深度学习入门

重磅！ 2020年最新计算机视觉学习路线教程

https://course.fast.ai/

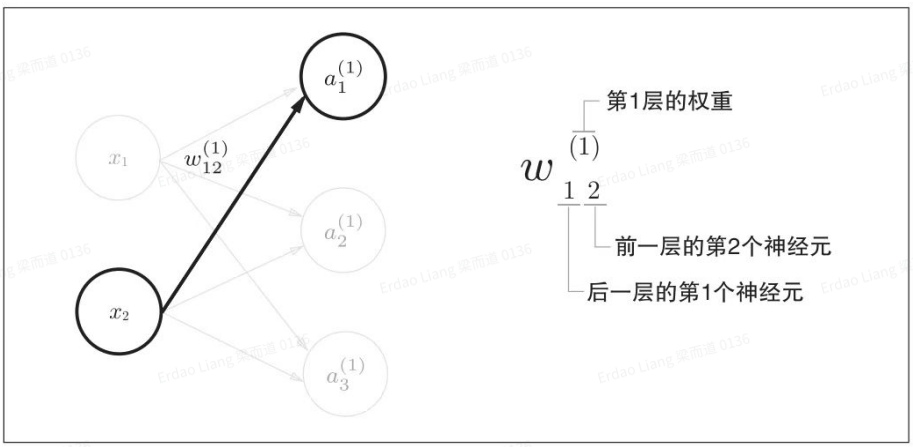
深度学习理论

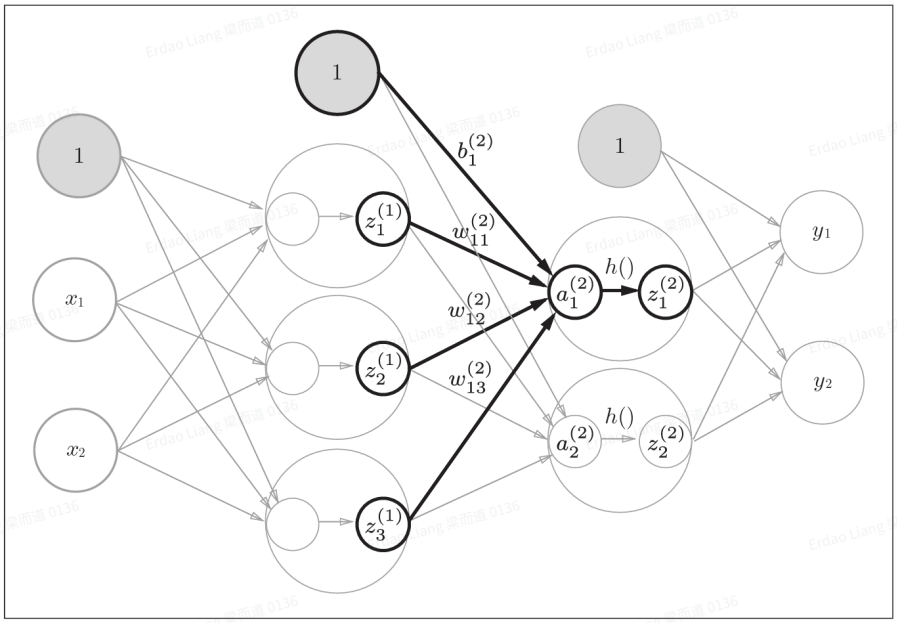
1 神经网络

- 起源于一种叫感知机（Perception）的东西
- 神经网络的动机：感知机的问题 - 设定权重的工作由人工进行
- 目标：神经网络的实现 = 参数学习（**反向传播**，backward propagation） + 推理（**前向传播**，forward propagation）

神经网络的结构

- 神经网络的基本结构：输入层，输出层，中间层（隐藏层）
  - 输入层和输出层只有一个，隐藏层可以有很多层
- 神经网络的层数=隐藏层数+1（输入层记作第0层，第一层隐藏层记作第1层）
- 权重的表示方式：右下角按照“后一层索引号、前一层索引号”的顺序排列

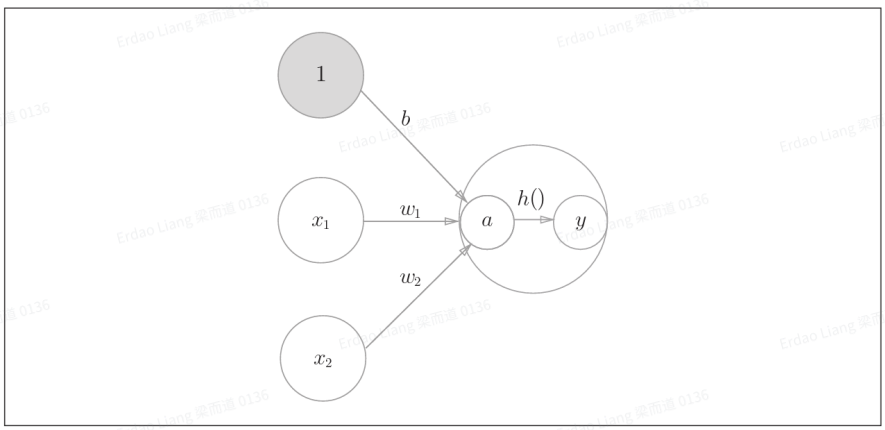




激活函数

深度学习笔记：如何理解激活函数？（附常用激活函数）

- 动机 - 神经网络的激活函数必须使用非线性函数，否则加深层数没有意义，因为总是存在与之等效的“无隐藏层的神经网络”
  - （在神经网络的计算过程中每层都相当于矩阵相乘，无论神经网络有多少层输出都是输入的线性组合，就算我们有几千层的计算，无非还是个矩阵相乘，总是等效于和一个矩阵相乘）
- 常见激活函数
  - h(x) = h(b+w1x1+w2x2+...)= y，转换加权输入信号和偏置的总和



  - Sigmoid 函数：$h(x) = \frac{1}{1+e^{-x}}$，具有平滑性
  - ReLU 函数 (Rectified Linear Unit)：$h(x) = max(0, x)$
  - Softmax 函数
  - Tanh 函数：$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- 激活函数的选择
  - 激活函数的具体优劣比较 pytorchbook.cn_3

- Sigmoid 函数已经很少被使用
- 隐藏层：一般用ReLU函数
- 输出层
  - 回归问题：恒等函数
  - 分类问题：softmax函数 $y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n}\exp(a_i)}$（输出值0-1，解释为概率；把输出值最大的神经元所对应的类别作为识别结果）

神经网络的的学习过程

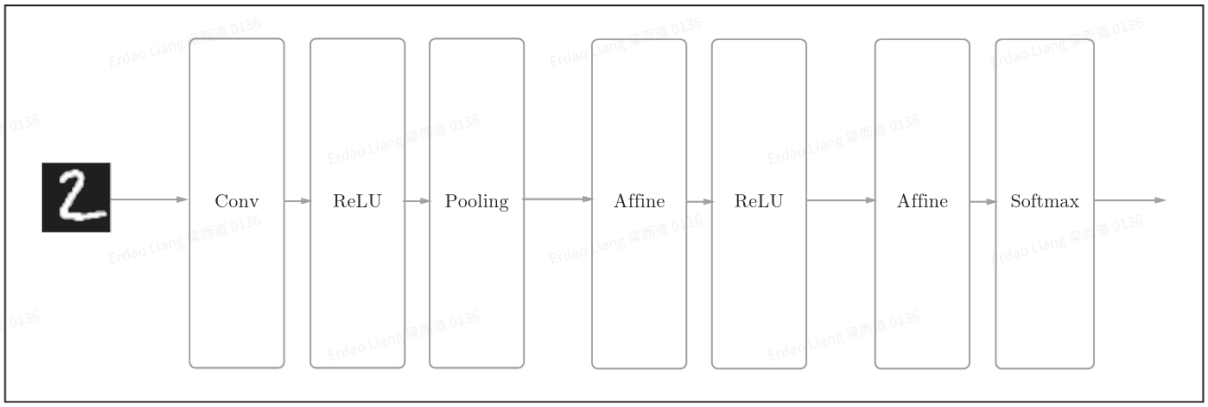
- 神经网络算法的**特征量**由机器学习（传统机器学习算法由人工提取特征量）
- 学习算法的逻辑：
  - mini-batch：从训练数据中随机选出一部分
  - 计算梯度：以损失函数为指标（目标是使得损失函数的值尽可能小），通过微分求出各个权重参数的梯度，梯度表示损失函数的值减少最多的方向
  - 更新参数：将权重参数沿梯度方向进行微小更新
  - 重复上述步骤
- 损失函数
  - 均方误差（mean squared error） $E = \frac{1}{2}\sum_k (y_k - t_k)^2$
  - 交叉熵误差（cross entropy error） $E = -\sum_k t_k \log y_k$
  - 不能用识别精度作为损失函数，否则参数的导数在大多数地方都会变为0
- 梯度计算：误差反向传播法
  - 计算图与反向传播（略）
    - 通过计算图集中精力于局部计算
  - 各层的反向传播实现

2 卷积神经网络 - 计算机视觉

- 动机：传统神经网络为**全连接**（相邻层每个神经元之间都有连接），在处理图像时，数据的形状被忽视了
  - 卷积计算可以减少重复的参数，提取局部特征（图像的平移不变性）
- 由Yann LeCun在1989年提出的LeNet中首先被使用，在手写数字识别中取得成功

CNN 网络的一般结构

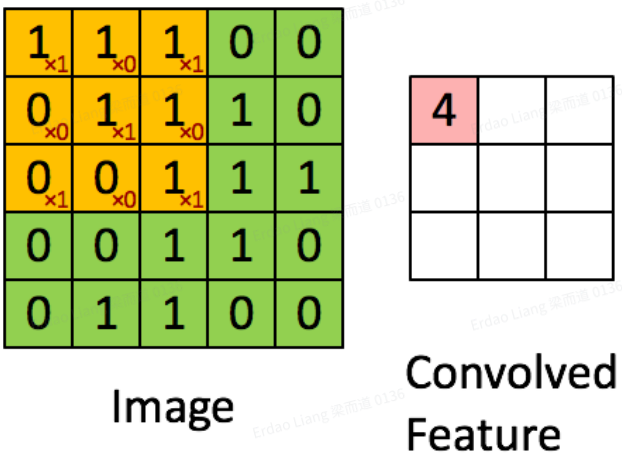
TODO 绘制示意图



- input_dim—输入数据的维度：（通道，高，长）
- conv_param—卷积层的超参数（字典）。字典的关键字如下：
  - filter_num—滤波器的数量
  - filter_size—滤波器的大小
  - stride—步幅
  - pad—填充
- hidden_size—隐藏层（全连接）的神经元数量
- output_size—输出层（全连接）的神经元数量
- weitght_int_std—初始化时权重的标准差

卷积层 Convolution Layer

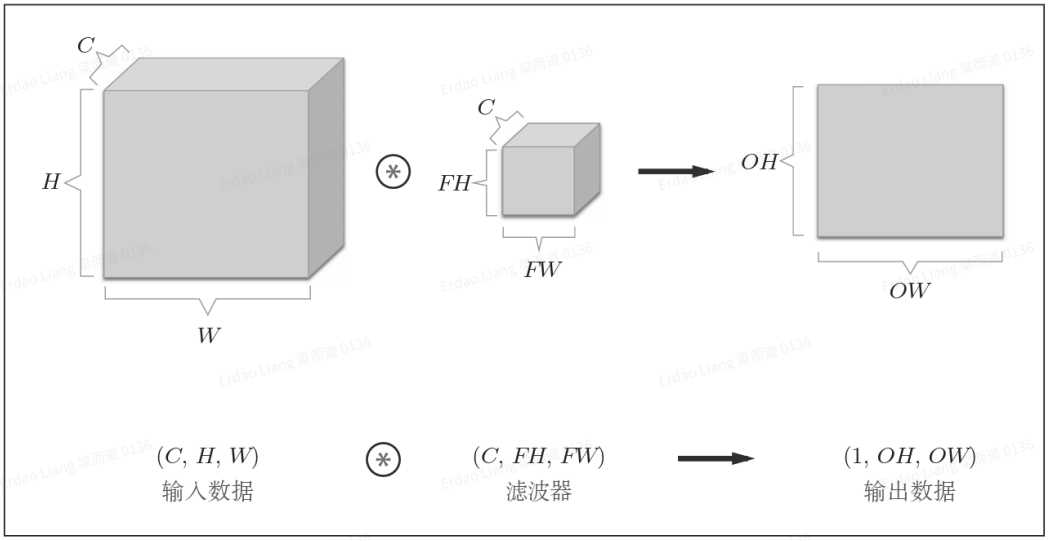
- 卷积计算：对输入数据应用权重矩阵（或称为 滤波器 filter）（或称为 核 kernel）
  - 在输入矩阵上使用权重矩阵进行滑动，每滑动一步，将所覆盖的值与矩阵对应的值相乘，并将结果求和并作为输出矩阵的一项，依次类推直到全部计算完成



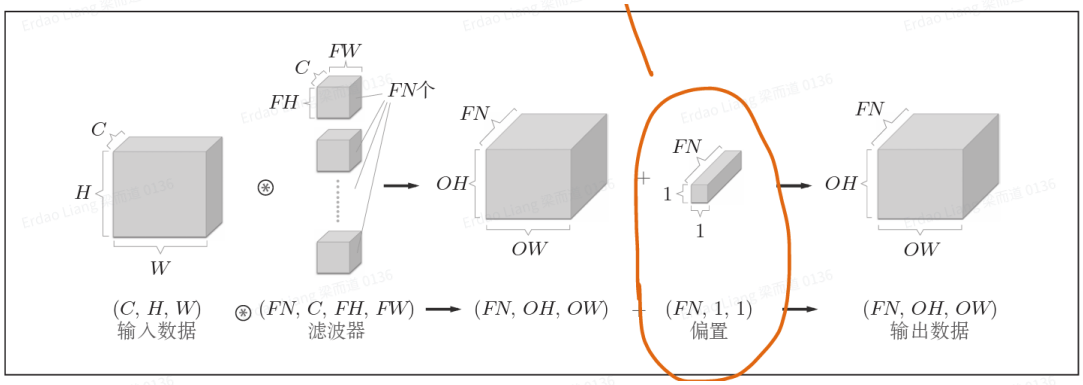
- 卷积层的参数和输出矩阵的大小：

- **卷积核大小**（滤波窗口大小）
- 通过**填充**（padding，向数据周围一圈填入0）以调整输出大小
- 调整**步幅**（stride，应用滤波器的窗口的间隔大小）
- 输出高度(宽度) $= floor(\frac{\text{输入高度(宽度)} + 2 \times \text{填充} - \text{滤波窗口高度(宽度)}}{\text{步幅}} + 1)$

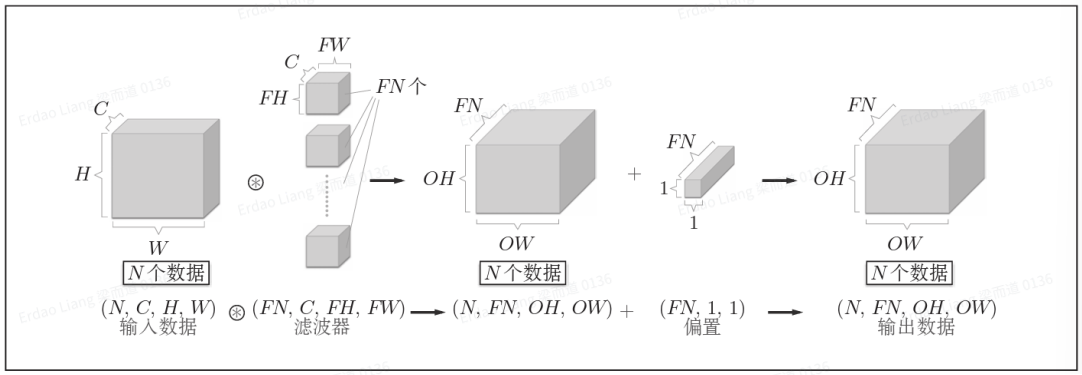
- 卷积操作也是线性的，也需要在以后加入激活函数
- 卷积运算的处理流：



- 输出多个通道：

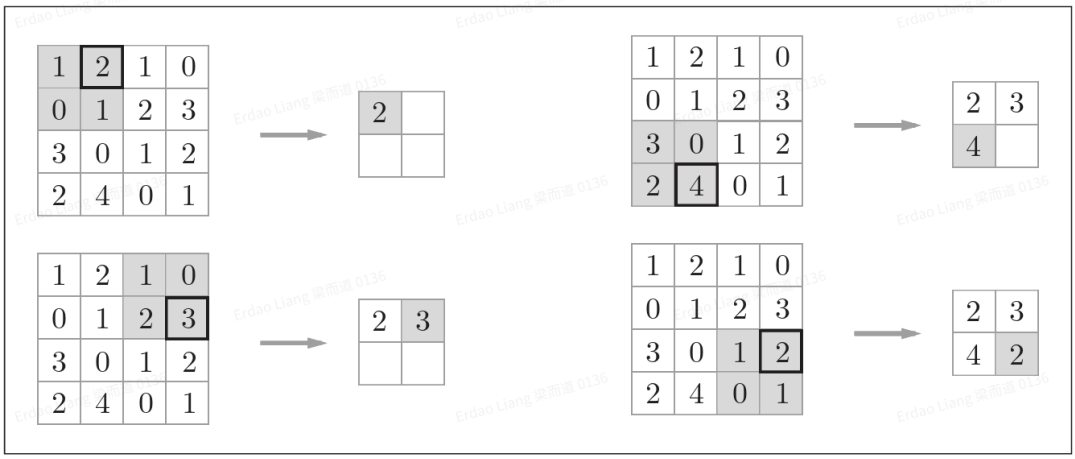


- 卷积运算的处理流（批处理）：



池化层 Pooling Layer

- 池化运算用于缩小高、长方向上的空间大小
- 池化运算：
  - 输入一个过滤器的大小，与卷积的操作一样，也是一步一步滑动，但是过滤器覆盖的区域进行合并，只保留一个值；合并的常见方式有取平均（AvgPool）、取最大值（MaxPool）或取最小值（MinPool）等



- 池化层的的特征
  - 没有要学习的参数
  - 池化窗口大小一般和步幅设定成相同值
  - 对输入数据的微小变化具有鲁棒性
- 在最后加入几层全连接层

代表性 CNN

- LeNet-5，1998，手写数字识别（CNN元祖）
- AlexNet，2012，出现在视觉识别比赛中，引发深度学习热潮的导火索
- VGG，2015；使用更小的卷积核，每次的图像像素缩小一倍，卷积核的数量增加一倍
- GoogLeNet，2014；每一模块都是用若干个不同的特征提取方式，最后再把这些结果通过Filter Concat来进行连接，找到这里面作用最大的

3 循环神经网络 - 语言建模

- 关键词：

$$h_t = \tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{(t-1)} + b_{hh})$$

神经网络的训练实践

4 附录：数学运算和编程

矩阵/多维数组的运算

- 具有 A=XW+B 形式的运算成为**仿射变换**
- 矩阵的乘积运算，可以这样检查维度大小的一致性

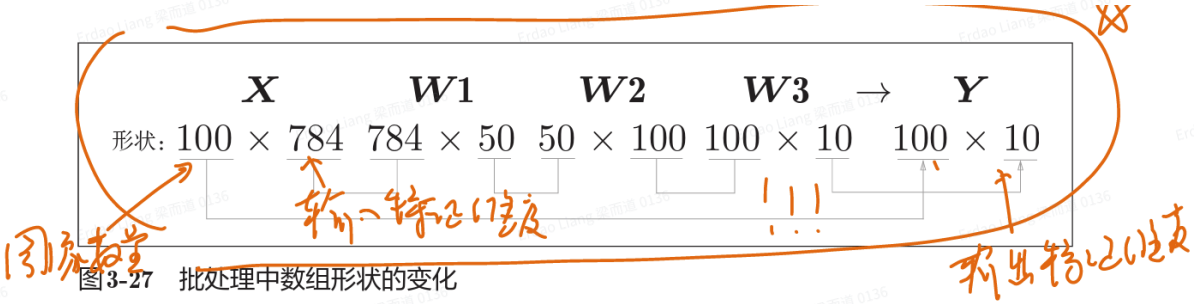


图 3-27　批处理中数组形状的变化

深度学习框架：PyTorch

0 深度学习框架对比

TODO

https://pytorchbook.cn/ PyTorch 中文手册

https://pytorch-cn.readthedocs.io/ PyTorch 中文文档

PyTorch is an open source machine learning library for Python, based on Torch, used for applications such as natural language processing. It is primarily developed by Facebook's artificial-intelligence research group.

- 作为NumPy的替代品，可以使用GPU的强大计算能力
- 提供最大的灵活性和高速的深度学习研究平台

深度学习平台的发展历史：Caffe - TensorFlow (Google) - Kerus - TensorFlow 2.x - (2019) PyTorch (Facebook)

1 前置工具

cs231n.github.io

1.1 Numpy

- 创建数组

```
a = np.array([1,2,3])
a = np.array([[1,2],[3,4]])
a = np.zeros((2,2)) # 参数为size
a = np.ones()
a = np.random.random((2,2))
```

- 数组索引

```
b = a[1:,1:]
b.shape # 查询形状
# 整数数组索引：能够从一个数组中任意提取数据构建另一个数组
a = np.array([[1,2], [3, 4], [5, 6]])
print(a[[0, 1, 2], [0, 1, 0]])  # Prints "[1 4 5]"
# 也就是用了原数组的 (0,0), (1,1), (2,0)
# 布尔值索引：用于选出一个数组中符合某个条件的元素
a = np.array([[1,2], [3, 4], [5, 6]])
bool_idx = (a>2)
a[bool_idx] # [3,4,5,6]
```

- 数组运算
  - Basic operation + - * /: operate elementwise
  - Vector and matrix operation

```
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])
v = np.array([9,10])
w = np.array([11, 12])
# Inner product / matrix & vector product / matrix and matrix product
res1 = np.dot(v,w) # function in the numpy module
res1 = v.dot(w)
res2 = np.dot(x,y)
```

- 广播机制

a. If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.

b. The two arrays are said to be *compatible* in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.

c. The arrays can be broadcast together if they are compatible in all dimensions.

d. After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.

e. In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension

1.2 SciPy

```
# read images
from scipy.misc import imread, imsave, imresize
img = imread('local destination')
```

1.3 Matplotlib

- Plotting

```
from matplotlib import pyplot
x = ...
y = ...
plt.plot(x,y)
plt.show()
plt.xlabel()
plt.title()
plt.legend()
```

- Subplotting

```
plt.subplot(2,1,1) # 设置一个子图网格，高2，宽1，激活第1张子图
plt.plot(x,y1)
plt.subplot(2,1,2) # 激活第2张子图
plt.plot(x,y2)
plt.show()
```

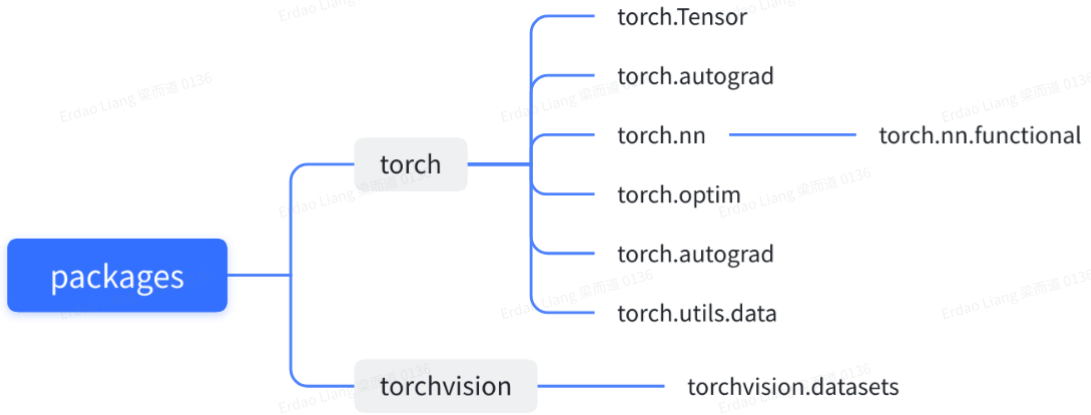
- Showing images

```
img = imread('local destination') # in scipy
plt.imshow(np.uint8(img))
```

```
\plt.show()
```

2 PyTorch 包和函数简要用法

PyTorch中常用packages的结构：



2.1 torch.Tensor - 定义张量

- 定义：类似于numpy的ndarray，主要区别在于tensor可以在gpu上运行

```
x = torch.empty(row, column)
x = torch.rand(row, column)
x = torch.ones(row, column)
x = torch.zeros(row, column)
x = torch.tensor([1,2]) # 根据数组创建张量
```

  - **.size()** 或 **.shape**
  - 对于标量，可以用 .item() 提取出数值
  - **.view()** 返回一个有相同数据但大小不同的tensor。 返回的tensor必须有与原tensor相同的数据和相同数目的元素

- 计算

```
x + y # 加法
torch.add(x,y)
y.add_(x) # 加法
```

- NumPy 与 PyTorch 转换

```
# tensor 转 NumPy
b = a.numpy()
# NumPy 转 tensor
b = torch.from_numpy(a)
```

- 如果设置 `.requires_grad` 为 `True`，那么将会追踪所有对于该张量的操作。 当完成计算后通过调用 `.backward()`，自动计算所有的梯度， 这个张量的所有梯度将会**自动积累到 `.grad` 属性**
  - 比如我们依次执行：x = [[1,1],[1,1]], y=x+2, z = y*y*3, out = z.mean() （z的结果为27）
  - 然后执行 out.backward()，然后x.grad会输出d(out)/x.grad的值**（？？待验证）**

2.2 torch.nn - 构建神经网络

容器

nn.Module 所有网络的基类。你的模型也应该继承这个类

- `.parameters()` 返回一个 包含模型所有参数 的迭代器。
- `.named_parameters()` 同时返回可学习的参数及名称
- `.zero_grad()` 将module中的所有模型参数的梯度设置为0.

隐藏层

隐藏层	用法
卷积层	**Conv2d(in_channels, out_channels, kernel_size)** • 一维卷积层，输入尺度(number_of_data, channel_in, height, width) • Shape: ◦ 输出尺度(number_of_data, channel_out, height_out, width_out) ◦ $H_{out} = floor((H_{in} + 2padding[0] - dilation[0](kernerl_size[0] - 1) - 1)/stride[0] + 1)$ • 参数: ◦ in_channels(`int`) – 输入信号的通道 ◦ out_channels(`int`) – 卷积产生的通道 ◦ kerner_size(`int` or `tuple`) - 卷积核的尺寸 ◦ stride(`int` or `tuple` , `optional`) - 卷积步长 ◦ padding (`int` or `tuple` , `optional`)- 输入的每一条边补充0的层数

池化层	**MaxPool2d(kernel_size)**（还有最大值池化、平均值池化等） • Shape： ◦ 输入: (N,C,H_{in},W_in) ◦ 输出: (N,C,H_out,W_out) $H_{out} = floor((H_{in} + 2padding[0] - dilation[0](kernel_size[0] - 1) - 1)/stride[0] + 1$ • 参数 ◦ kernel_size(`int` or `tuple`) - max pooling的窗口大小 ◦ stride(`int` or `tuple` , `optional`) - max pooling的窗口移动的步长。默认值是 `kernel_size` ◦ padding(`int` or `tuple` , `optional`) - 输入的每一条边补充0的层数
线性层	**torch.nn.Linear (in_features, out_features, bias=True)** 对输入数据做线性变换 y=Wx+b • Shape： ◦ 输入: (N,in_features) ◦ 输出： (N,out_features) • 参数： ◦ in_features - 每个输入样本的大小 ◦ out_features - 每个输出样本的大小

激活函数

- nn.ReLU(), nn.Softmax(), nn.Sigmoid() 等
- Shape：这些函数的输入和输出shape都应该相同

损失函数

- nn.MSELoss() 均方误差

$$loss(x, y) = \frac{1}{n}\sum(x_i - y_i)^2$$

- nn.CrossEntropyLoss() 交叉熵误差

$$loss(x, class) = -\log\frac{\exp(x[class])}{\sum_j \exp(x[j])}$$

2.3 torch.autograd - 自动求导

- 用法

Autograd模块实现了深度学习的算法中的向传播求导数，在张量（Tensor类）上的所有操作，Autograd都能为他们自动提供微分，简化了手动计算导数的复杂过程。

Variable类中的的**grad**和**grad_fn**属性已经整合进入了Tensor类中。只需要设置**.requires_grad=True**，PyTorch会自动追踪和记录对与张量的所有操作，当计算完成后调用.backward()方法自动计算梯度并且将计算结果保存到grad属性中。

在张量进行操作后，grad_fn已经被赋予了一个新的函数，这个函数引用了一个创建了这个Tensor类的Function对象。

如果Tensor类表示的是一个标量（即它包含一个元素的张量），则不需要为backward()指定任何参数，但是如果它有更多的元素，则需要指定一个gradient参数，它是形状匹配的张量

我们可以使用with torch.no_grad()上下文管理器临时禁止对已设置requires_grad=True的张量进行自动求导

- 自动求导原理 pytorchbook.cn

2.4 torch.optim- 实现优化算法

- 用法：
  - 构建：要给一个包含了需要优化的参数 optimizer = optim.SGD(**model.parameters()**, lr = 0.01, momentum=0.9)
  - 单次优化：

```
for input, target in dataset:
    optimizer.zero_grad() # 先梯度清零
    # 和net.zero_grad() 效果相同
    output = model(input)
    loss = loss_fn(output, target)
    loss.backward()
    optimizer.step()
```

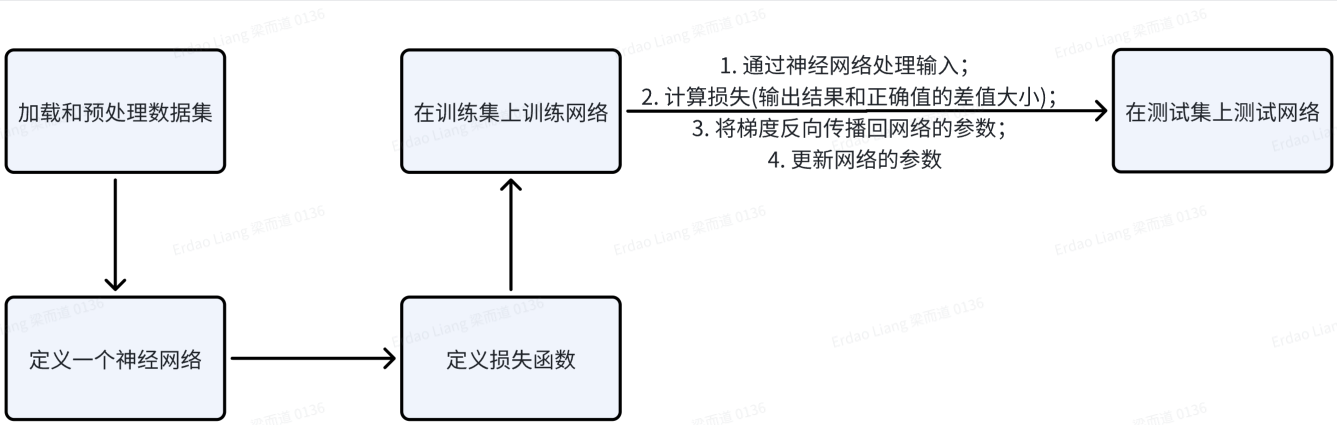
- 算法
  - **.SGD(params, lr=, momentum=0)** 随机梯度下降算法
    - params (iterable) – 待优化参数的iterable或者是定义了参数组的dict（通常是网络的.parameters()）
    - lr (float) – 学习率
    - momentum (float, 可选) – 动量因子（默认：0）

2.5 torchvision.datasets - 获取常用图像数据集

- Torchvision 是 PyTorch 中专门用来处理图像的库
- Datasets 包括了CIFAR-10、ImageNet、MNIST等
  - **torch.utils.data.CIFAR10()**
- 数据读取（例子见训练网络实例）
  - torch.utils.data.Dataloader()
    - batch_size(每个batch的大小)
    - shuffle(是否进行shuffle操作)
    - num_workers(加载数据的时候使用几个子进程)

3 用 PyTorch 训练神经网络流程

训练一个神经网络模型的过程



数据加载和预处理

- 从torchvision.datasets中获取数据集
- 用dataloader加载数据

```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=0)
```

```
    testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                           download=True, transform=transform)
    testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                             shuffle=False, num_workers=0)

    classes = ('plane', 'car', 'bird', 'cat',
               'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

- 数据预处理 *TODO*

定义神经网络

- 利用nn.Module创建一个类，类成员包含网络各层
  - 要自定义一个forward前向传播函数（输入和输出都应该是tensor）
  - backward函数会被自动创建
  - 在卷积层中，特征都是使用矩阵表示的，所以再传入全连接层之前还需要对特征进行压扁，将他这些特征变成一维的向量

```
    import torch
    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.pool = nn.MaxPool2d(2,2)
            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 = self.pool(F.relu(self.conv1(x)))
            x = self.pool(F.relu(self.conv2(x)))
            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

    net = Net()
    print(net)
```

定义损失函数和优化器

- 定义损失函数和优化器

```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

训练网络

- 第一层for指定了做几次训练
- 第二层for指定了训练中的迭代次数
  - for中的optimizer.step() 做单次迭代

```
for epoch in range(2): # 多批次循环

    running_loss = 0.0

    for i, data in enumerate(trainloader, 0):
        # 获取输入
        inputs, labels = data
        # 梯度置零
        optimizer.zero_grad()
        # 正向传播，反向传播，优化
        outputs = net(inputs)
        loss = criterion(outputs, labels) # 这里的criterion是交叉熵误差
        loss.backward()
        optimizer.step()
        #状态信息
        running_loss += loss.item()
        if i % 2000 == 1999:
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0
```

测试

4 附录零散笔记

AttributeError: '_MultiProcessingDataLoaderIter' object has no attribute 'next'

Python 补充

Python super init (继承)

Python __call__

Python with 语句

Pandas