色色的传统软件来对图像进行滤波吗？

直接答案是：能！而且代码简洁，运行高效。

首先我们要解决的问题是：图像的读取和图像对象与Tensor间的互转。

```
  1. 安装视觉处理包torchvision，我们使用的PyTorch是最新版本，建议从源码安装，可以避免烦人的包以来，
地址在：https://github.com/pytorch/vision；

  2. 安装图像处理包Pillow，anaconda默认已经安装，如果没有安装，请安装。
```

注意数据格式: 原始PIL图像格式呈现为HxWxC, 取值[0,255], H,W,C分别代表图像高，宽和颜色通道数量; Torch网络接受的输入数据格式为BxCxHxW, 取值[0,1.0], B代表批大小(Batch_Size)。下面给出读取图像和数据转换的代码：

```
from PIL import Image
from torchvision import transforms

import torch
import torch.nn as nn

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

```python
def open(filename):
    return Image.open(filename).convert('RGB')


def to_tensor(image):
    """
    return 1xCxHxW tensor
    """
    transform = transforms.Compose([transforms.ToTensor()])
    t = transform(image)
    return t.unsqueeze(0).to(device)


def from_tensor(tensor):
    """
    tensor format: 1xCxHxW
    """
    transform = transforms.Compose([transforms.ToPILImage()])
    return transform(tensor.squeeze(0).cpu())

img = open("images/roma.jpg")
img.show()
```

2.1.4.1 高斯滤波

下面选用3x3高斯核，注意卷积分组(即groups=3)的设定，为了更容易看出滤波效果，滤波进行了１０次。

```python
class GaussFilter(nn.Module):
    """
    3x3 Guassian filter
    """
```

```python
    def __init__(self):
        super(GaussFilter, self).__init__()
        self.conv = nn.Conv2d(
            3, 3, kernel_size=3, padding=1, groups=3, bias=False)

        # self.conv.bias.data.fill_(0.0)
        self.conv.weight.data.fill_(0.0625)
        self.conv.weight.data[:, :, 0, 1] = 0.125
        self.conv.weight.data[:, :, 1, 0] = 0.125
        self.conv.weight.data[:, :, 1, 2] = 0.125
        self.conv.weight.data[:, :, 2, 1] = 0.125
        self.conv.weight.data[:, :, 1, 1] = 0.25

    def forward(self, x):
        x = self.conv(x)
        return x

def gauss_filter(device, img):
    model = GaussFilter()
    model = model.to(device)
    t = to_tensor(img)
    for i in range(10):
        t = model(t)
        t.detach_()

    return from_tensor(t)


if __name__ == '__main__':
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

    img = open("images/roma.jpg")
    img = gauss_filter(device, img)
    img.show()
```

```
119 ms ± 83.2 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

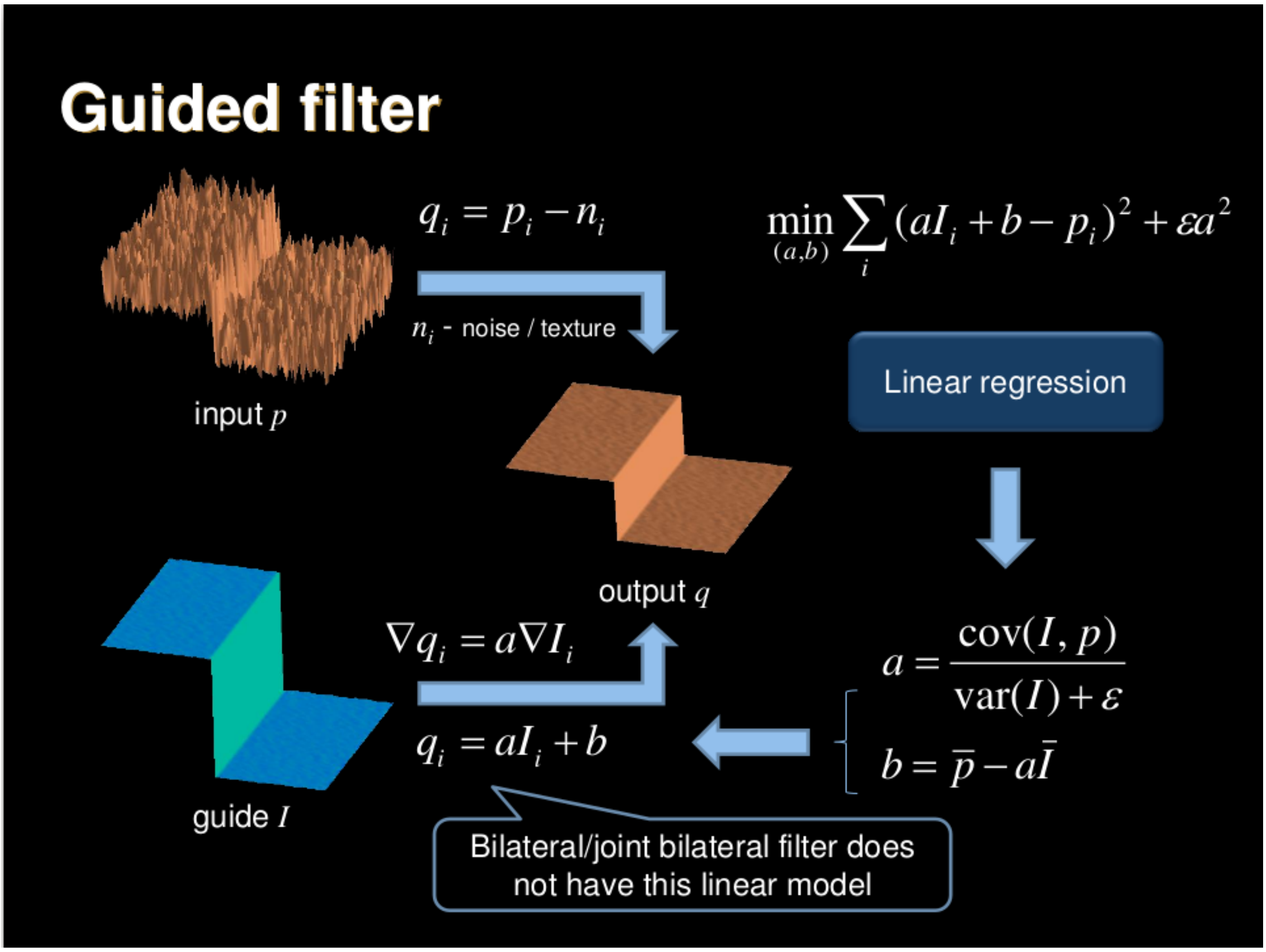
滤波实例

	
原图	Gaussian 3x3 Filter 10 times

选用不同的核，可以实现不同滤波效果，如图像锐化,浮雕,运动模糊等等，也可以实现简易的边缘检测。

2.1.4.2 导向滤波

导向滤波是一种保边滤波，由何凯明博士提出，原理可以简单地用下图描述。



详细参见作者的论文，(http://kaiminghe.com/eccv10/)，限于篇幅，此处从简。下面的实现是一种快速算法。

```
class GuidedFilter(nn.Module):
    """
    Guided filter with r, e
    """
    def __init__(self, r, e):
        super(GuidedFilter, self).__init__()
        self.radius = r
        self.eps = e

    def box_sum(self, mat, r):
        """
            # Ai = Si+r - Si-r-1
            ===> i + r < n, i-r-1 >= 0
            ===> [0, r + 1), [r + 1, n - r), [n - r, n]
        """
        height, width = mat.size(0), mat.size(1)
        assert 2 * r + 1 <= height
        assert 2 * r + 1 <= width

        dmat = torch.zeros_like(mat)

        mat = torch.cumsum(mat, dim=0)
        dmat[0:r + 1, :] = mat[r:2 * r + 1, :]
        dmat[r + 1:height -
             r, :] = mat[2 * r + 1:height, :] - mat[0:height - 2 * r - 1, :]
        for i in range(height - r, height):
            dmat[i, :] = mat[height - 1, :] - mat[i - r - 1, :]

        dmat = torch.cumsum(dmat, dim=1)
        mat[:, 0:r + 1] = dmat[:, r:2 * r + 1]
        mat[:, r + 1:width -
            r] = dmat[:, 2 * r + 1:width] - dmat[:, 0:width - 2 * r - 1]
        for j in range(width - r, width):
            mat[:, j] = dmat[:, width - 1] - dmat[:, j - r - 1]
        return mat

    def box_filter(self, x, N):
        """
        x format is 1xCxHxW, here C = 3
        """
        y = torch.zeros_like(x)
        for i in range(x.size(1)):
            y[0][i] = self.box_sum(x[0][i], self.radius).div(N)
        return y

    def forward(self, i, p):
        N = torch.ones_like(i[0][0])
        N = self.box_sum(N, self.radius)
        mean_i = self.box_filter(i, N)
        mean_p = self.box_filter(p, N)
```

```
        mean_pi = self.box_filter(p * i, N)
        mean_ii = self.box_filter(i * i, N)

        cov_ip = mean_pi - mean_p * mean_i
        cov_ii = mean_ii - mean_i * mean_i

        a = cov_ip / (cov_ii + self.eps)
        b = mean_p - a * mean_i

        q = a * i + b
        q.clamp_(0, 1)

        return q

    def self_guided(self, p):
        N = torch.ones_like(p[0][0])
        N = self.box_sum(N, self.radius)
        mean_p = self.box_filter(p, N)
        mean_pp = self.box_filter(p * p, N)

        cov_pp = mean_pp - mean_p * mean_p

        a = cov_pp / (cov_pp + self.eps)
        b = mean_p - a * mean_p

        q = a * p + b
        q.clamp_(0, 1)

        return q

def guided_filter(device, i, p, r=3, e=0.01):
    model = GuidedFilter(r, e)
    model.to(device)
    ti = to_tensor(i)
    tp = to_tensor(p)
    t = model(ti, tp)

    return from_tensor(t)


def self_guided_filter(device, img, r=5, e=0.01):
    model = GuidedFilter(r, e)
    model.to(device)

    t = to_tensor(img)
    t = model.self_guided(t)

    return from_tensor(t)


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

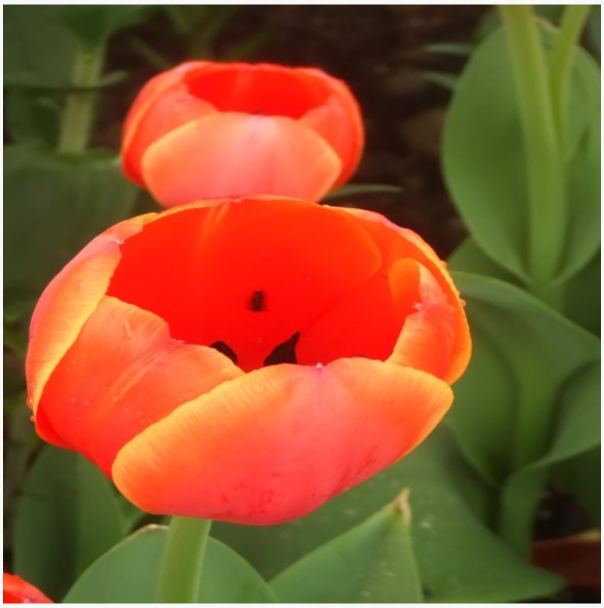
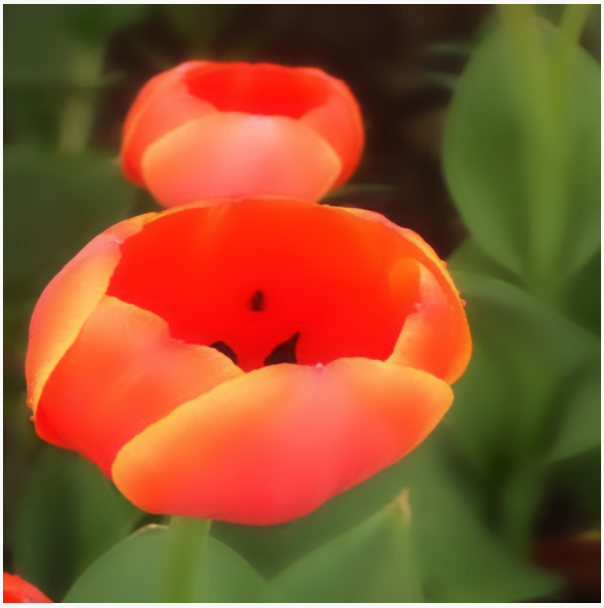
```
    img = open("images/guided_girl.jpg")
    img = self_guided_filter(device, img, 10, 0.01)

    img.show()
```

美容实例

	
原图	Guided Filter: Radius = 10, eps = 0.01

增强实例

	
原图	Guided Filter: Radius = 10, eps = 0.01

2.1.4.3 图像去雾

下面的去雾算法，基于暗通道先验，同样由何凯明博士发明。

```python
class DehazeFilter(nn.Module):
    """
    Dehaze filter with r
    """
    def __init__(self, r=7):
        super(DehazeFilter, self).__init__()
        self.radius = r
        self.maxpool = nn.MaxPool2d(2 * r + 1, stride=1, padding=r)

    def min_filter(self, x):
        """
        suppose x is : HxW, y ==> 1x1xHxW
        """
        y = x.unsqueeze(0).unsqueeze(0)
        y = y * (-1.0)
        y = self.maxpool(y)
        y = y * (-1.0)
        return y.squeeze(0).squeeze(0)

    def dark_channel(self, x):
        rgb = x[0]
        dc, _ = torch.min(rgb, dim=0)
        # dc size: HxW
        dc = self.min_filter(dc)
        return dc

    def atmos_light(self, dc, x):
        # dc -- HxW
        sorted, _ = dc.view(-1).sort(descending=True)
        index = int(dc.size(0) * dc.size(1) / 1000)
        thres = sorted[index].item()
        mask = dc.ge(thres)

        a = torch.zeros(3)
        for i in range(3):
            rgb = x[0][i]
            dx = torch.masked_select(rgb, mask)
            a[i] = torch.mean(dx)

        # RGB atmos light
        avg = 0.299 * a[0].item() + 0.587 * a[1].item() + 0.114 * a[2].item()
        a[0] = a[1] = a[2] = avg

        return a[0].item(), a[1].item(), a[2].item()

    def forward(self, x):
        """
        I = J*t + A*(1-t), here I = x, target is J
        t = 1.0 - omega*min_filter(Ic/Ac) for c = R, G, B, here w = 0.95
        J = (Ic - Ac)/t + Ac
        """
        omega = 0.95
```

```
        dc = self.dark_channel(x)

        # atmos light
        a_r, a_g, a_b = self.atmos_light(dc, x)

        # t -- from 1x3xHxW--> HxW
        t = torch.zeros_like(x)
        t[0][0] = x[0][0] / a_r
        t[0][1] = x[0][1] / a_g
        t[0][2] = x[0][2] / a_b
        t = self.dark_channel(t)
        t = 1 - omega * t

        refined_t = torch.zeros_like(x)
        refined_t[0][0] = t
        refined_t[0][1] = t
        refined_t[0][2] = t

        model = GuidedFilter(60, 0.0001)
        model.to(device)
        refined_t = model(x, refined_t)

        refined_t.clamp_(min=0.1)

        y = torch.zeros_like(x)
        y[0][0] = (x[0][0] - a_r) / refined_t[0][0] + a_r
        y[0][1] = (x[0][1] - a_g) / refined_t[0][1] + a_g
        y[0][2] = (x[0][2] - a_b) / refined_t[0][2] + a_b
        y.clamp_(0, 1)

        return y

def dehaze_filter(device, img, r=3):
    model = DehazeFilter(r)
    model.to(device)
    t = to_tensor(img)
    t = model(t)
    return from_tensor(t)


if __name__ == '__main__':
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

    img = open("images/haze.jpg")
    img = dehaze_filter(device, img, 5)
    img.show()
```

去雾实例

	
原图	Haze Filter: Radius = 5

2.1.4.4 完整资源

https://github.com/delldu/BeyondFilter

2.1.5 自动求导

计算图是一种有向无环图(**DAG**)，用于记录算子与变量间的关系，一般用矩形表示算子，椭圆表示变量。它是现代深度学习框架的核心，为高效自动求导算法—反向传播(Back Propogation)提供支持。理论上有了计算图，要计算各节点的梯度，只需从根节点出发，自动沿计算图反向传播，就能计算出每个叶节点的梯度。但是，手动实现反向传播，费时费力，容易出错。为此，PyTorch专门开发出自动求导引擎torch.autograd。在前向传播中，autograd会记录当前Tensor的所有操作，并建立计算图。在反向传播中，autograd沿着这个图从根节点回溯，利用链式求导法则计算叶节点的梯度。每个前向传播操作的函数都有对应的反向传播函数，用来计算节点的梯度，这些函数名通常以Backward结尾。

实例

```
def test_autograd():
    x = torch.ones(3, 3, requires_grad=True)
    y = x.sum()

    y.backward()
    print(x.grad)

    x.grad.data.zero_()

test_autograd()
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])
```

注意：

1. grad在反向传播过程中是累加的，即每次运行反向传播，梯度都会累加之前的梯度，所以反向传播前应把梯度清零;
2. 具备requires_grad的Tensor不支持部分in-place函数，因为这些函数会修改Tensor自身，而在反向传播中，Tensor需要缓存原来的Tensor来计算反向梯度。

2.1.6 向量化思维

向量化计算是一种特殊的并行计算，相对于一般程序同时只执行一个操作的方式，可同时执行多个操作，通常是对不同数据执行同样的一个或一批指令，或者说把指令应用于一个向量上。

Python是一门高级语言，使用方便，但很多操作低效，尤其是for循环，应尽量调用内建函数(buildin-function)，这些函数底层由C/C++实现，能通过底层优化。我们平常编写代码应养成向量化的思维习惯，避免对较大的Tensor逐元遍历，对PyTorch没有的功能，如果性能非常关键，应通过C/C++扩展完成。