深度学习入门（1）

前言 3

一、Python 基础知识 3

1.1 class 和 function 4

1.2 Numpy 4

1.2.1 Ndarray 4

1.2.2 广播 5

1.2.3 其他知识点 5

1.3 matplotlib 和 skimage 6

1.4 python 序列化 6

1.5 实践：mnist 数据集解析 6

二、Percetron（感知机） 8

2.1 感知机简介 8

2.2 简单的逻辑门的实现 9

2.3 多层的感知机 10

2.4 总结 11

三、神经网络的 forward 实现 12

3.1 激活函数的实现 12

3.1.1 Sigmoid 函数的实现 13

3.1.2 阶跃函数 13

3.1.3 Relu 系列 14

3.2 forward 的流程 14

3.3 输出层的设计与实现 14

3.4 测试神经网络的推理 16

四、神经网络的 backward 实现 16

4.1 损失函数 17

均方误差（mean squared error） 17

交叉熵误差（cross entropy error） 17

4.2 梯度 17

4.2.1 实现梯度的计算 18

4.2.2 测试梯度计算方法是否有用 18

4.2.3 梯度法的实现 19

4.3 学习算法的实现 19

4.4 基于 mnist 数据集的神经网络的训练 20

五、计算图 23

5.1 简单层的实现 23
5.1.1 乘法层的实现 23
5.1.2 加法层的实现 24
5.2 激活层的实现 24
5.2.1 ReLU 层 24
5.2.2 Sigmoid 层 25
5.3 Affine 层 25
5.5 Softmax-with-Loss 层 26
5.6 基于计算图构建的神经网络 27
5.7 使用 mnist 数据集测试 28
六 optimization 的实现 29
6.1 SGD 的实现 29
6.2 Momentum 的实现 31
6.3 AdaGrad 32
6.5 RMSProp 的实现 33
6.4 adam 的实现 33
6.5 构造神经网络 34
6.6 opt 的对比（基于 mnist 数据集） 36
4.6 总结 37
七、解决过拟合的方法 37
7.1 DropOut 的实现 37
7.2 权值衰减的实现 38
八、Batch Normalization 40
~~3.1 观察每层的数值分布~~ 40
3.2 batch normalization 的实现 40
3.3 测试 batch normalization(基于 mnist 数据集) 42
九、卷积层的实现 43
9.1 卷积层的运算 43
卷积的 forward 43
卷积的 backward 44
9.2 引入 im2col 概念 44
9.3 单元测试 im2col 45
9.3 卷积操作的实现 47
9.4 单元测试卷积操作 48
十、池化层的实现 49
10.1 池化层的运算 49

池化层的 forward.....49
池化层的 backward 的运算.....50
10.2 池化层的实现.....50
10.3 pool 单元测试.....51
十一、卷积神经网络.....52
11.1 卷积神经网络的实现.....52
11.2 神经网络的可视化.....53
11.3 深度神经网络.....53

前言

最近由于疫情被困在家，于是准备每天看点专业知识，准备写成博客，不定期发布。

博客大概会写 5~7 篇，**主要是“解剖”一些深度学习的底层技术**。关于深度学习，计算机专业的人多少都会了解，知道 Conv\Pool 的过程，也看过论文，做过实验或是解决方案。在写的各种卷积网路时候，有没有问问自己：这些网络到底是怎么“运作”起来的？如果自己要实现一个具备基本功能的神经网络应该怎么去实现？

知道事物的表面现象，不知事物的本质及其产生的原因是一件很可悲的事情，正如鲁迅所说：What I cannot create, I do not understand. 只有亲自实践去创造一个东西，才算真正弄懂一个问题。

本着“知其然，知其所以然”的目的，会尽可能的用 Python 库和基本的数学知识，创建经典的深度学习网络。

每篇的计划如下：

第一篇：基础知识简介

第一篇是基础知识简介，对于过于简单的知识点，不会详细叙述，分为两部分：

1. python 基础知识：将后期需要的了解的知识点列出，并给出相关资料。
2. 神经网络基础知识：感知机是神经网络的前身，对感知机简单的介绍。

本篇的目的和内容主要为：介绍感知机和 python

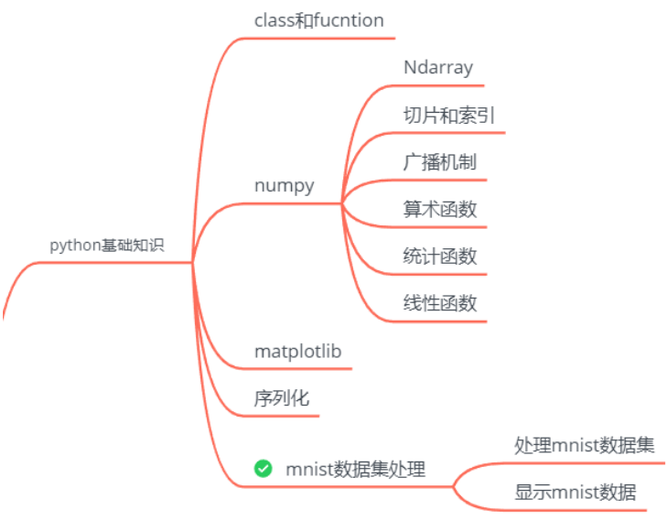
一、Python 基础知识

本章会**列出**实现的神经网络所需要的基础知识，**并给出参考资料**。

TODO

介绍 numpy 库和 matplotlib 库、读写二进制的方法、pkl 等。这些知识会在后面用到，在本篇的最后会以 mnist 数据集为例，创建处理手写体图片的函数，供后使用。

1. *class 和 function*
2. *numpy*
3. *Matplotlib*
4. *序列化*



1.1 class 和 function

[python3 函数]

[python3 面向对象]

1.2 Numpy

Nump(Numerical Python)是 Python 的运算库，支持大规模的数组和矩阵运算。在深度学习的实现中会使用矩阵进行计算，numpy 中实现了很多数据组的运算方法，在后期会用到的有:

- Nmupy 的数据结构 ndarray
- Numpy 的切片和索引
- 广播功能函数
- 算术函数

1.2.1 Ndarray

Numpy 中主要的数据结构是 Ndarray,用于存放同类型元素的多维数组。

Ndarray 的内部如图 1 所示。

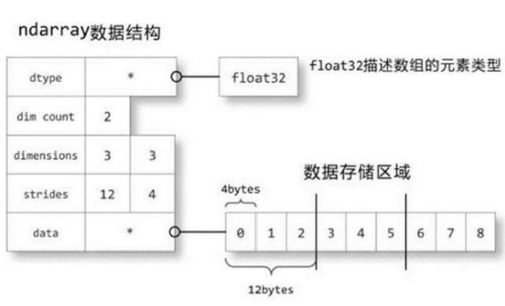


图 1 ndarray 的数据结构

数据类型：dtype,描述数据类型，可以计算每个元素大小;

数组形状：shape,描述数组的大小和形状；

跨度元组，stride：表示从前一个维度到下一个维度需要跨越的字节数；

data: 指向数组的地址；

> ps: 后期会用到 dtype, shape 等成员变量

1.2.2 广播

Numpy 对于不同形状的乘法采用了广播机制。

广播**可以对不同形状的数组做点乘**：将较小的形状按照一定的规则填充，填充的方向依次为由内向外；广播机制在 cudnn、tensorflow 等深度学习框架中同样会使用。

广播是一种 ufunc 的机制是不同形状的数组之间执行算数运算的方式，需要遵循 4 个原则：

1.让所有输入数组都向其中 shape 最长的数组看齐，shape 中不足的部分都通过在前面加 1 补齐

2.输入数组的 shape 是输入数组 shape 的各个轴上的最大值

3.如果输入数组的某个轴和输出数组的对应轴的长度相同或者其长度为 1 时，这个数组能够用来计算，否则出错。

4.输入数组的某个轴的长度为 1 时，沿着此轴运算时都用此轴上的第一组值。

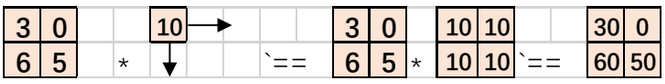
如下图所示，如下图所示:

第一个矩阵是 22，但第二个矩阵并不是 22 的，按照数学运算法则是不能做点乘的；

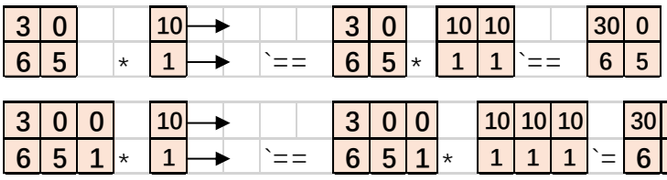
但如果有广播机制，会按照以下方式填充数据，并做乘法：

先从行广播，然后再从列广播，举例如下

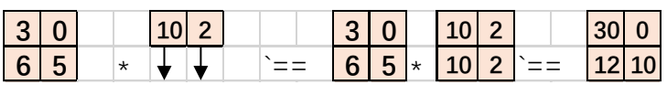
Case1: 行列都不一致。先填充行，再填充列



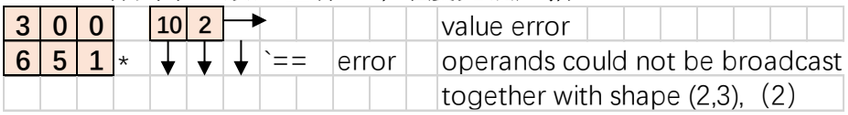
Case2: 行不一致，列一致。先填充行



Case3: 行一致，列不一致。由于行已经一致了，不需要填充，直接填充列。



Case4：行列不一致，且有一个维度无法广播



> 更多关于广播机制，详见：basics.broadcasting

1.2.3 其他知识点

numpy 的切片和索引的有关内容在 fancy-indexing-and-index-tricks 中可以找到。

至于算术运算等网上的资料已经足够多的了，不需要我再重复操作了，这里给出一个官方的资料：numpy-quickstart.html

1.3 matplotlib 和 skimage

matplotlib 和 skimage 在可视化数据的时候会用到。网上的资料足够多的了，在此不多介绍，给出参考资料：

https://www.runoob.com/w3cnote/matplotlib-tutorial.html

https://www.runoob.com/numpy/numpy-matplotlib.html

scikit-image

https://cloud.tencent.com/developer/section/1414638

1.4 python 序列化

Serialization 序列化，是将内存中对象以二进制的方式存储起来，存到磁盘。如果将磁盘中的文件解析成一个对象，这个过程称为 deSerialization。序列化的数据可以用于网络传输，不会因为编码方式而改变。Python 中的序列化由 pickle 模块实现。以下是参考资料：

pickle:Python object serialization

https://docs.python.org/zh-cn/2.7/library/pickle.html

> *在本片的最后会将 mnist 数据集解析以后进行序列化*

1.5 实践：mnist 数据集解析

本章会撰写程序实现一下功能：

1.下载 mnsit 数据集，解析 mnist 数据放在 numpy 的 array 中；

2.将解析的数据先序列化，然后持久化

3.反序列化，读取 mnist 中的一样图像，用 plt 或者 skimage 显示。

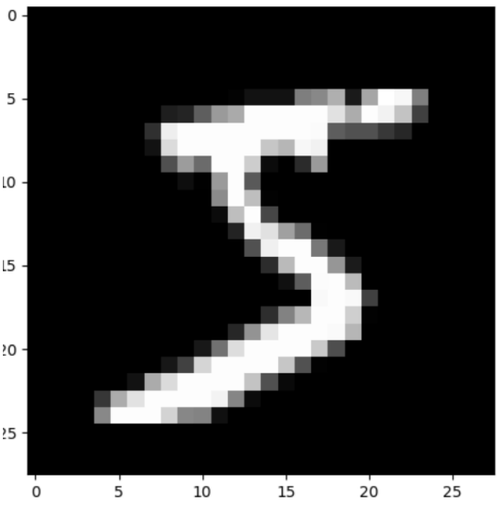
```
# -*- coding: utf-8 -*-
# @File  : mnist.py
# @Author: lizhen
# @Date  : 2020/2/4
# @Desc  : 工具类, datasets/mnist.py

import urllib.request # python3
import os.path
import gzip
import pickle
import os
import numpy as np

# http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
# http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
# http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
# http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz


url_base = "http://yann.lecun.com/exdb/mnist/"
key_file = {
        'train_img':'train-images-idx3-ubyte.gz',
        'train_label':'train-labels-idx1-ubyte.gz',
        'test_img':'t10k-images-idx3-ubyte.gz',
        'test_label':'t10k-labels-idx1-ubyte.gz'
        }

dataset_dir=os.path.dirname(os.path.abspath(__file__))
save_file=dataset_dir + "/mnist.pkl"

train_num = 60000;
test_num  = 10000;
img_dim   = (1, 28, 28)
img_size  = 28*28;


def _download(file_name):
    """
    :param file_name: 下载 mnist 的文件
    :return: null
    """
```

```
    file_path = os.path.join(dataset_dir, file_name)

    if os.path.exists(file_path):
        return

    print("downloading"+file_name+"...")
    urllib.request.urlretrieve(url_base + file_name , file_path)
    print("Done.")

def download_mnist():
    """

    :return:
    """
    for file_name in key_file.values():
        _download(file_name);

def _load_label(file_name):
    """
    解析标签
    :param file_name:
    :return:
    """
    file_path = dataset_dir+'/'+file_name

    print("converting "+file_name+" to numpy Array.")
    with gzip.open(file_path) as f:
        labels = np.frombuffer(f.read(), np.uint8, offset=8)
    print("Done")

    return labels

def _load_img(file_name):
    """
    解析 压缩的图片
    :param file_name:
    :return:
    """
    file_path = dataset_dir +'/' + file_name

    print("converting " + file_name + "to numpy Array")
    with gzip.open(file_path) as f:
        data = np.frombuffer(f.read(), np.uint8, offset=16) # 16*8=
    data = data.reshape(-1, img_size) # N, (W*H*C)=[N,28*28*1]
    print("Done")

    return data

def _convert_numpy():
    """
     解析 image 和 label, 将其转换为 numpy
    """
    dataset = {}
    dataset['train_img'] = _load_img(key_file['train_img'])
    dataset['train_label'] = _load_label(key_file['train_label'])
    dataset['test_img'] = _load_img(key_file['test_img'])
    dataset['test_label'] = _load_label(key_file['test_label'])

    return dataset

def init_mnist():
    """
    初始化 mnist 数据集:
    1. 下载 mnist,
    2. 以二进制的方式读取, 并转换成 numpy 的 ndarray 对象
    3. 将转换后的 ndarray 序列化

    :return:
    """
    print("download mnist dataset...")
    download_mnist()
    print("convert to numpy array...")
    dataset = _convert_numpy()
    print("creating pickle file...")
    with open(save_file, 'wb') as f:
        pickle.dump(dataset, f, -1)
    print("Done!")

def _change_one_hot_label(Y):
    T = np.zeros((Y.size,10))
    for idx,row in enumerate(T):
        row[Y[idx]] = 1
    return T

def load_mnist(normalize=True, flatten=True, one_hot_label=False):
    """

    :param normalize: 将数据标准化到 0.0~1.0
    :param flatten: 是否要将数据拉伸层 1D 数组的形式
    :param one_hot_label:
    :return: (训练数据, 训练标签), (测试数据, 测试 label)
    """


    if not os.path.exists(save_file):
        init_mnist()

    with open(save_file,'rb') as f:
        dataset = pickle.load(f)

    if normalize:
        for key in ('train_img','test_img'):
            dataset[key] = dataset[key].astype(np.float32)
            dataset[key] /=255.0
    if one_hot_label:
        dataset['train_label'] = _change_one_hot_label(dataset['train_label'])
        dataset['test_label']  = _change_one_hot_label(dataset['test_label'])

    if not flatten:
        for key in ('train_img', 'test_img'):
            dataset[key] = dataset[key].reshape(-1,1,28,28) # NCHW

    return (dataset['train_img'],dataset['train_label']),(dataset['test_img'], dataset['test_label'])

if __name__ == '__main__':
    init_mnist()
```

显示图片

```
# -*- coding: utf-8 -*-
# @File  : show_mnist.py
# @Author: lizhen
# @Date  : 2020/1/27
# @Desc  : 显示图片
```

```
from src.datasets.mnist import load_mnist

from skimage import io


def img_show(data):
    # pil_img = Image.fromarray(np.uint8(data))
    io.imshow(data)
    io.show()


(x_train, t_train), (x_test, t_test) = load_mnist(flatten=True, normalize=False)
img = x_train[0]
label = t_train[0]
print(label)

print(img.shape)
img = img.reshape(28,28)
print(img.shape)

img_show(img)
```



2002 年 2 月 11 日修改

二、Percetron（感知机）

TODO:

2.1 感知机简介

感知机是由康奈尔航空实验室心理学家 Frank Rosenblatt 提出的，对，没错。他是一个心理学家，这位心理学家在看到 Hebb 的神经心理学理论以后认为简单的神经元通过有限次的组合是可以创造出识别出物体的机器，在 1957 年的时候 Frank 发布算法模型，并在 1958 年的时候，在《New York Times》上发表文章《Electronic ‘Brain’ Teaches Itself.》，正式把算法取名为“precetron(感知机)”。

PS: 此处的感知机为朴素感知机

感知机如 fig1 所示，◯代表“神经元”或者“节点”。输入的信号（x1,x2）被送往神经元时，会分别乘以固定的权重(w1,w2),神经元会计算出信号的总和，如果总和超过阈值，则会输出，否则不会。

> *本篇属于神经网络的基础知识。*
>
> 1. *阐述感知机的概念*
> 2. *借助感知机实现逻辑与或非以及组合逻辑*
> 3. *阐述感知机的局限性*

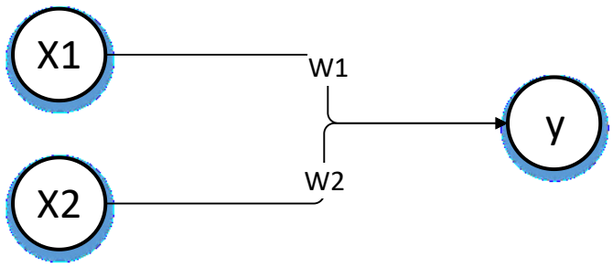


图 1 感知机

感知机的原理就这些，如果用数学的符号表示:

$$y = \begin{cases} 0, (x_1 w_1 + x_2 w_2 \ <= \ thresh) \\ 1, (x_1 w_1 + x_2 w_2 \ > \ thresh) \end{cases}$$

公式 1 感知机原理

朴素感知机自己不会学习权重，也没有复杂的激活函数。一个感知机可以由多个神经元组成，神经元由神经元的输入信号和权重组成;

2.2 简单的逻辑门的实现

感知机之所以能够解决问题，是因为有限神经元通过不同的组合可以解决简单问题的问题，如果一个复杂的问题可以分解为若干个简单的问题，那么就能用感知机解决。

现在先以逻辑门为题材，用感知机的方式实现逻辑**与、或、非，不借助&、||和~**

A. 与门

输入和信号与输出信号均为 1 时，输出 1,；否则输出 0

Table 1 与门的输入和输出信号的真值表

X1	X2	Y
0	0	0
1	0	0
0	1	0
1	1	1

模拟感知机的实现方式:

```
def AND(x1, x2):
    '''
    AND gate
    :param x1: must be {0,1}
    :param x2: must be {0,1}
    :return:
    '''
    w1,w2, theta = 1,1,1
    temp = x1*w1 + x2*w2

    if temp > theta:
        return 1
    elif temp <= theta:
        return 0
    else:
        print("input error, must be in {0,1}")

if __name__ == '__main__':
    x1 =1
    x2 =1
    y = AND(x1,x2)
    print("input x1=%d, x2=%d, output= %s"%(x1,x2,str(y)))
```

B. 非门

与输出的信号相反

```
def Not(x1):
    w1,theta=1,1

    tmp = w1*x1
    if tmp >= theta:
        return 0
```

```
    else:
        return 1
```

Table 2 非门真值表

X1	Y
0	1
1	0

C. 或门

或门是只要有一个输入信号是1，那么输出信号则为1

```
import numpy as np
def OR(x1,x2):
    x = np.array([x1,x2])
    w = np.array([1,1])
    b = 0
    tmp = np.sum(x*w) +b

    if tmp <1:
        return 0
    else:
        return 1

if __name__ == '__main__':
    x1 =1
    x2 =0
    y = OR(x1,x2)
    print("input x1=%d, x2=%d, output= %s"%(x1,x2,str(y)))
```

Table 3 或门真值表

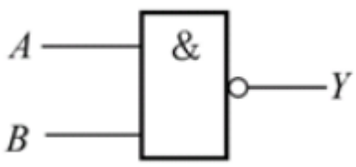
X1	X2	Y
0	0	0
1	0	1
0	1	1
1	1	1

2.3 多层的感知机

以上实现了3个基本的逻辑门；根据这三个基本的逻辑门对应的感知机是可以组合产生新的感知机，这便是多层的感知机。例如：与非门和异或门都是可以通过以上三种简单的逻辑门组合实现。

以**与非门**为例：

与非门是在逻辑与后面添加了逻辑非：




```
def NAND(x1,x2):
    y = AND(x1,x2)
    return Not(y)

if __name__ == '__main__':
    x1 =1
    x2 =1
    y = NAND(x1,x2)
    print("input x1=%d, x2=%d, output= %s"%(x1,x2,str(y)))
```

通过实现的代码和相应的公式公式2可以了解到其输出只能是{0,1}，目前只能对线性问题求解。

对于非线性问题是无法求解的

而且，权重w是依靠人为设置实现的；

2.4 总结

感知机**自己不会学习权重，也没有复杂的激活函数**。一个感知机可以由多个神经元组成，神经元由神经元的输入信号和权重组成；

感知机的局限性：

1. 无法解决非线性的问题
2. 权重 w 是人为给定的

针对以上的两个局限性，人们对感知机新型了改进，提出了神经网络，在第二篇中会详细介绍神经网络的实现。

敬请期待~

2020 年 2 月 2 日星期日

本篇主要介绍有关神经网络的设计与实现。
首先会介绍神经网络和感知机的区别，为什么会有神经网络，它的优势有哪些？
其次会介绍神经网络 forward 推理中涉及的概念和代码实现，并使用 mnist 的程序来进行测试。
最后会介绍神经网络在 backward 中设计的模块和实现代码，并写一个识别 mnist 的程序用于测试。

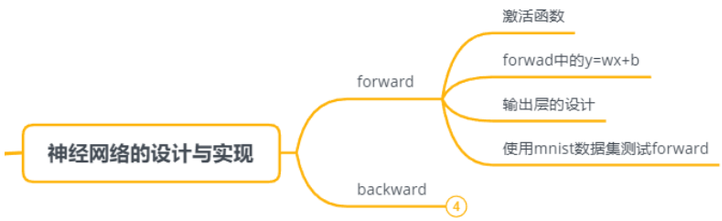
第二篇：神经网络的设计与实现

感知机在**理论上**是可以将计算机上复杂的处理表示出来的，**但是权重的是需要人为手动设置的**。人为的找到合适的、符合预期的输入和输出权重是一件比较耗时且复杂的事情。神经网络的出现可以解决以上的问题：神经网络可以自动的从数据中学习到合适的权重参数。

为了简化表述，先实现神经网络的 forward（推理），然后实现 backward。

三、神经网络的 forward 实现

Forward 指的是神经网络推理，forward 与感知机相比，多了一个激活函数的模块。因此本章需要实现激活函数，另外也需要重新实现 y=w*x+b。本章的顺序如下：



3.1 激活函数的实现

在感知机中讲到用阈值来切换输出，这样的函数称为“**阶跃函数**”：一旦输入超出了阈值，就切换输出。**阶跃函数也算是一种激励函数**。

需要注意激励函数应该具有以下数学特性：

第一：由于后期训练过程中会对激励函数求导，因此这些函数必须符合数学上的可导。

第二：必须为非线性的函数。这可以用公式推一下：

若激励函数为线性函数

从本质上看，激活函数等同于原来的输入的：

$$out^i = W^i * X^i + b^i$$
$$X^{i+1} = k^i * out^i \quad , f(x) = k^i * x$$

即：

$$X^{i+1} = k^* X^i$$

第 i 层的输入是第 i+1 层的 k 倍。

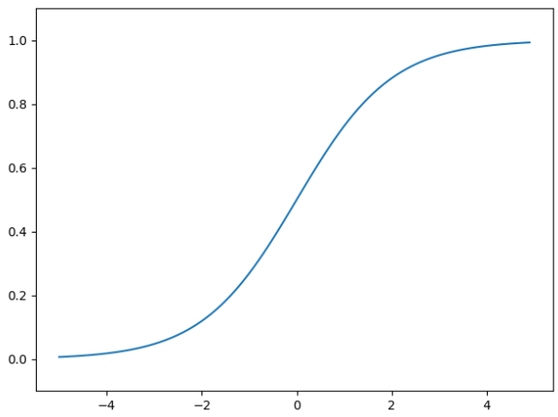
从表象上看，加深网络层次已经失去了意义；等效于无隐含层的网络。

3.1.1 Sigmoid 函数的实现

Sigmoid 函数在（-1,1）区间内变化较大，超出这个范围以后变化较小，可以很好的影响。

$$h(x) = \frac{1}{1+\exp(-x)} \quad ...(1)$$

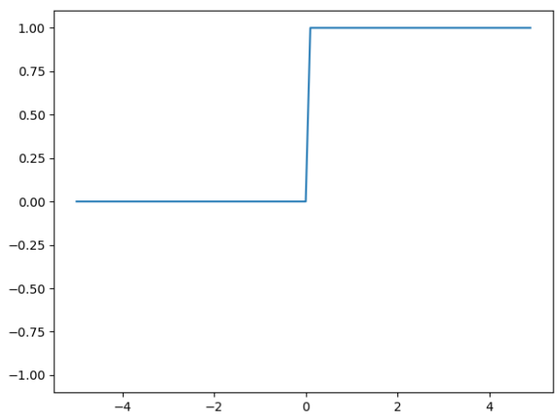
```
def Sigmoid(x):
    return 1/(np.exp(-x) +1)

if __name__ == '__main__':
    x = np.arange(-5.0, 5.0, 0.1)
    y = Sigmoid(x)

    plt.plot(x,y)
    plt.ylim(-0.1,1.1)
    plt.show()
```



3.1.2 阶跃函数

$$h(x) = \begin{cases} x >= \theta \\ x < \theta \end{cases} \quad ... (2)$$

```
def step_func(x):
    temp = x.copy()
    temp = np.where(x > 0, temp, 0)
    temp = np.where(x <= 0, temp, 1)
    return temp


if __name__ == '__main__':
    x = np.arange(-5.0, 5.0, 0.1)

    y = step_func(x)

    plt.plot(x,y)
    plt.ylim(-1.1,1.1)
    plt.show()
```



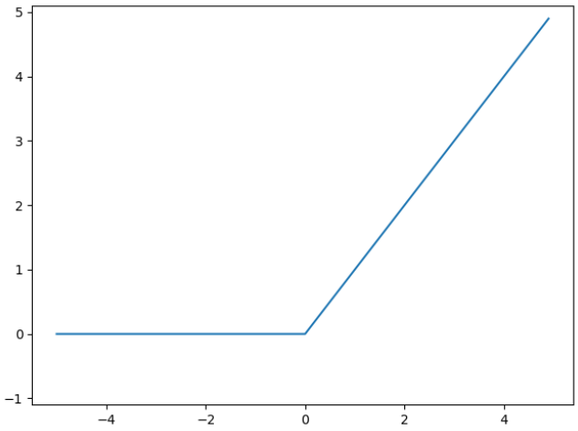
3.1.3 Relu 系列

Relu(Rectified Linear Unit) 函数在输入值大于 0 的情况下保持不变，在输入值小于 0 的情况下，输出等于 0。

阶跃函数和 sigmoid 函数都属于非线性的函数，

$$h(x) = \begin{cases} x, (x > 0) \\ 0, (x <= 0) \end{cases} ..(3)$$

```
def Relu(x):
    return np.maximum(0, x)

if __name__ == '__main__':
    x = np.arange(-5.0, 5.0, 0.1)

    y = Relu(x)

    plt.plot(x, y)
    plt.ylim(-1.1, 5.1)
    plt.show()
```



3.2 forward 的流程

主要是介绍 y=WX+b 的实现。神经网络的 forward 本质是多维数组的运算+激励函数。激活函数已经实现了，因此只要将多维数组的运算了解清楚，便可以实现 forward。forward 的流程如下：

$$A^{(i)} = X^{(i)}W^{(i)} + B^{(i)}$$
$$X^{(i+1)} = f(A^{(i)})$$

p.s.: f 即为激活函数

本质上是矩阵的乘法，借助 np.dot 可以实现；在此不赘述。

3.3 输出层的设计与实现

目前来看，神经网络在分类的问题上可以大致分为两类：

1. 分类问题：数据属于哪个类别，可以使用恒等函数，直接获取预测结果。
2. 回归问题：根据输入，预测一个连续的数值问题。可以使用 softmax。

Ps: 分类问题的输出层也是可以使用 softmax 的，只不过用 softmax 以后得到的数值是一个线性的数值，还需要选取阈值才能划分为类别。

恒等函数是不需要实现了，神经网络的输出节点就是 label 的输出，如下图所示：

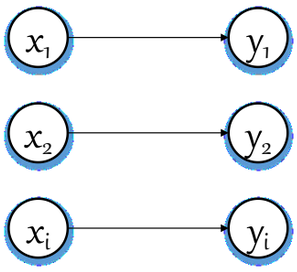


Figure 2 恒等函数

Softmax 函数的数学公式如下：

$$y_k = \frac{\exp(a_k)}{\sum \exp(a_i)}$$

Equation 3 softmax 函数

从公式中可以看出，输出层的各个神经元的输出都会受到输入信号的影响，如下图所示。

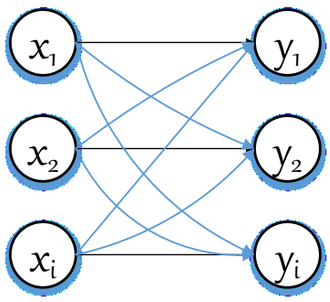


Figure 3 softmax 的表达图

```
def softmax(x):
    '''
    softmax 实现，没有考虑数值溢出
    :param x: ndarray
    :return: y, ndarray
    '''
    a = np.exp(x)
    sum_exp = np.sum(a)
    y= a/sum_exp
    return y
```

上图的代码是按照公式实现的，但是没有考虑数值溢出的情况；由于是 exp 是指数函数，当指数特别大的时候，进行除法的时候会的时候会出现数值溢出。为了避免以上情况，将分子和分母同时乘上常量 C（必须足够大）

$$y_k = \frac{\exp(a_k)}{\sum \exp(a_i)} = \frac{C \cdot \exp(a_k)}{C \cdot \sum \exp(a_i)}$$

$$= \frac{\exp(a_k + \log(C))}{\sum \exp(a_i + \log(C))}$$

Equation 4 sofitmax 公式 2

```
def softmax(x):
    '''
    softmax 实现
    :param x: ndarray
    :return: y, ndarray
    '''
    C = np.max(x)
    exp_a = np.exp(x - C)
    sum_exp = np.sum(exp_a)
    y= exp_a / sum_exp
    return y
```

3.4 测试神经网络的推理

```
# -*- coding: utf-8 -*-
# @File  : day7.py
# @Author: lizhen
# @Date  : 2020/2/4
# @Desc  : 第二篇的实现：对文件的实现

import sys, os
import numpy as np
import pickle
from src.datasets.mnist import load_mnist
from src.common.functions import sigmoid, softmax


def get_data():
    (x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, flatten=True, one_hot_label=False)
    return x_test, t_test


def init_network():
    with open("../datasets/sample_weight.pkl", 'rb') as f:
        network = pickle.load(f)
    return network


def predict(network, x):
    W1, W2, W3 = network['W1'], network['W2'], network['W3']
    b1, b2, b3 = network['b1'], network['b2'], network['b3']

    a1 = np.dot(x, W1) + b1
    z1 = sigmoid(a1)
    a2 = np.dot(z1, W2) + b2
    z2 = sigmoid(a2)
    a3 = np.dot(z2, W3) + b3
    y = softmax(a3)

    return y


x, t = get_data()
network = init_network()
accuracy_cnt = 0
for i in range(len(x)):
    y = predict(network, x[i])
    p= np.argmax(y)
    if p == t[i]:
        accuracy_cnt += 1

print("Accuracy:" + str(float(accuracy_cnt) / len(x)))
```

输出：

Accuracy: 0.9352

四、神经网络的 backward 实现

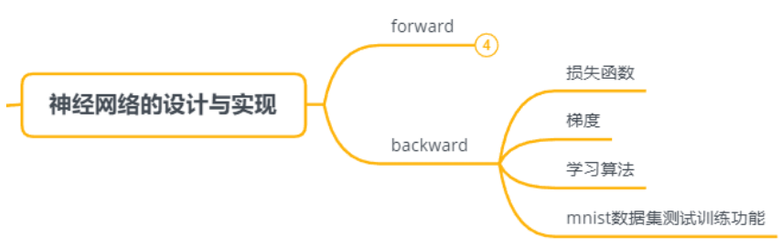
Backward 是神经网络训练过程中包含的一个过程，在这个过程中会通过反馈调节网络中各个节点的权重，以此达到最佳权重参数。在反馈中，loss value 是起点，是衡量与 label 之间差距的值。Loss value 自然是 loss function 计算得出的。

TODO:

本章会讲解常见的两种 loss function

然后会介绍梯度，梯度是用于修改节点权重的。

最后会实现 backward，用 mnist 数据集训练



4.1 损失函数

均方误差（mean squared error）

$$E = \frac{1}{2} \cdot \sum_k (y_k - t_k)^2$$

y_k 表示神经网络的输出

t_k 表示监督数据的$label$

k 表示维度

```
def mean_squared_error(y, t):
    return 0.5*np.sum((y-t)**2)
```

交叉熵误差（cross entropy error）

$$E = - \sum_k t_k \log(y_k)$$

t_k 为标签， y_k 为输出

```
def cross_entropy_error(y,t):
    delta = 1e-7
    return -np.sum(t*np.log(y+delta))
```

mini-batch 版本的损失函数

$$E = - \frac{1}{N} \sum_n \sum_k t_k \log(y_k)$$

t_k 为标签， y_k 为输出

N 为$\min i - batch$

```
def cross_entropy_error(y,t):
# batch版本的交叉熵
    if y.ndim == 1:
        t = t.reshape(1, t.size)
        y = y.reshape(1, y.size)
    batch_size = y.shape[0] # batch

    delta = 1e-7
    return -np.sum(t*np.log(y+delta))/batch_size
```

4.2 梯度

本节介绍梯度法的实现，不涉及神经网络的反馈算法。本节内容是为下一节反馈算法做铺垫。

神经网络学习的本质是根据数据的 label 和预测值的误差，即 loss value，然后根据误差修改权重信息。

数据的 label 和预测值的误差可以使用损失函数来衡量，获得 loss value。

修改权重的信息可以用求数学统计求极值的方式获得。

已知导数为 0 的点为极值点，可以通过求导数 $\frac{\partial L}{\partial W}$ =0 一次性找到极值点。**但是这种方法在数据样本或者 W 的规模相当大的情况下是无法计算的**，在具有多个变量的情况下会计算多次的偏导数，可以想象到是一件耗时耗力的事情。

梯度法来恰恰可以弥补上述的缺陷。

4.2.1 实现梯度的计算

梯度法优势在于可以一次计算出多个变量的偏导数，并汇总成向量，像$\left(\frac{\partial f}{\partial x_0},\frac{\partial f}{\partial x_1},...,\frac{\partial f}{\partial x_i},...,\frac{\partial f}{\partial x_n}\right)$这种汇总而成的向量，称为梯度（注：来自 deep learning from scratch）。

接下来就是梯度的实现了，计算机是无法直接求出偏导数或者导数的，不过根据数学知识可以得到偏导数的近似值，因此可以：

$$\frac{\partial f}{\partial x_0}=\lim\frac{f(x)}{x}\mid x逼近x_0\ \approx\frac{\Delta y}{\Delta t}\approx\ \frac{f(x_0+\Delta t)\ +\ f(x_0\ -\ \Delta t)}{2\Delta t}$$

对应的实现程序如下：

```
def numerical_gradient_1d(f, x):
    '''
数值微分，求 f(x)的梯度
    :param f: 函数
    :param x: 梯度值
    :return: df 在 x 处的导数
    '''
    h = 1e-4 # 0.0001
    grad = np.zeros_like(x)

    for idx in range(x.size):
        tmp_val = x[idx] # 缓存原来的值
        x[idx] = float(tmp_val) + h
        fxh1 = f(x) # f(x+h)

        x[idx] = tmp_val - h
        fxh2 = f(x) # f(x-h)
        grad[idx] = (fxh1 - fxh2) / (2*h)

        x[idx] = tmp_val # 原来的值

    return grad
```

4.2.2 测试梯度计算方法是否有用

在此简单的验证$f(x_0,\ x_1)=x_0^2\ +\ x_1^2$的在点$[3,4]$上的梯度；

根据数学知识，可以知道梯度为：$\left(\frac{\partial f}{\partial x_0},\frac{\partial f}{\partial x_1}\right)=(2x_0,\ 2x_1)$

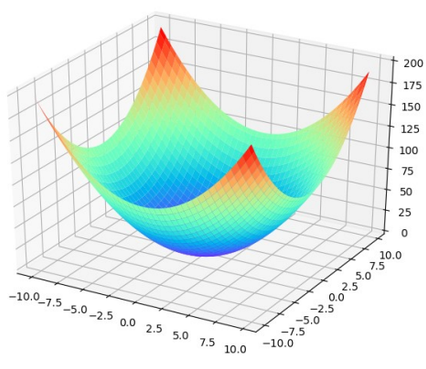
那么$(x_0,x_1)\ =\ (3,4)$时，梯度为$(6,8)$

下面开始验证

片段 1：function 的实现：

```
def function(x):
    return x[0]**2 + x[1]**2
```

function 的坐标图如下，可见整个函数的最小值在（0，0）处，因此点（3，4）的梯度方向应该指向（0，0），即：梯度值应该为正数。




```
from matplotlib import pyplot as plot  # 用来绘制图形
import numpy as np  # 用来处理数据
```

```
from mpl_toolkits.mplot3d import Axes3D  # 用来给出三维坐标系。
figure = plot.figure()

# 画出三维坐标系：
axes = Axes3D(figure)
X = np.arange(-10, 10, 0.25)
Y = np.arange(-10, 10, 0.25)
X, Y = np.meshgrid(X, Y) # 限定图形的样式是网格线的样式：
Z = function([X,Y])
axes.plot_surface(X, Y, Z, cmap='rainbow') # 绘制曲面，采用彩虹色着色：
# 图形可视化：
plot.show()

if __name__ == '__main__':
    # show()
    print(numerical_gradient(function, np.array([3.0,4.0])))
```

对应的输出：

[6. 8.]

验证正确,说明 grad 的书写没有问题。

4.2.3 梯度法的实现

神经网络学习的主要任务是在学习过程中寻找最优参数，这些参数使得 loss function 取得最小值。这里的神经网络可以用 g(x)表示。

一般来说，神经网络的参数空间较大，损失函数也较为复杂，往往会通过梯度来寻找 g(x)的最小值。但需要注意：梯度表示的各个点处函数值减小最多的方向，无法保证是真正的应该进行梯度下降的方向。

尽管如此，沿着梯度的方向依旧是可以最大限度的找到减小损失函数的值。通过不断的向梯度的方向迈进，便会使得 loss function 逐渐减小(这个过程被称为梯度法，gradient method)。

梯度法是解决机器学习中优化问题的常用方法，根据优化的目标可以分为：梯度下降法和梯度上升法。

用数学表达为:

$$x_0 = x_0 - \eta \cdot \frac{\partial f}{\partial x_0}$$

$$x_1 = x_1 - \eta \cdot \frac{\partial f}{\partial x_1}$$

Equation 5 梯度下降的表达式

利用上面的公式和 numerical_gradient，梯度法的实现如下：

```
def gradient_descent(f, init_x, learning_rate=0.01, step_num=200):
    '''
    通过一步一步的迭代，优化目标函数，找出使得目标函数最小的点
    :param f: 目标函数
    :param init_x: 初始位置
    :param learning_rate: 学习率
    :param step_num: 迭代次数
    :return:
    '''
    x = init_x
    for i in range(step_num):
        grad = numerical_gradient(f,x)
        x = x - learning_rate*grad # 公式的实现
    return x

if __name__ == '__main__':
    # show()
    #print(numerical_gradient(function, np.array([3.0,4.0])))
    min_value = gradient_descent(function, init_x=np.array([3.0,4.0]))
    print(min_value)
```

到此为止，我们已经实现修改权重的功能。

4.3 学习算法的实现

本节会阐述学习算法的伪代码，具体实现会在 3.4 给出。

神经网络的学习中使用到了梯度法（见 3.2 节），根据梯度法我们可以了解到神经网络学习的过程：可以按照以下 4 个步骤进行：

Step1：获取 minibatch：

从数据集中选取一部分数据，这部分数据称为 mini-batch，现在的目标就是减小 minibatch 的 loss fucntion value。

Step2：计算梯度值

为了减小 loss function 的值，求出各个权重参数的梯度。梯度是表示损失函数值减小最多的方向。

Step3：更新参数

梯度表示的是损失函数减小的方向；因此将权重参数沿着梯度所指的方向进行微小的更新。

Step4：重复

重复 Step1,2,3

以上使用的是梯度下降的方法，由于是随机选择的 mini-batch 数据（也就是说随机选择的初始点），所以称为随机梯度下降（stochastic gradient descent）注：见 Deep Learning from Scratch

4.4 基于 mnist 数据集的神经网络的训练

本节整合前面的代码，实现一个两层的神经网络。用 mnist 数据集训练，来了解整个学习的过程。整个过程会尽量简化.

为了简化起见，网络输入层是 784*1，有两个隐含层,神经元的数量分别是 50 和 10; 由于输出层和上一层的输出数量一致,因此用恒等函数即可，损失函数使用交叉熵。

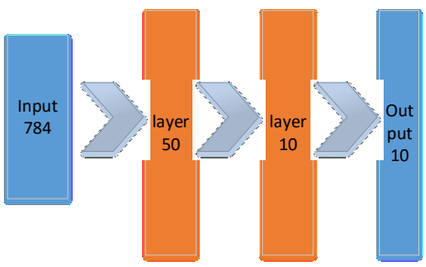


Figure 4 网络结构

见 code：

```
# -*- coding: utf-8 -*-
# @File  : two_layer_net.py
# @Author: lizhen
# @Date  : 2020/1/28
# @Desc  : 使用梯度的网络

from src.common.functions import *
from src.common.gradient import numerical_gradient
import numpy as np

class TwoLayerNet:

    def __init__(self, input_size, hidden_size, output_size, weight_init_std=0.01):

        self.params = {}
```

```
        self.params['W1'] = weight_init_std * np.random.randn(input_size, hidden_size)
        self.params['b1'] = np.zeros(hidden_size)
        self.params['W2'] = weight_init_std * np.random.randn(hidden_size, output_size)
        self.params['b2'] = np.zeros(output_size)

    def predict(self, x):
        W1, W2 = self.params['W1'], self.params['W2']
        b1, b2 = self.params['b1'], self.params['b2']

        a1 = np.dot(x, W1) + b1
        z1 = sigmoid(a1)
        a2 = np.dot(z1, W2) + b2
        y = softmax(a2)

        return y

    # x:输入参数, t:label
    def loss(self, x, t):
        y = self.predict(x)

        return cross_entropy_error(y, t)

    def accuracy(self, x, t):
        y = self.predict(x)
        y = np.argmax(y, axis=1)
        t = np.argmax(t, axis=1)

        accuracy = np.sum(y == t) / float(x.shape[0])
        return accuracy

    # x:输入参数, t:label
    def numerical_gradient(self, x, t):
        loss_W = lambda W: self.loss(x, t)

        grads = {}
        grads['W1'] = numerical_gradient(loss_W, self.params['W1'])
        grads['b1'] = numerical_gradient(loss_W, self.params['b1'])
        grads['W2'] = numerical_gradient(loss_W, self.params['W2'])
        grads['b2'] = numerical_gradient(loss_W, self.params['b2'])

        return grads

    def gradient(self, x, t):
        W1, W2 = self.params['W1'], self.params['W2']
        b1, b2 = self.params['b1'], self.params['b2']
        grads = {}

        batch_num = x.shape[0]

        # forward
        a1 = np.dot(x, W1) + b1
        z1 = sigmoid(a1)
        a2 = np.dot(z1, W2) + b2
        y = softmax(a2)

        # backward
        dy = (y - t) / batch_num
        grads['W2'] = np.dot(z1.T, dy)
        grads['b2'] = np.sum(dy, axis=0)

        dz1 = np.dot(dy, W2.T)
        da1 = sigmoid_grad(a1) * dz1
        grads['W1'] = np.dot(x.T, da1)
        grads['b1'] = np.sum(da1, axis=0)

        return grads
```

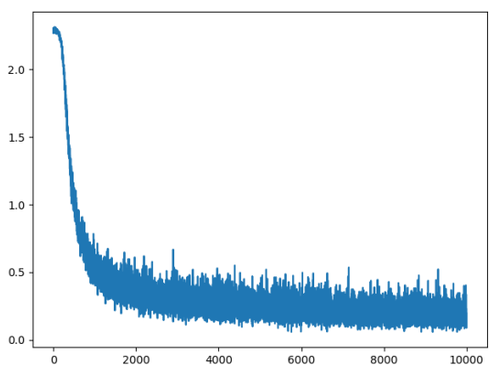
```
# -*- coding: utf-8 -*-
# @File  : train_neuralnet.py
# @Author: lizhen
# @Date  : 2020/2/2
# @Desc  : 第三篇的实现: 利用梯度

import numpy as np
import matplotlib.pyplot as plt
from src.datasets.mnist import load_mnist
from src.test.two_layer_net import TwoLayerNet

# 获取训练数据
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, one_hot_label=True)

network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)

iters_num = 10000  # 迭代次数
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.1

train_loss_list = []
train_acc_list = []
test_acc_list = []

iter_per_epoch = max(train_size / batch_size, 1)

for i in range(iters_num):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    # 计算梯度
    grad = network.numerical_gradient(x_batch, t_batch)
    # grad = network.gradient(x_batch, t_batch) # 较快

    # 更新权重
```

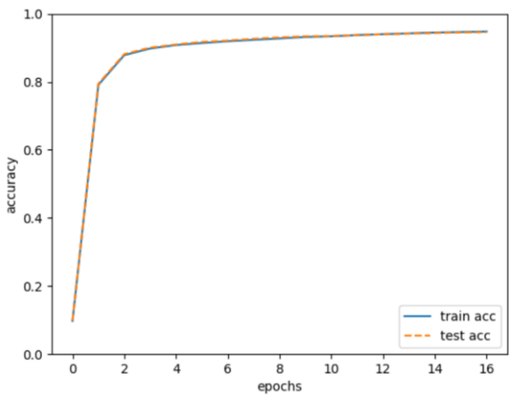
```
    for key in ('W1', 'b1', 'W2', 'b2'):
        network.params[key] -= learning_rate * grad[key]

    loss = network.loss(x_batch, t_batch)
    train_loss_list.append(loss)

    if i % iter_per_epoch == 0:
        train_acc = network.accuracy(x_train, t_train)
        test_acc = network.accuracy(x_test, t_test)
        train_acc_list.append(train_acc)
        test_acc_list.append(test_acc)
        print("train acc, test acc , loss | " + str(train_acc) + ", " + str(test_acc)+", "+str(loss))


# 绘制图
markers = {'train': 'o', 'test': 's'}
x = np.arange(len(train_acc_list))
plt.plot(x, train_acc_list, label='train acc')
plt.plot(x, test_acc_list, label='test acc', linestyle='--')

plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
plt.legend(loc='lower right')
plt.show()

# plt.plot(train_loss_list,label='loss value')
# plt.show()
```

输出：

```
train acc, test acc , loss | 0.9361, 0.9374, 0.25693256148898147
train acc, test acc , loss | 0.9386, 0.9376, 0.19519060413492628
train acc, test acc , loss | 0.94075, 0.9406, 0.25280034505164123
train acc, test acc , loss | 0.9432333333333334, 0.9413, 0.14324609003783176
train acc, test acc , loss | 0.9454666666666667, 0.9427, 0.19827864727352307
train acc, test acc , loss | 0.9465833333333333, 0.9435, 0.1750376927445897
```

Loss 曲线图和 acc 变化曲线如下





2020 年 2 月 3 日星期一

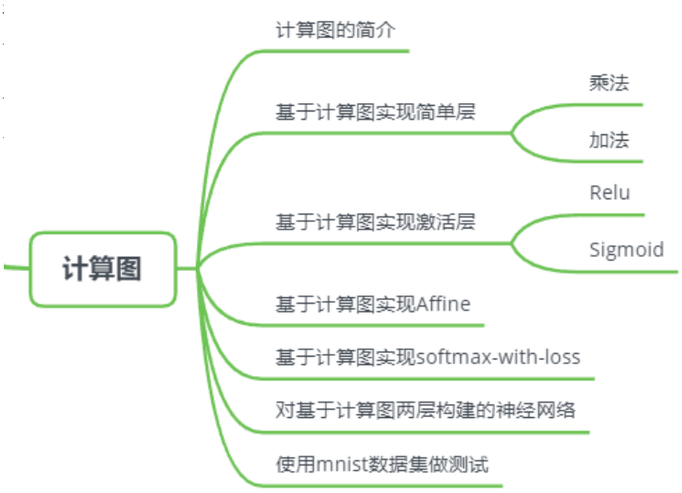
第三篇：基于计算图的神经网络的设计与实现

在第二篇中介绍了用数值微分的形式计算神经网络的梯度，数值微分的形式比较简单也容易实现，但是计算上比较耗时。本章会介绍一种能够较为高效的计算出梯度的方法：基于图的误差反向传播。

根据 deep learning from scratch 这本书的介绍，在误差反向传播方法的实现上有两种方法：一种是基于数学式的（第二篇就是利用的这种方法），一种是基于计算图的。这两种方法的本质是一样的，有所不同的是表述方法。计算图的方法可以参考 feifei li 负责的斯坦福大学公开课 CS231n 或者 theano 的 tutorial/Futher readings/graph Structures.

五、计算图

[使用文档中的独特引言吸引读者的注意力，或者使用此空间强调要点。要在此页面上的任何位置放置此文本框，只需拖动它即可。]



P.S.：利用计算图的求导数的步骤类似于链式法则，这里先挖个坑，稍后求 sigmoid 的微分的时候会举例。

5.1 简单层的实现

Ps: 在前面的几章中，我对代码的重视程度并不大，这是因为前几章的涉及的代码都是作为理论基础。在后面的章节中会注意代码的组织结构。

在实现方面会尽量使用 python 的类。

为此，创建一个所有类的基类：BaseLayer

Forward 函数是用于推理的

Backward 函数是反向求导使用的

```
class BaseLayer:
    '''
    所有层的基类
    '''
    def forward(self,x,y):
        raise NotImplementedError
    def backward(self,dout):
        raise NotImplementedError
    def toString(self):
        raise NotImplementedError
```

5.1.1 乘法层的实现

```
    class MulLayer(BaseLayer):
        def __init__(self):
            self.x = None
            self.y = None

        def forward(self,x,y):
            self.x = x
            self.y = y
            out = x*y

            return out

        def backward(self,dout):
            '''
            反馈方面是反转 x,y
            :param dout:
            :return:
```

```
        '''
        dx = dout * self.y
        dy = dout * self.x
        return  dx, dy

    def toString(self):
        print("name: Multi")
        print("x.shape %s"%str(self.x.shape))
        print("y.shape %s"%str(self.y.shape))
```

测试：

5.1.2 加法层的实现

```
class AddLayer(BaseLayer):
    def __init__(self):
        self.x = None
        self.y = None

    def forward(self,x,y):
        self.x = x
        self.y = y
        out = self.x+self.y
        return  out
    def backward(self,dout):
        dx = dout*1
        dy = dout*1
        return dx,dy
    def toString(self):
        print("name: Add")
        print("x.shape %s"%str(self.x.shape))
        print("y.shape %s"%str(self.y.shape))
```

5.2 激活层的实现

本节的函数是基于 **第二篇：2.激活函数的设计与实现**

由于需要将计算图的思路用在神经网络中，所以需要把之前实现的激活函数重新实现一次。

5.2.1 ReLU 层

Relu 的 backward 是 relu 函数的微分。

$$h(x) = \begin{cases} x, & (x > 0) \\ 0, & (x <= 0) \end{cases}$$

微分的计算如下：

$$\frac{dh(x)}{x} = \begin{cases} 1, & (x > 0) \\ 0, & (x <= 0) \end{cases}$$

$$\partial y = \partial L \cdot y^2 \cdot \exp(-x) \ = \partial L \cdot y(1-y)$$

$$y \ = \ \mathbf{W} \cdot \mathbf{X} + \mathbf{b}$$

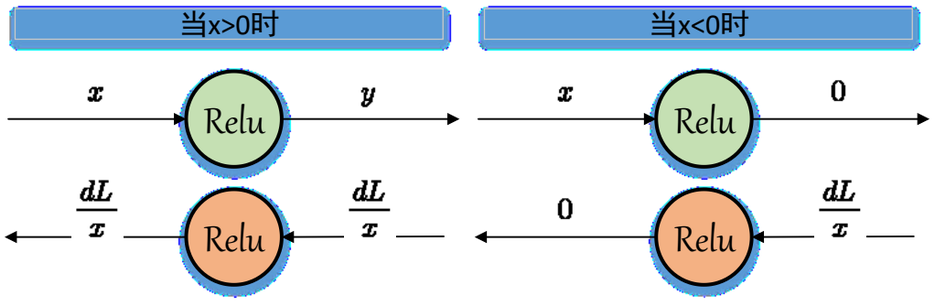
$$\frac{\partial L}{\partial W} \ = \ \boldsymbol{X}^T \cdot \frac{\partial L}{\partial Y}$$

$$\frac{\partial L}{\partial \boldsymbol{X}} \ = \ \frac{\partial L}{\partial Y} \cdot \boldsymbol{W}^{\boldsymbol{T}}$$

其计算图的表示如下：

可见当 x<0 的时候，在 backward 部分应该输出 0，其余的情况输出的是微分。

Backward 中的 dout 变量表示的是 dL



用使用公式求微分，和使用计算图求微分的结果是一样的。

```
class Relu:

    def __init__(self):
        self.mask = None

    def forward(self, x):
        self.mask = (x <= 0)
        out = x.copy()
        out[self.mask] = 1

        return out

    def backward(self, dout):
        dout[self.mask] = 0
        dx = dout

        return dx
```

5.2.2 Sigmoid 层

$$y = \frac{1}{1+\exp(-x)}$$

微分的公式为：

$$\partial y = \partial L \cdot y^2 \cdot \exp(-x) = \partial L \cdot y(1-y)$$

Backward 中的 dout 变量表示的是 dL，

```
class Sigmoid:
    def __init__(self):
        self.out = None

    def forward(self, x):
        out = sigmoid(x)
        self.out = out
        return out

    def backward(self, dout):
        dx = dout * (1.0 - self.out) * self.out

        return dx
```

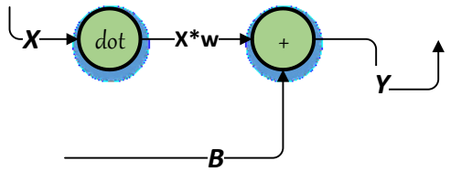
5.3 Affine 层

Affine 层其实是全连接层，Affine 的名字来源于斯坦福的 C231，网上也很多实现:
https://blog.csdn.net/achcxchca/article/details/80959735

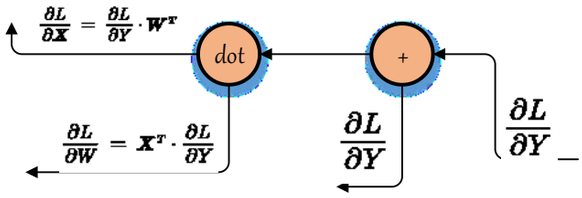
Affine（仿射）的意思可以理解为前面一层中的每一个神经元都连接到当前层中的每一个神经元，加入仿射层输出的形状和数值都有所改变，这点类似于图像处理的仿射变换。在许多情况下，这是神经网络的「标准」层。仿射层通常被加在卷积神经网络或循环神经网络做出最终预测前的输出的顶层。

PS: Affine 层的输出 X 是需要考虑到输入是以单个数据为对象的，还是以多个输入为对象的。

Forward 的计算图：



Backward 计算图：




```
class Affine:
    def __init__(self, W, b):
        self.W = W
        self.b = b

        self.x = None
        self.original_x_shape = None
        # 计算微分
        self.dW = None
        self.db = None

    def forward(self, x):
        # 为了支持支持张量的计算，将 x 先做形状修改 相当于 transOp
        self.original_x_shape = x.shape
        x = x.reshape(x.shape[0], -1)

        # out = w*x+b
        self.x = x
        out = np.dot(self.x, self.W) + self.b

        return out

    def backward(self, dout):

        dx = np.dot(dout, self.W.T)
        self.dW = np.dot(self.x.T, dout)
        self.db = np.sum(dout, axis=0)

        # 将 dx 形状 trans 回去（张量支持）
        dx = dx.reshape(*self.original_x_shape)
        return dx
```

5.5 Softmax-with-Loss 层

SoftMax 函数会将函数正则话以后在输出，比如在手写体识别中，输出层一般是 softmax。考虑到里面也使用了交叉熵(cross entroy error)，所以命名为 softmax-with-loss。

```
class SoftmaxWithLoss:
    def __init__(self):
        self.loss = None
        self.y = None  # softmax 的输出
        self.t = None  # 标签函数

    def forward(self, x, t):
        self.t = t
        self.y = softmax(x)
        self.loss = cross_entropy_error(self.y, self.t)

        return self.loss

    def backward(self, dout=1):
```

```
        batch_size = self.t.shape[0]
        if self.t.size == self.y.size:  # 处理 one-hot
            dx = (self.y - self.t) / batch_size
        else:
            dx = self.y.copy()
            dx[np.arange(batch_size), self.t] -= 1
            dx = dx / batch_size

        return dx
```

5.6 基于计算图构建的神经网络

```
# -*- coding: utf-8 -*-
# @File  : two_layer_net_v2.py
# @Author: lizhen
# @Date  : 2020/1/28
# @Desc  : 这个是使用计算图的网络
import numpy as np
from src.common.layers import *
from src.common.gradient import numerical_gradient
from collections import OrderedDict


class TwoLayerNet:

    def __init__(self, input_size, hidden_size, output_size, weight_init_std = 0.01):
        # 初始化成员
        self.params = {}
        self.params['W1'] = weight_init_std * np.random.randn(input_size, hidden_size)
        self.params['b1'] = np.zeros(hidden_size)
        self.params['W2'] = weight_init_std * np.random.randn(hidden_size, output_size)
        self.params['b2'] = np.zeros(output_size)

        # 网络层
        self.layers = OrderedDict()
        self.layers['Affine1'] = Affine(self.params['W1'], self.params['b1'])
        self.layers['Relu1'] = Relu()
        self.layers['Affine2'] = Affine(self.params['W2'], self.params['b2'])

        self.lastLayer = SoftmaxWithLoss()

    def predict(self, x):
        for layer in self.layers.values():
            x = layer.forward(x)

        return x

    def loss(self, x, t):
        y = self.predict(x)
        return self.lastLayer.forward(y, t)

    def accuracy(self, x, t):
        y = self.predict(x)
        y = np.argmax(y, axis=1)
        if t.ndim != 1 : t = np.argmax(t, axis=1)

        accuracy = np.sum(y == t) / float(x.shape[0])
        return accuracy


    def numerical_gradient(self, x, t):
        loss_W = lambda W: self.loss(x, t)

        grads = {}
        grads['W1'] = numerical_gradient(loss_W, self.params['W1'])
        grads['b1'] = numerical_gradient(loss_W, self.params['b1'])
        grads['W2'] = numerical_gradient(loss_W, self.params['W2'])
        grads['b2'] = numerical_gradient(loss_W, self.params['b2'])

        return grads

    def gradient(self, x, t):
        # forward
        dout=self.loss(x, t)

        # backward
        # dout = 1
        dout = self.lastLayer.backward(dout)

        layers = list(self.layers.values())
        layers.reverse()
        for layer in layers:
            dout = layer.backward(dout)

        # 更新
        grads = {}
        grads['W1'], grads['b1'] = self.layers['Affine1'].dW, self.layers['Affine1'].db
        grads['W2'], grads['b2'] = self.layers['Affine2'].dW, self.layers['Affine2'].db

        return grads
```

5.7 使用 mnist 数据集测试

本节的测试是为了测试简单层的书写是否符合逻辑。本次依旧是使用 mnist 数据集训练识别手写体。与第二篇所不同的是：本次使用的是基于计算图的逻辑实现。

```
# coding: utf-8
import sys, os
sys.path.append(os.pardir)

import numpy as np
from src.datasets.mnist import load_mnist
from src.test.two_layer_net_v2 import TwoLayerNet


(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, one_hot_label=True)

network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)

iters_num = 10000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.1

train_loss_list = []
train_acc_list = []
test_acc_list = []

iter_per_epoch = max(train_size / batch_size, 1)

for i in range(iters_num):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    grad = network.gradient(x_batch, t_batch)

    # 更新
    for key in ('W1', 'b1', 'W2', 'b2'):
        network.params[key] -= learning_rate * grad[key]

    loss = network.loss(x_batch, t_batch)
    train_loss_list.append(loss)

    if i % iter_per_epoch == 0:
        train_acc = network.accuracy(x_train, t_train)
        test_acc = network.accuracy(x_test, t_test)
        train_acc_list.append(train_acc)
        test_acc_list.append(test_acc)
        print(train_acc, test_acc)
```

训练过程中的输出

train loss:0.933130708198716

train loss:0.8530083545338488

train loss:0.8060872314284586

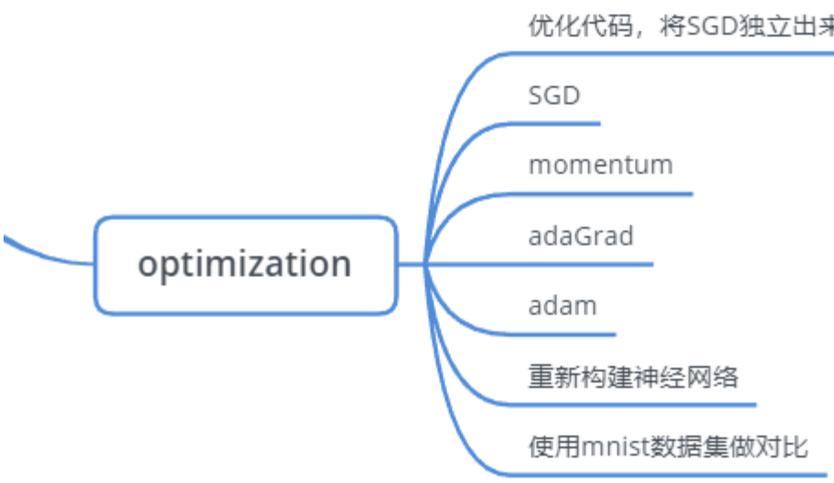
=============== Final Test Accuracy ===============

test acc:0.9939

Saved Network Parameters!

本篇主
首先，
其次，

使用不同的优化器了



第四篇： optimization 的实现

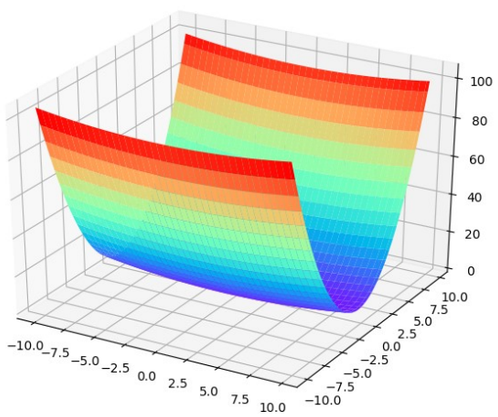
六 optimization 的实现

神经网络的学习目的是找到使得损失函数的值尽可能小的参数，这个寻找最优参数的过程称为最优化（optimization）。本章会介绍 4 中优化函数，给出实现代码和测例。

测例方面，首先会用函数做测试，观察是否可以逼近极小值；然后会用 mnist 数据集做测试观察是否能够收敛。

所测试的函数公式如下：

$$z = \frac{x^2}{20} + y^2$$



图表 2 f(x,y,z) 的图像

6.1 SGD 的实现

在前几篇两篇中一直使用同一种的方法修改权重。这种方法为了寻找最优参数，将参数的梯度（导数）作为依据，根据参数的梯度，就知道梯度的方向，并沿着梯度的方向更新参数。通过不断的更新迭代上述步骤，从而逐渐接近最优参数。这个过程称为随机梯度下降（stochastic gradient descent, SGD），**前面几篇的训练中一直使用的是 SGD**。SGD 的数学表达式如下：

$$W = W - \eta \cdot \frac{\partial L}{\partial W}$$

虽然实现简单，也能得到优化参数。**但是 SGD 有很多缺点,比较明显的一个就是梯度的方向并没有指向最小值的方向,**有可能得到的最优解是局部最优解。

因此我们还应该实现其他的优化参数的方法。不过在此之前，需要对程序做修改，之前代码的耦合性很高，先把 SGD 独立抽出来，写成一个 class,代码如下。

```
class SGD:

    """梯度下降 (Stochastic Gradient Descent) """

    def __init__(self, lr=0.01):
        self.lr = lr

    def update(self, params, grads):
        for key in params.keys():
            params[key] -= self.lr * grads[key]
```

PS:2/3 里面均是采用 SGD 寻找最优参数，我们也可以从代码中找到，优化代码结构。

在第 3 篇的 mnist 数据集训练手写体:

```
def gradient(self, x, t):
    # forward
    dout=self.loss(x, t)

    # backward
    # dout = 1
    dout = self.lastLayer.backward(dout)

    layers = list(self.layers.values())
    layers.reverse()
    for layer in layers:
        dout = layer.backward(dout)

    # 更新
    grads = {}
    grads['W1'], grads['b1'] = self.layers['Affine1'].dW, self.layers['Affine1'].db
    grads['W2'], grads['b2'] = self.layers['Affine2'].dW, self.layers['Affine2'].db

    return grads
```

第二篇里的 2.4 节，mnist 数据集训练手写体:

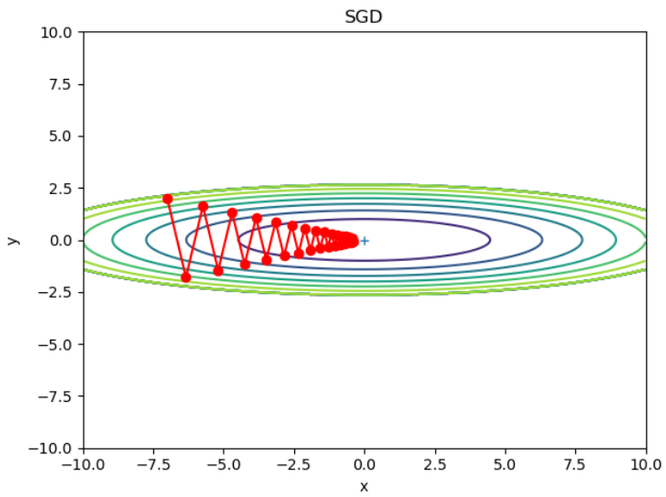
```
def gradient(self, x, t):
    W1, W2 = self.params['W1'], self.params['W2']
    b1, b2 = self.params['b1'], self.params['b2']
    grads = {}

    batch_num = x.shape[0]

    # forward
    a1 = np.dot(x, W1) + b1
    z1 = sigmoid(a1)
    a2 = np.dot(z1, W2) + b2
    y = softmax(a2)

    # backward
    dy = (y - t) / batch_num
    grads['W2'] = np.dot(z1.T, dy)
    grads['b2'] = np.sum(dy, axis=0)

    dz1 = np.dot(dy, W2.T)
    da1 = sigmoid_grad(a1) * dz1
    grads['W1'] = np.dot(x.T, da1)
    grads['b1'] = np.sum(da1, axis=0)

    return grads
```

测试 SGD:

与之前测试梯度的方法类似：给出一个函数公式，求它的梯度。但这次会将每次梯度下降的点绘制出来。

```
# coding: utf-8
import numpy as np

import matplotlib.pyplot as plt
from collections import OrderedDict
from src.common.optimizer import *


def f(x, y):
    return x**2 / 20.0 + y**2


def df(x, y):
    return x / 10.0, 2.0*y

init_pos = (-7.0, 2.0)
params = {}
params['x'], params['y'] = init_pos[0], init_pos[1]
grads = {}
grads['x'], grads['y'] = 0, 0
```

```
optimizer= SGD(lr=0.95)
# optimizer=Momentum(lr=0.1)
# optimizer=AdaGrad(lr=1.5)
# optimizer= Adam(lr=0.3)
# optimizer=RMSprop()



x_history = []
y_history = []
params['x'], params['y'] = init_pos[0], init_pos[1]

for i in range(30):
    x_history.append(params['x'])
    y_history.append(params['y'])

    grads['x'], grads['y'] = df(params['x'], params['y'])
    optimizer.update(params, grads)


x = np.arange(-10, 10, 0.01)
y = np.arange(-5, 5, 0.01)

X, Y = np.meshgrid(x, y)
Z = f(X, Y)

mask = Z > 7
Z[mask] = 0

plt.plot(x_history, y_history, 'o-', color="red")
plt.contour(X, Y, Z)
plt.ylim(-10, 10)
plt.xlim(-10, 10)
plt.plot(0, 0, '+')
    # colorbar()
    # spring()
plt.title("Momentum")
plt.xlabel("x")
plt.ylabel("y")

plt.show()
```



图表 3 SGD 在 f(x,y,z)上的梯度曲线

6.2 Momentum 的实现

针对 SGD 的缺点，添加动量 v，对应物理上的速度。这样公式类似于物理上的瞬时速度的公式，$\alpha \boldsymbol{v}$ 对应物理上的阻力。

α 是加速度，一般是常量 0.9

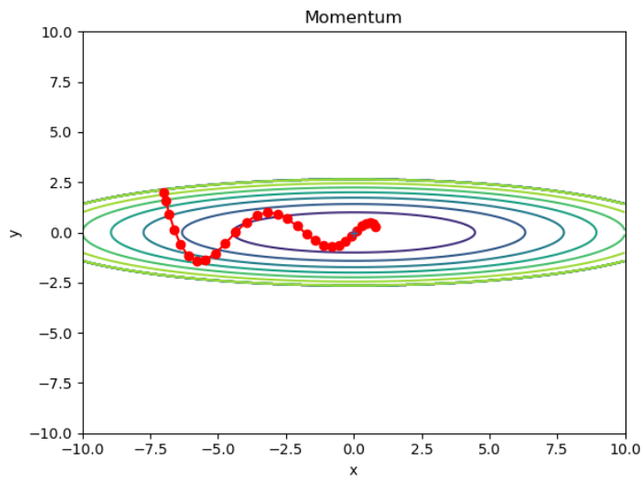
$$\boldsymbol{v} = \alpha \boldsymbol{v} - \eta \cdot \frac{\partial L}{\partial \boldsymbol{W}}$$

$$\boldsymbol{W} = \boldsymbol{W} + \boldsymbol{v}$$

```
class Momentum:

    """添加动量后的 SGD"""

    def __init__(self, lr=0.01, momentum=0.9):
        self.lr = lr
        self.momentum = momentum
        self.v = None

    def update(self, params, grads):
```

```
        if self.v is None:
            self.v = {}
            for key, val in params.items():
                self.v[key] = np.zeros_like(val)

        for key in params.keys():
            self.v[key] = self.momentum*self.v[key] - self.lr*grads[key]
            params[key] += self.v[key]
```

代码测试：



参考资料：

https://tensorflow.google.cn/api_docs/cc/class/tensorflow/ops/apply-momentum?hl=en

6.3 AdaGrad

AdaGrad 又称为学习率衰减。在神经网络的训练过程中，学习率的设置很重要，如果学习设置过小，导致学习的时间很长；学习率过大导致不能收敛。

针对以上的这种情况，有人提出了 learning rate decay 的方法：可以让学习率一开始比较大，随着训练的次数增加，学习率不断减少。AdaGrad（Adaptive Grad）会对每个元素的学习率进行适当的调整。AdaGrad 的数学表达式如下：

$$\boldsymbol{h} = \boldsymbol{h} + \frac{\partial L}{\partial \boldsymbol{W}} \odot \frac{\partial L}{\partial \boldsymbol{W}}$$

$$\boldsymbol{W} = \boldsymbol{W} - \eta \frac{1}{\sqrt{\boldsymbol{h}}} \frac{\partial L}{\partial \boldsymbol{W}}$$

$\frac{\partial L}{\partial \boldsymbol{W}}$ 表示损失函数关于 $\boldsymbol{W}$ 的梯度；

η 表示学习率

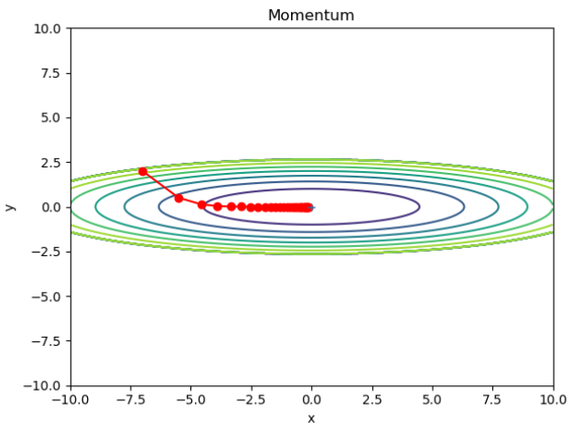
由于 $\boldsymbol{h} = \boldsymbol{h} + \frac{\partial L}{\partial \boldsymbol{W}} \odot \frac{\partial L}{\partial \boldsymbol{W}}$ 保存了以前所有梯度值的**平方和**，所以在更新参数的时候需要对除以 $\sqrt{\boldsymbol{h}}$ 才可以。这样一来，参数元素中变动较大的元素的学习率会降低。可以看做按照参数的元素进行学习率的衰减，使得变动较大的参数的学习率减小。、

AdaGrad 会记录所有梯度的平方和，然后在减去对应的梯度的权重，因此学习越深入，更新的幅度会越小。

```
class AdaGrad:

    """AdaGrad 的实现"""

    def __init__(self, lr=0.01):
        self.lr = lr
        self.h = None

    def update(self, params, grads):
        if self.h is None:
```

```
            self.h = {}
            for key, val in params.items():
                self.h[key] = np.zeros_like(val)

        for key in params.keys():
            self.h[key] += grads[key] * grads[key] """求平方和，平方和中包含了每个元素 """
            params[key] -= self.lr * grads[key] / (np.sqrt(self.h[key]) + 1e-7)
```

代码测试：



参考资料：

https://tensorflow.google.cn/api_docs/python/tf/keras/optimizers/Adagrad?hl=en

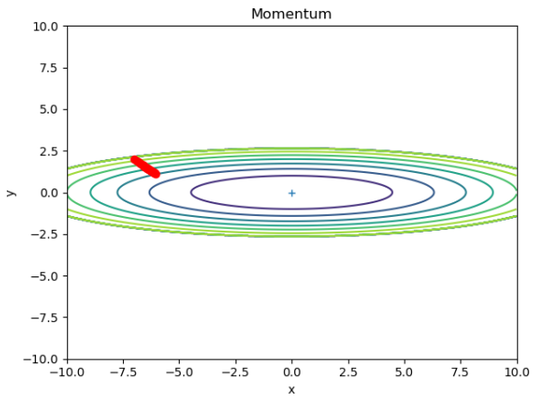
https://tensorflow.google.cn/api_docs/cc/class/tensorflow/ops/apply-adagrad?hl=en

6.5 RMSProp 的实现

从公式上看 AdaGrad 会迭代更新权重，这意味着会记录过去所有的平方和。可能会存在学习越深入更新的幅度会越小，甚至到最后几乎为 0 的情况。从上图可以看出确实出现了这种情况。

RMSProp 可以改善这种情况。

RMSProp 不是将过去的所有梯度都一视同仁的相加，而是逐渐的遗忘过去的梯度。



6.4 adam 的实现

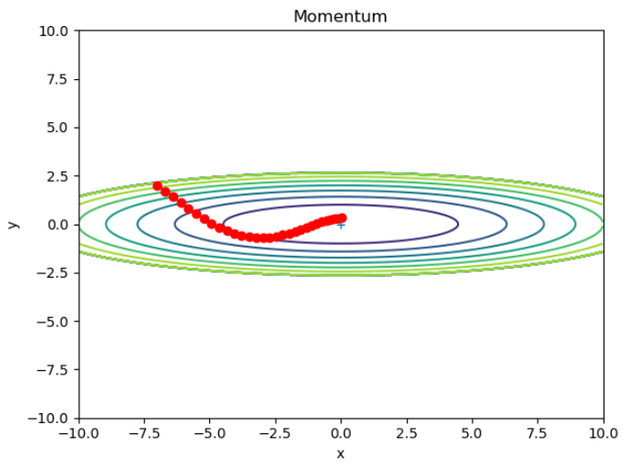
// 此处没有写代码和说明

Adam 的思路是将 adaGrad 和 momnentum 结合在一起。论文中 Adam 会设置 3 个参数：学习率α，一次动量系数β_1,二次动量系数β_2。

https://www.cnblogs.com/yifdu25/p/8183587.html

```python
class Adam:

    """
    Adam (http://arxiv.org/abs/1412.6980v8)
    """

    def __init__(self, lr=0.001, beta1=0.9, beta2=0.999):
        self.lr = lr
        self.beta1 = beta1
        self.beta2 = beta2
        self.iter = 0
        self.m = None
        self.v = None

    def update(self, params, grads):
        if self.m is None:
            self.m, self.v = {}, {}
            for key, val in params.items():
                self.m[key] = np.zeros_like(val)
                self.v[key] = np.zeros_like(val)

        self.iter += 1
        lr_t = self.lr * np.sqrt(1.0 - self.beta2**self.iter) / (1.0 - self.beta1**self.iter)

        for key in params.keys():
            #self.m[key] = self.beta1*self.m[key] + (1-self.beta1)*grads[key]
            #self.v[key] = self.beta2*self.v[key] + (1-self.beta2)*(grads[key]**2)
            self.m[key] += (1 - self.beta1) * (grads[key] - self.m[key])
            self.v[key] += (1 - self.beta2) * (grads[key]**2 - self.v[key])

            params[key] -= lr_t * self.m[key] / (np.sqrt(self.v[key]) + 1e-7)
```

测试代码：



参考资料：

https://tensorflow.google.cn/api_docs/python/tf/keras/optimizers/Adam?hl=en

6.5 构造神经网络

此处是用第五章的程序构建的神经。

```python
# -*- coding: utf-8 -*-
# @File  : multi_layer.py
# @Author: lizhen
# @Date  : 2020/2/4
# @Desc  : 多层神经网络

import numpy as np
from collections import OrderedDict
from src.common.layers import Sigmoid,Relu,Affine,SoftmaxWithLoss
from src.common.gradient import numerical_gradient


class MultiLayerNet:

    def __init__(self, input_size, hidden_size_list, output_size,
                 activation='relu', weight_init_std='relu', weight_decay_lambda=0):
        '''
        全连接的神经网络
        :param input_size: 输入大小, 784
        :param hidden_size_list: 隐含层的大小(e.g. [100, 100, 100])
        :param output_size: 输出层的大小, 10
        :param activation: 激活函数 'relu' or 'sigmoid'
        :param weight_init_std: 数据做标准化 (e.g. 0.01)
        :param weight_decay_lambda: 权值衰减系数
        '''
        self.input_size = input_size
        self.output_size = output_size
        self.hidden_size_list = hidden_size_list
        self.hidden_layer_num = len(hidden_size_list)
        self.weight_decay_lambda = weight_decay_lambda
        self.params = {}
```

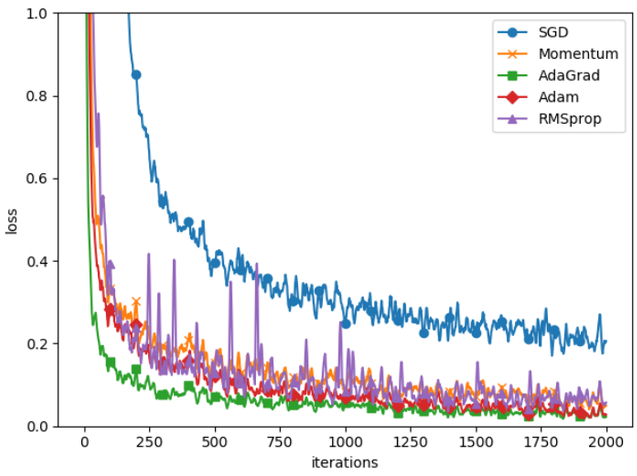
```
        # 初始化权重
        self.__init_weight(weight_init_std)

        # 激活层
        activation_layer = {'sigmoid': Sigmoid, 'relu': Relu}
        self.layers = OrderedDict()
        for idx in range(1, self.hidden_layer_num+1):
            self.layers['Affine' + str(idx)] = Affine(self.params['W' + str(idx)],
                                                      self.params['b' + str(idx)])
            self.layers['Activation_function' + str(idx)] = activation_layer[activation]()

        idx = self.hidden_layer_num + 1
        self.layers['Affine' + str(idx)] = Affine(self.params['W' + str(idx)],
            self.params['b' + str(idx)])

        self.last_layer = SoftmaxWithLoss()

    def __init_weight(self, weight_init_std):
        """
        初始化权重
        :param weight_init_std: 指定权重使用的标准差, {'relu','He','Sigmoid','xavier'}
        weight_init_std 可以使用的输入
        :return:
        """

        all_size_list = [self.input_size] + self.hidden_size_list + [self.output_size]
        for idx in range(1, len(all_size_list)):
            scale = weight_init_std
            if str(weight_init_std).lower() in ('relu', 'he'):# 使用 relu 的情况
                scale = np.sqrt(2.0 / all_size_list[idx - 1])
            elif str(weight_init_std).lower() in ('sigmoid', 'xavier'):
                scale = np.sqrt(1.0 / all_size_list[idx - 1])  # 使用 sigmoid 的情况

            self.params['W' + str(idx)] = scale * np.random.randn(all_size_list[idx-1], all_size_list[idx])
            self.params['b' + str(idx)] = np.zeros(all_size_list[idx])

    def predict(self, x):
        for layer in self.layers.values():
            x = layer.forward(x)

        return x

    def loss(self, x, t):
        """
        损失函数
        :param x: 输入数据
        :param t: 标签数据
        :return: 损失值
        """

        y = self.predict(x)

        weight_decay = 0
        for idx in range(1, self.hidden_layer_num + 2):
            W = self.params['W' + str(idx)]
            weight_decay += 0.5 * self.weight_decay_lambda * np.sum(W ** 2)

        return self.last_layer.forward(y, t) + weight_decay

    def accuracy(self, x, t):
        y = self.predict(x)
        y = np.argmax(y, axis=1)
        if t.ndim != 1 : t = np.argmax(t, axis=1)

        accuracy = np.sum(y == t) / float(x.shape[0])
        return accuracy

    def numerical_gradient(self, x, t):
        '''
        求数值微分
        :param x: 输入的数据
        :param t: 标签数据
        :return: 返回每一层的梯度
        '''

        loss_W = lambda W: self.loss(x, t)

        grads = {}
        for idx in range(1, self.hidden_layer_num+2):
            grads['W' + str(idx)] = numerical_gradient(loss_W, self.params['W' + str(idx)])
            grads['b' + str(idx)] = numerical_gradient(loss_W, self.params['b' + str(idx)])

        return grads

    def gradient(self, x, t):
        '''
        寻找每层的梯度
        :param x: 输入数据
        :param t: 标签
        :return:返回每层的梯度变量
        '''
        # forward
        self.loss(x, t)

        # backward
        dout = 1
        dout = self.last_layer.backward(dout)

        layers = list(self.layers.values())
        layers.reverse()
        for layer in layers:
            dout = layer.backward(dout)


        grads = {}
        for idx in range(1, self.hidden_layer_num+2):
            grads['W' + str(idx)] = self.layers['Affine' + str(idx)].dW + self.weight_decay_lambda * self.layers['Affine' + str(idx)].W
            grads['b' + str(idx)] = self.layers['Affine' + str(idx)].db
```

```
        return grads
```

6.6 opt 的对比（基于 mnist 数据集）

P.S：这里面把 RMSprop 也放进来做参考了。

```
# coding: utf-8
#  opt 的对比（基于mnist数据集）

import matplotlib.pyplot as plt
from src.datasets.mnist import load_mnist
from src.common.util import smooth_curve
from src.common.multi_layer_net import MultiLayerNet
from src.common.optimizer import *


# 0:MNIST数据集
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True)

train_size = x_train.shape[0]
batch_size = 128
max_iterations = 2000


# 1:设置优化器
optimizers = {}
optimizers['SGD'] = SGD()
optimizers['Momentum'] = Momentum()
optimizers['AdaGrad'] = AdaGrad()
optimizers['Adam'] = Adam()
optimizers['RMSprop'] = RMSprop()

networks = {}
train_loss = {}
for key in optimizers.keys():
    networks[key] = MultiLayerNet(
        input_size=784, hidden_size_list=[100, 100, 100, 100],
        output_size=10)
    train_loss[key] = []


# 2:训练:
for i in range(max_iterations):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    for key in optimizers.keys():
        grads = networks[key].gradient(x_batch, t_batch)
        optimizers[key].update(networks[key].params, grads)

        loss = networks[key].loss(x_batch, t_batch)
        train_loss[key].append(loss)

    if i % 100 == 0:
        print( "========" + "iteration:" + str(i) + "========")
        for key in optimizers.keys():
            loss = networks[key].loss(x_batch, t_batch)
            print(key + ":" + str(loss))


# 3.绘图
markers = {"SGD": "o", "Momentum": "x", "AdaGrad": "s", "Adam": "D","RMSprop":"^"}
x = np.arange(max_iterations)
for key in optimizers.keys():
    plt.plot(x, smooth_curve(train_loss[key]), marker=markers[key], markevery=100, label=key)
plt.xlabel("iterations")
plt.ylabel("loss")
plt.ylim(0, 1)
plt.legend()
plt.show()
```



4.6 总结

在网上找到了一些资料：

https://www.cnblogs.com/itmorn/p/11123789.html#ct2

> *本篇会做实现两种解决过拟合的方法：*
> 1. *实现 DroupOut,并使用 mnist 做测试，观察使用权值衰减和没有使用的区别。*
> 2. *实现权值衰减，使用 mnist 做测试，观察使用权值衰减和没有使用的区别。*

第五篇：解决过拟合的方法

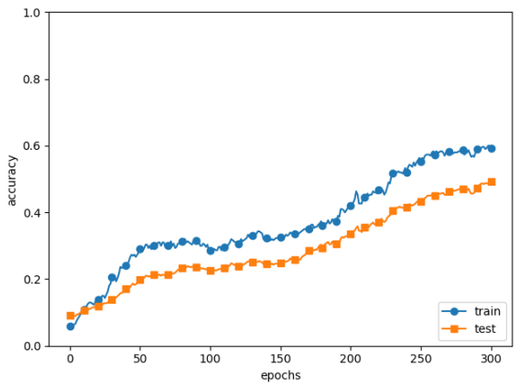
七、解决过拟合的方法

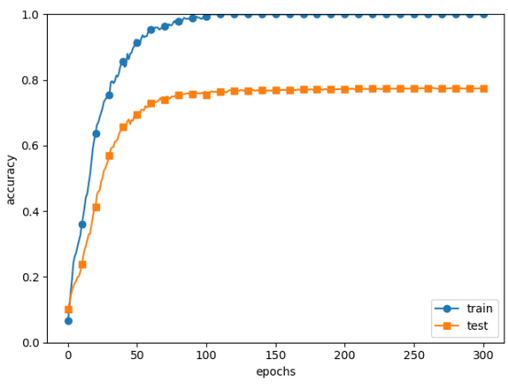
7.1 DropOut 的实现

DroupOut 作为抑制过拟合的一种方法，会**在每次训练的时候随机的将一些神经元失活**。

```
class Dropout:
    """
    http://arxiv.org/abs/1207.0580
    """
    def __init__(self, dropout_ratio=0.5):
        self.dropout_ratio = dropout_ratio
        self.mask = None

    def forward(self, x, train_flg=True):
        """
        正向传播时，随机标记 mask，这样看起来就像是在反馈的时候失活了
        """
        if train_flg:
            self.mask = np.random.rand(*x.shape) > self.dropout_ratio
            return x * self.mask
        else:
            return x * (1.0 - self.dropout_ratio)

    def backward(self, dout):
        '''
        return dout * self.mask
```

左图是使用了 dropout,右图是没有使用 dropout.之所以会出现这种情况，是因为 dropout 会是得一些神经元失活





7.2 权值衰减的实现

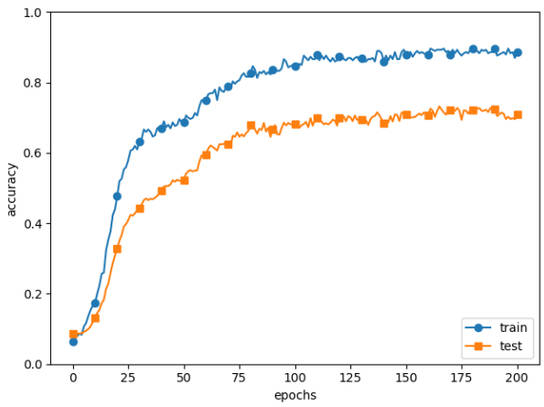
权值衰减也是抑制过拟合的一种方法。

这种方法通过在训练过程中的权值进行“惩罚”，从而抑制拟合。

本案例在损失函数获取以后，更新权重。在更新权重的时候加上 L2 范数，这样便可以抑制权重变大。L2 范数如下，lamda 是权值衰减系数。

$$\boldsymbol{W} = \frac{1}{2}\lambda \boldsymbol{W}^2$$

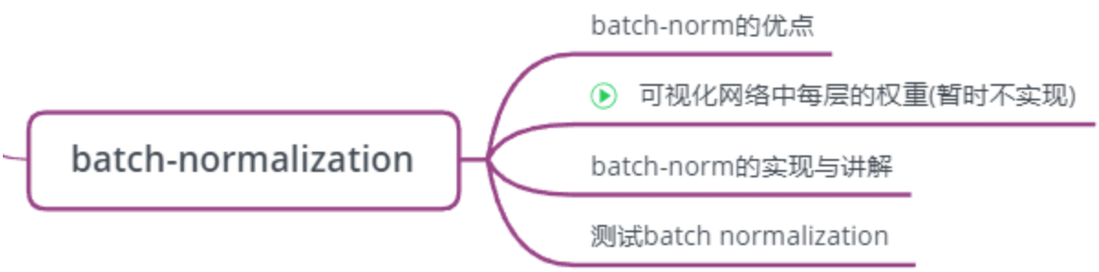
```
def loss(self, x, t):
    """
    损失函数
    :param x: 输入数据
    :param t: 标签数据
    :return: 损失值
    """

    y = self.predict(x)

    weight_decay = 0
    for idx in range(1, self.hidden_layer_num + 2):
        W = self.params['W' + str(idx)]
        weight_decay += 0.5 * self.weight_decay_lambda * np.sum(W ** 2)

    return self.last_layer.forward(y, t) + weight_decay
```



```python
# coding: utf-8

import numpy as np
import matplotlib.pyplot as plt
from src.datasets.mnist import load_mnist
from src.common.multi_layer_net import MultiLayerNet
from src.common.optimizer import SGD

(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True)

# 只是选取300个样本
x_train = x_train[:300]
t_train = t_train[:300]

# weight decay系数
#weight_decay_lambda = 0
weight_decay_lambda = 0.1
# ==========================================

network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100, 100], output_size=10,
                        weight_decay_lambda=weight_decay_lambda)
optimizer = SGD(lr=0.01)

max_epochs = 201
train_size = x_train.shape[0]
batch_size = 100

train_loss_list = []
train_acc_list = []
test_acc_list = []

iter_per_epoch = max(train_size / batch_size, 1)
epoch_cnt = 0

for i in range(1000000000):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    grads = network.gradient(x_batch, t_batch)
    optimizer.update(network.params, grads)

    if i % iter_per_epoch == 0:
        train_acc = network.accuracy(x_train, t_train)
        test_acc = network.accuracy(x_test, t_test)
        train_acc_list.append(train_acc)
        test_acc_list.append(test_acc)

        print("epoch:" + str(epoch_cnt) + ", train acc:" + str(train_acc) + ", test acc:" + str(test_acc))

        epoch_cnt += 1
        if epoch_cnt >= max_epochs:
            break


# 3.绘制
markers = {'train': 'o', 'test': 's'}
x = np.arange(max_epochs)
plt.plot(x, train_acc_list, marker='o', label='train', markevery=10)
plt.plot(x, test_acc_list, marker='s', label='test', markevery=10)
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
plt.legend(loc='lower right')
plt.show()
```

第六篇： Batch Normalization

> *本章主要介绍batch-nom 的实现*
>
> *首先介绍batch-nom 的优势*
>
> *然后本~~来打算可视化网络权重分布图（有点费时间~~，暂时不写了）*
>
> *最后实现batch-nomr，用mnist 做测试*



八、 Batch Normalization

Batch-normalization 是可以使各个层的激活值分布适当，从而顺利进行“学习”。BatchNorm 有已下的优点:

1. 可以使得学习快速收敛
2. 不再依赖初始值
3. 抑制过拟合（权重的值抑制）

~~3.1观察每层的数值分布~~

3.2 batch normalization 的实现

batch-norm 是以训练的 mini-batch 为单位，按照 mini-batch 进行标准化，使得数据分布的均值为$\boldsymbol{\mu}$=0，方差为$\boldsymbol{\sigma}^2$=1。

Batch-norm 的思路是调整各个层的激活值的分布，使得其拥有适当的广度，是神经网络对数据分布进行标准化的层。

举例：对 mini-batch=m 的输入样本$B=\{\boldsymbol{x_i},\ \boldsymbol{x}_2,\ \ldots\ \boldsymbol{x_m}\}$,求均值$\boldsymbol{\mu}$和方差$\boldsymbol{\sigma}^2$后，将 B 修改成符合标准正态分布的数据$\hat{B}=\{\hat{\boldsymbol{x}}_{\boldsymbol{i}},\ \hat{\boldsymbol{x}}_2,\ \ldots\ \hat{\boldsymbol{x}}_{\boldsymbol{m}}\}$。然后对标准化的数据进行缩放和平移。

因此，数据 forwad 处理的过程如下:

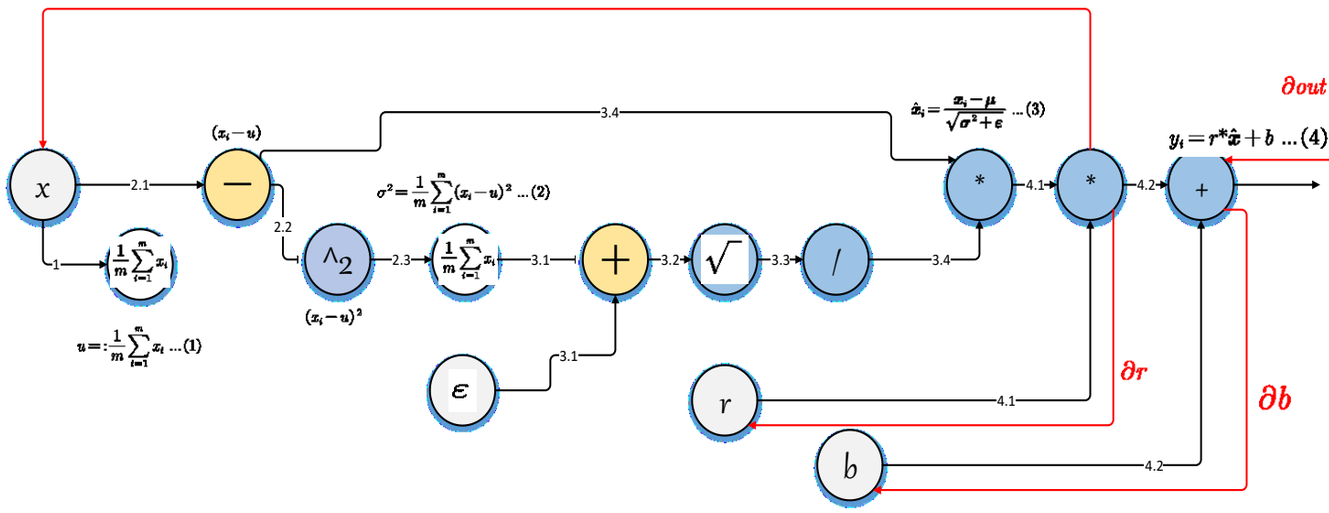
$$\boldsymbol{\mu}=\frac{1}{\boldsymbol{m}}\sum_{i=1}^{m}\boldsymbol{x}_i\ (\text{求平均值})\ \ldots 1$$

$$\boldsymbol{\sigma}^2=\frac{1}{\boldsymbol{m}}\sum_{i=1}^{m}(\boldsymbol{x}_2-\boldsymbol{\mu})^2\ (\text{根据平均值求方差})\ \ldots 2$$

$$\hat{\boldsymbol{x}}_{\boldsymbol{i}}=\frac{\boldsymbol{x}_i-\boldsymbol{\mu}}{\sqrt{\boldsymbol{\sigma}^2+\boldsymbol{\varepsilon}}}\ (\text{修改成标准正态分布})\ \ldots 3$$

$$y_i=\gamma\hat{x}_i+b\ (\text{对数据进行缩放和平移，初始值}\gamma=1,\ b=0)\ \ldots(4)$$

其 backward 流程本质上是 forwad 的导数，可以使用计算图进行推导:

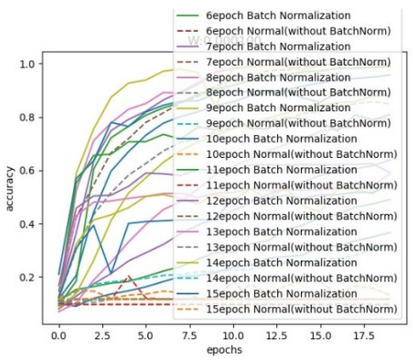



```
class BatchNormalization:
    """
    http://arxiv.org/abs/1502.03167
    """
    def __init__(self, gamma, beta, momentum=0.9, running_mean=None, running_var=None):
        self.gamma = gamma
        self.beta = beta
        self.momentum = momentum
        self.input_shape = None  # 转换层为 4D, 全连接层为 2D

        # 平均值和方差
        self.running_mean = running_mean
        self.running_var = running_var

        # backward时使用的中间结果
        self.batch_size = None
        self.xc = None
        self.std = None
        self.dgamma = None
        self.dbeta = None

    def forward(self, x, train_flg=True):
        self.input_shape = x.shape
        if x.ndim != 2:
            N, C, H, W = x.shape
            x = x.reshape(N, -1)

        out = self.__forward(x, train_flg)

        return out.reshape(*self.input_shape)

    def __forward(self, x, train_flg):
        if self.running_mean is None:
            N, D = x.shape
            self.running_mean = np.zeros(D)
            self.running_var = np.zeros(D)

        if train_flg: # batch-norm
            mu = x.mean(axis=0)
            xc = x - mu
            var = np.mean(xc ** 2, axis=0)
            std = np.sqrt(var + 10e-7)
            xn = xc / std

            self.batch_size = x.shape[0]
            self.xc = xc
            self.xn = xn
            self.std = std
            self.running_mean = self.momentum * self.running_mean + (1 - self.momentum) * mu
            self.running_var = self.momentum * self.running_var + (1 - self.momentum) * var
        else:
            xc = x - self.running_mean
            xn = xc / ((np.sqrt(self.running_var + 10e-7)))

        out = self.gamma * xn + self.beta
        return out

    def backward(self, dout):
        if dout.ndim != 2:
            N, C, H, W = dout.shape
            dout = dout.reshape(N, -1)

        dx = self.__backward(dout)

        dx = dx.reshape(*self.input_shape)
        return dx

    def __backward(self, dout):
        dbeta = dout.sum(axis=0)
        dgamma = np.sum(self.xn * dout, axis=0)
        dxn = self.gamma * dout
        dxc = dxn / self.std
        dstd = -np.sum((dxn * self.xc) / (self.std * self.std), axis=0)
        dvar = 0.5 * dstd / self.std
        dxc += (2.0 / self.batch_size) * self.xc * dvar
        dmu = np.sum(dxc, axis=0)
        dx = dxc - dmu / self.batch_size
```

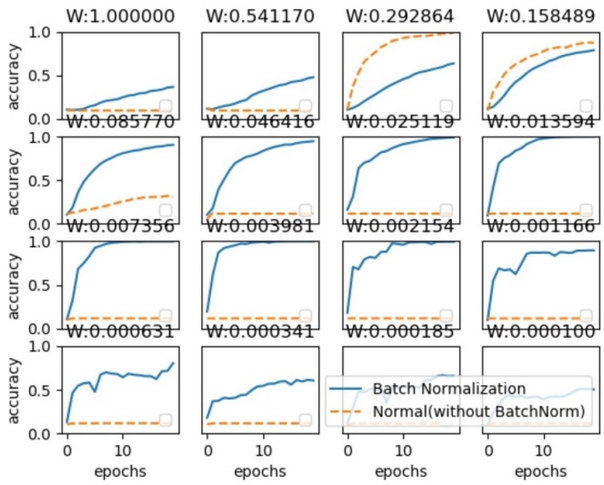
```
        self.dgamma = dgamma
        self.dbeta = dbeta

        return dx
```

3.3 测试 batch normalization(基于 mnist 数据集)

该程序选了 1000 个样本作为训练集，使用了 16 个 epoch,将使用 bn 层和没有使用 bn 层网络的权重变化进行了对比，如下图所示：



下面这张图和上面的一样，



2020 年 2 月 7 日星期五

第七篇：卷积层和池化层的实现

本篇概要：

本篇分为 4 个章节介绍：

第九章：卷积层的实现。本章首先会讲解 convOp，在实现 convOp 之前引入 im2col。最后讲解 conv 的 backward，并做单元测试。

第十章：池化层的实现。本章首先讲解 poolOP,然后是 pool 的实现，最后是单元测试。

第十一章：卷积神经网络的实现。首先会实现卷积神经网络

九、卷积层的实现

在之前的神经网络中一直在使用全连接层。然而全连接层存在有两个问题：

将数据的形状忽视了，比如输入图像是，图像一般有 W，H，C 的 3D 形状，包含有空间信息：比如相邻元素的相似值、RGB 各个通道之间的关联性。但如果使用全连接就需要把数据转换成 1D 数据。**这也就是为什么在第七章之前的 mnist 数据集的读取都是 28*28*1，因为无法利用图像的空间信息**。

9.1 卷积层的运算

卷积的 forward

卷积的计算过程网上的资料已经做够好了，没必要自己再写一遍。只把资料搬运到这里：

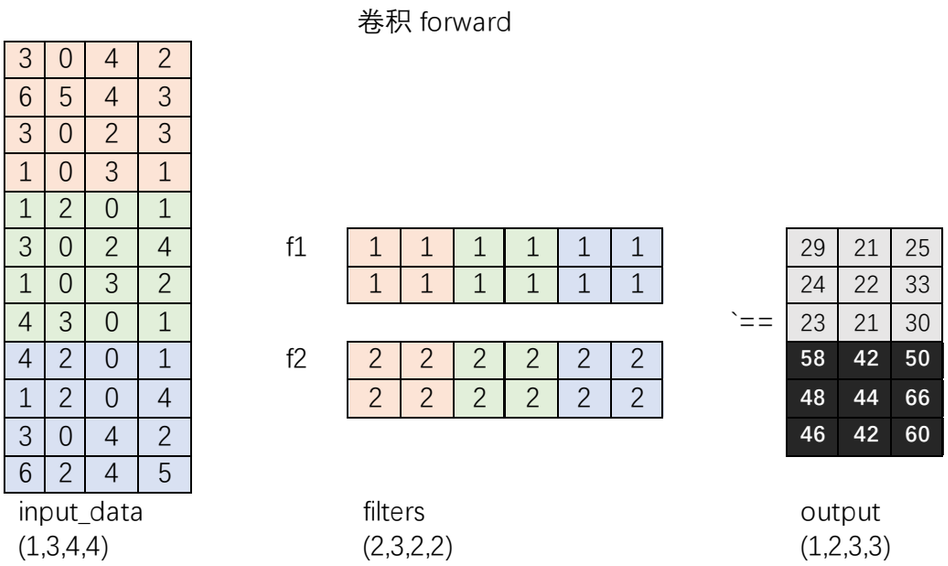
http://deeplearning.net/software/theano_versions/dev/tutorial/conv_arithmetic.html#transposed-convolution-arithmetic

https://www.zhihu.com/question/43609045

https://blog.csdn.net/weixin_44106928/article/details/103079668

这里总结一下有 padding\stride 的卷积操作：

卷积 forward



假设，输入大小为（H,W,C）,fileter 大小为（FH，FW,C）*N；padding=P, stride=S, 卷积后的形状为(OH,OW,OC)

$$OH = \frac{H + 2P - FH}{S} + 1$$

$$OW = \frac{W + 2P - FW}{S} + 1$$

$$OC = N$$

```
def forward(self, x):
    '''
    使用 im2col 将输入的 x 转换成 2D 矩阵
    然后 y= w*x+b 以矩阵的形式完成
    最后返回 y
    :param x: x 为 4D tensor，输入数据
    :return: out=w*x+b
    '''
    FN, C, FH, FW = self.W.shape
    N, C, H, W = x.shape
    out_h = 1 + int((H + 2 * self.pad - FH) / self.stride)
    out_w = 1 + int((W + 2 * self.pad - FW) / self.stride)

    col = im2col(x, FH, FW, self.stride, self.pad)
    col_W = self.W.reshape(FN, -1).T
    print("col.shape=%s"%str(col.shape))
    print("col_W.shape=%s"%str(col_W.shape))

    out = np.dot(col, col_W)
    print("out.shape=%s"%str(out.shape))
    out=out+self.b
    out = out.reshape(N, out_h, out_w, -1).transpose(0, 3, 1, 2)

    self.x = x
    self.col = col
    self.col_W = col_W

    return out
```

卷积的 backward

https://zhuanlan.zhihu.com/p/33802329

卷积的 backward 是对卷积的求导。

```
def backward(self, dout):
    '''
    反馈过程中也需要将 2D 矩阵转换为 4D tensor
    :param dout: 梯度差
    :return:
    '''
    FN, C, FH, FW = self.W.shape
    dout = dout.transpose(0, 2, 3, 1).reshape(-1, FN) # NCHW

    self.db = np.sum(dout, axis=0)# NHWC， 求和
    self.dW = np.dot(self.col.T, dout) # 点乘 w
    self.dW = self.dW.transpose(1, 0).reshape(FN, C, FH, FW)

    dcol = np.dot(dout, self.col_W.T)
    dx = col2im(dcol, self.x.shape, FH, FW, self.stride, self.pad)

    return dx
```

9.2 引入 im2col 概念

再讲卷积的实现之前，首先抛出一个问题：如果按照上述的卷积方式计算，是否会影响性能？

答案是肯定会受影响的。

因此，我们需要向优化一下 conv 的计算方式.

按照“以空间换时间”的思想，我们可以做一些优化，使得在 conv 和 pool 的时候运算速度加快。

首先，我们知道 Numpy 对大型矩阵的运算是有做优化的，这个特点我们应该好好利用；

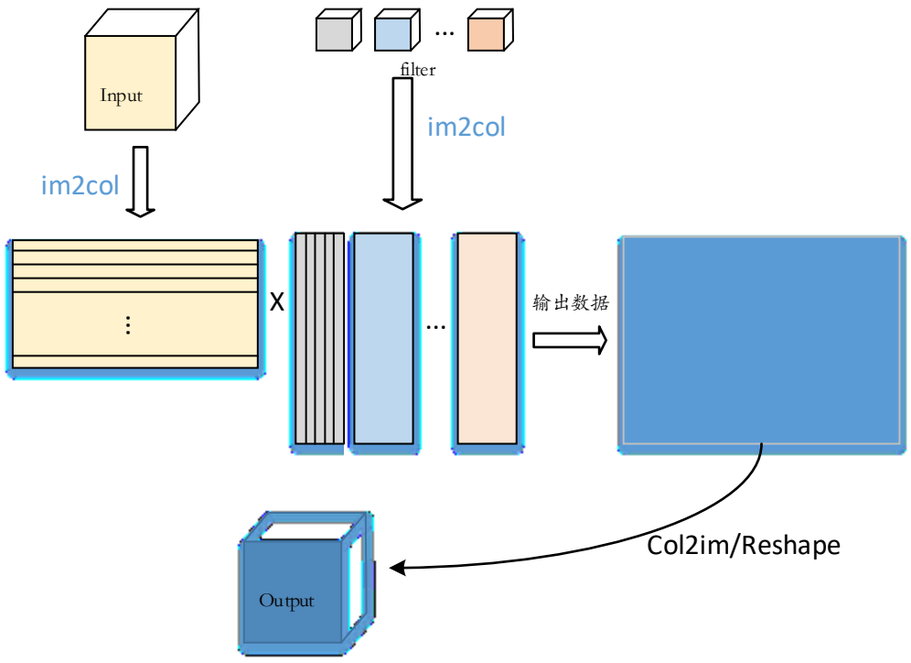
其次，我们知道 Numpy 在做多个嵌套的 for 循环的时候，O(n)会很大；应该避免做多个 for 循环；

因此，要是将 4D 的卷积运算转换成 2D 的矩阵乘法就会好很多；filter 也可以变成 2D 的数组；

Im2col 便是将 4D 数据转换成 2D 矩阵的函数。

该函数大致的思路是：filter 按照行列展开成一个 2D 矩阵即可，input_data 按照计算的单元重新组合。因此需要写一个函数将图像转换成 2D 矩阵，该函数可以将图像展开成适合与滤波去做乘法的矩阵。

展开和计算的流程如下：



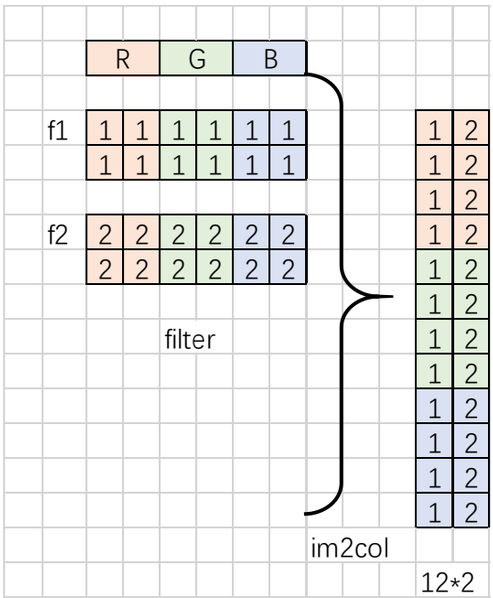
9.3 单元测试 im2col

对 filter 计算有影响的因素有 input_data,filter_h，filter_w，stride, padding；im2col 会应该根据以上的因因素展开 input_data,展开后的 input_data 一定是比之前要大的；

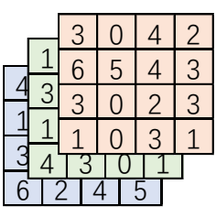
我们可以尝试计算一下 input_data 展开后的数据形状：

假设，输入数据为 4*4*3 大小的 tensor;filter 有两个为 2*（2*2*3），filter_h=2，filter_w=2,stride=1, padding=0；这里可以计算出展开以后的大小：

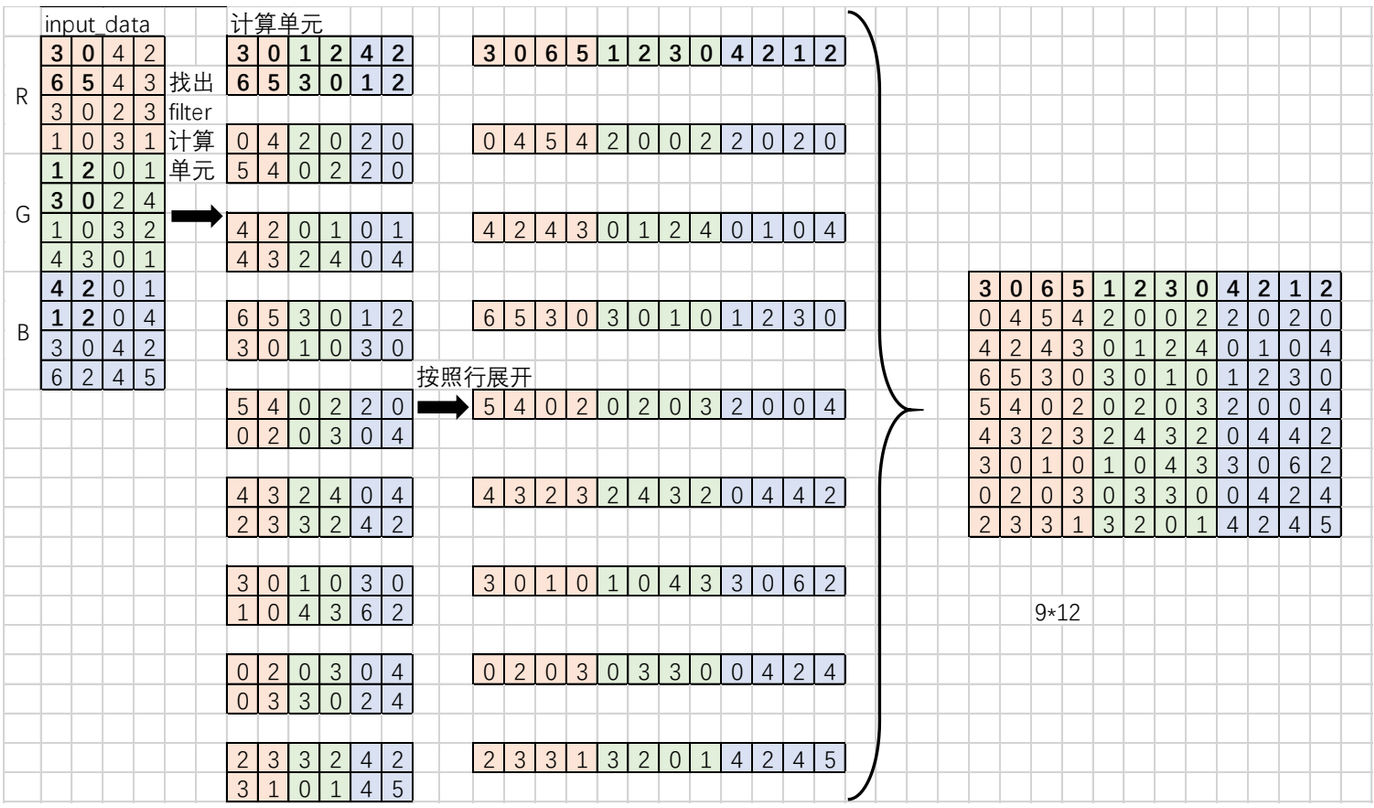
Filter 为有两个，分别为 f1 和 f2;shape=（2*2*3）,按照行展开成 2D 的矩阵以后如下图所示：



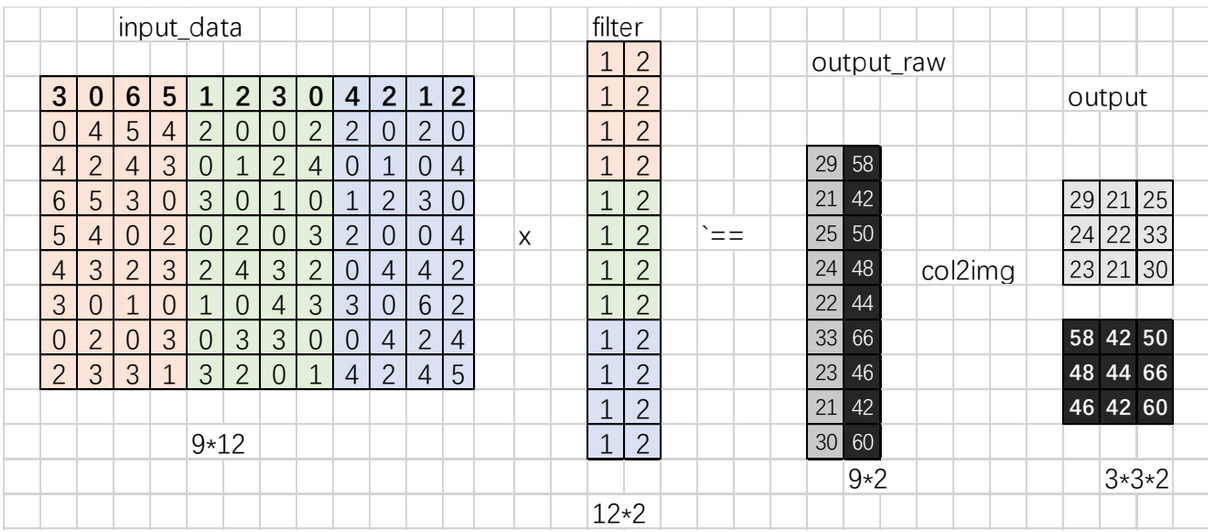
Input_data 为 4*4*3 的 tensor,如下图所示：



Input_data 首先会找出 filter 对应的计算单元，这些还是需要 padding\stride\filter_w\filter_h 相关，找出计算的单元以后，按照行展开。最后得到的数据便是 im2col 的结果:



Input_data 和 filter 这样展开以后，卷积计算就可以按照矩阵乘法的方式计算，避免了重复的 for 循环。如下图所示，黑色和灰色区域是计算的结果。不必担心矩阵过大是否会影响计算速度，Numpy 对大规模矩阵乘法内部有优化加速，这样展开以后恰恰也能充分的利用 numpy 的特性。



Im2col 的实现:

```
def im2col(input_data, filter_h, filter_w, stride=1, pad=0):
    '''

    :param input_data: 输入数据由4维数组组成 (N,C, H,W)
    :param filter_h:   filer 的高
    :param filter_w:   filter的宽
    :param stride:     stride
    :param pad:        padding
    :return:           2D 矩阵
    '''
    # 计算输出的大小
    N, C, H, W = input_data.shape
    out_h = (H + 2*pad - filter_h)//stride + 1
    out_w = (W + 2*pad - filter_w)//stride + 1
    # padding
    img = np.pad(input_data, [(0,0), (0,0), (pad, pad), (pad, pad)], 'constant')
    col = np.zeros((N, C, filter_h, filter_w, out_h, out_w))
    # 计算单元
    for y in range(filter_h):
        y_max = y + stride*out_h
        for x in range(filter_w):
            x_max = x + stride*out_w
```

```
            col[:, :, y, x, :, :] = img[:, :, y:y_max:stride, x:x_max:stride]
    # 重新排列
    col = col.transpose(0, 4, 5, 1, 2, 3).reshape(N*out_h*out_w, -1)
    return col
```

测试代码：

```
# -*- coding: utf-8 -*-
# @File  : test_im2col.py
# @Author: lizhen
# @Date  : 2020/2/14
# @Desc  : 测试 im2col
import numpy as np

from src.common.util import im2col,col2im

if __name__ == '__main__':
    raw_data = [3, 0, 4, 2,
                6, 5, 4, 3,
                3, 0, 2, 3,
                1, 0, 3, 1,

                1, 2, 0, 1,
                3, 0, 2, 4,
                1, 0, 3, 2,
                4, 3, 0, 1,

                4, 2, 0, 1,
                1, 2, 0, 4,
                3, 0, 4, 2,
                6, 2, 4, 5
    ]

    input_data = np.array(raw_data)
    input_data = input_data.reshape(1,3,4,4)
    print(input_data.shape)
    col1 = im2col(input_data=input_data,filter_h=2,filter_w=2,stride=1,pad=0)#input_data, filter_h, filter_w, stride=1, pad=0
    print(col1)
```

========输出：可以发现和上面的绘图的结果是一致的 =====

(1, 3, 4, 4)

[[3. 0. 6. 5. 1. 2. 3. 0. 4. 2. 1. 2.]

[0. 4. 5. 4. 2. 0. 0. 2. 2. 0. 2. 0.]

[4. 2. 4. 3. 0. 1. 2. 4. 0. 1. 0. 4.]

[6. 5. 3. 0. 3. 0. 1. 0. 1. 2. 3. 0.]

[5. 4. 0. 2. 0. 2. 0. 3. 2. 0. 0. 4.]

[4. 3. 2. 3. 2. 4. 3. 2. 0. 4. 4. 2.]

[3. 0. 1. 0. 1. 0. 4. 3. 3. 0. 6. 2.]

[0. 2. 0. 3. 0. 3. 3. 0. 0. 4. 2. 4.]

[2. 3. 3. 1. 3. 2. 0. 1. 4. 2. 4. 5.]]

9.3 卷积操作的实现

卷积操作也需要实现 forward 和 backward 函数。

Forward 函数中用到了 9.1\9.2 的 im2col

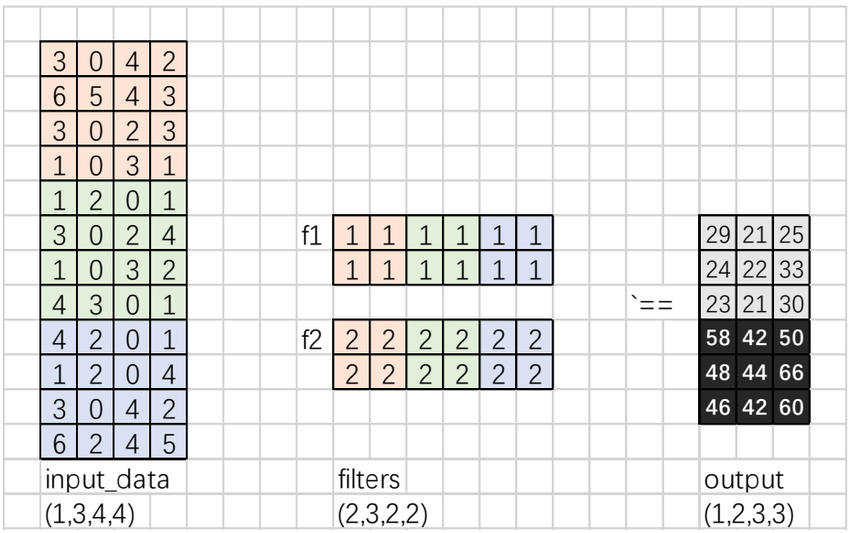
```
class Convolution:
    def __init__(self, W, b, stride=1, pad=0):
        '''
        conv 的构造函数
        :param W: 2D 矩阵
        :param b:
        :param stride:
        :param pad:
        '''
        self.W = W
        self.b = b
        self.stride = stride
        self.pad = pad

        # 中间结果 (backward 的时候使用)
        self.x = None
        self.col = None
        self.col_W = None

        # 权重的梯度/偏置的梯度
        self.dW = None
        self.db = None
```

```
    def forward(self, x):
        '''
        使用 im2col 将输入的 x 转换成 2D 矩阵
        然后 y= w*x+b 以矩阵的形式完成
        最后返回 y
        :param x: x 为 4D tensor，输入数据
        :return: out=w*x+b
        '''
        FN, C, FH, FW = self.W.shape
        N, C, H, W = x.shape
        out_h = 1 + int((H + 2 * self.pad - FH) / self.stride)
        out_w = 1 + int((W + 2 * self.pad - FW) / self.stride)

        col = im2col(x, FH, FW, self.stride, self.pad)
        col_W = self.W.reshape(FN, -1).T

        out = np.dot(col, col_W) + self.b
        out = out.reshape(N, out_h, out_w, -1).transpose(0, 3, 1, 2)

        self.x = x
        self.col = col
        self.col_W = col_W

        return out

    def backward(self, dout):
        '''
        反馈过程中也需要将 2D 矩阵转换为 4D tensor
        :param dout：梯度差
        :return:
        '''
        FN, C, FH, FW = self.W.shape
        dout = dout.transpose(0, 2, 3, 1).reshape(-1, FN)

        self.db = np.sum(dout, axis=0)
        self.dW = np.dot(self.col.T, dout)
        self.dW = self.dW.transpose(1, 0).reshape(FN, C, FH, FW)

        dcol = np.dot(dout, self.col_W.T)
        dx = col2im(dcol, self.x.shape, FH, FW, self.stride, self.pad)

        return dx
```

9.4 单元测试卷积操作

输入：input_data\filters

输出：output



测试代码：

```
# -*- coding: utf-8 -*-
# @File  : test_im2col.py
# @Author: lizhen
# @Date  : 2020/2/14
# @Desc  : 测试 im2col
import numpy as np

from src.common.util import im2col,col2im
from src.common.layers import Convolution


if __name__ == '__main__':
    raw_data = [3, 0, 4, 2,
                6, 5, 4, 3,
                3, 0, 2, 3,
                1, 0, 3, 1,

                1, 2, 0, 1,
                3, 0, 2, 4,
```

```
            1, 0, 3, 2,
            4, 3, 0, 1,

            4, 2, 0, 1,
            1, 2, 0, 4,
            3, 0, 4, 2,
            6, 2, 4, 5
]

raw_filter=[
    1,    1,    1,    1,    1,    1,
    1,    1,    1,    1,    1,    1,
    2,    2,    2,    2,    2,    2,
    2,    2,    2,    2,    2,    2,

]



input_data = np.array(raw_data)
filter_data = np.array(raw_filter)

x = input_data.reshape(1,3,4,4)#NCHW
W = filter_data.reshape(2,3,2,2) #NHWC
b = np.zeros(2)
# b = b.reshape((2,1))
# col = im2col(input_data=x,filter_h=2,filter_w=2,stride=1,pad=0)#input_data, filter_h, filter_w, stride=1, pad=0
# print(col)

print("input_data.shape=%s"%str(input_data.shape))
print("W.shape=%s"%str(W.shape))
print("b.shape=%s"%str(b.shape))
conv = Convolution(W,b) # def __init__(self, W, b, stride=1, pad=0)
out = conv.forward(x)
print("bout.shape=%s"%str(out.shape))
print(out)
```

Conv 的输出结果，与上图的结果一致。

```
input_data.shape=(48,)
W.shape=(2, 3, 2, 2)
b.shape=(2,)
col.shape=(9, 12)
col_W.shape=(12, 2)
out.shape=(9, 2)
bout.shape=(1, 2, 3, 3)
[[[[29. 21. 25.]
   [24. 22. 33.]
   [23. 21. 30.]]

  [[58. 42. 50.]
   [48. 44. 66.]
   [46. 42. 60.]]]]
```

十、池化层的实现

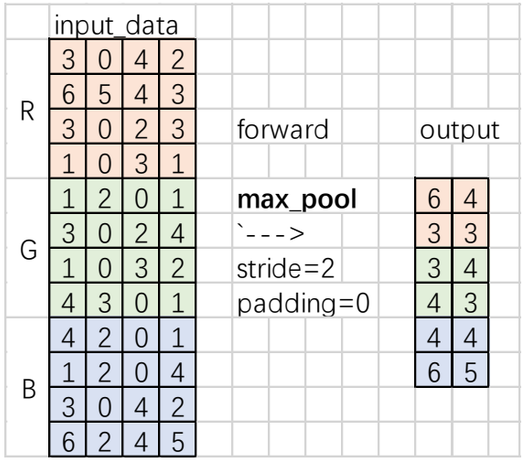
10.1 池化层的运算

池化层的 forward

Pool 分为三类 mean-pool, max-pool 和 min-pool， 本章只讨论 max-pool

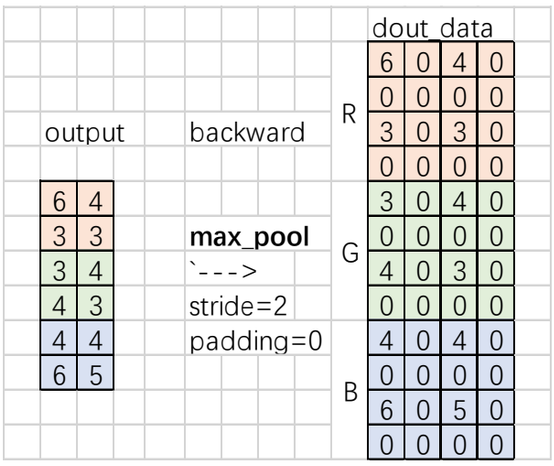
以下是 forwad 的运算：

https://blog.csdn.net/nanhuaibeian/article/details/100664570



池化层的 backward 的运算

Max-pool 的反传是将原来的单元扩大 stride_h*stride_w，其余的地方填充 0

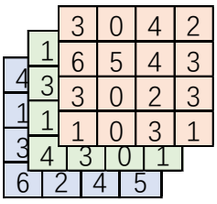


10.2 池化层的实现

```
class Pooling:
    def __init__(self, pool_h, pool_w, stride=1, pad=0):
        self.pool_h = pool_h
        self.pool_w = pool_w
        self.stride = stride
        self.pad = pad

        self.x = None
        self.arg_max = None

    def forward(self, x):
        N, C, H, W = x.shape
        out_h = int(1 + (H - self.pool_h) / self.stride)
        out_w = int(1 + (W - self.pool_w) / self.stride)

        col = im2col(x, self.pool_h, self.pool_w, self.stride, self.pad)
        col = col.reshape(-1, self.pool_h * self.pool_w)

        arg_max = np.argmax(col, axis=1)
        out = np.max(col, axis=1)
        out = out.reshape(N, out_h, out_w, C).transpose(0, 3, 1, 2)

        self.x = x
        self.arg_max = arg_max

        return out

    def backward(self, dout):
        dout = dout.transpose(0, 2, 3, 1)

        pool_size = self.pool_h * self.pool_w
        dmax = np.zeros((dout.size, pool_size))
        dmax[np.arange(self.arg_max.size), self.arg_max.flatten()] = dout.flatten()
        dmax = dmax.reshape(dout.shape + (pool_size,))

        dcol = dmax.reshape(dmax.shape[0] * dmax.shape[1] * dmax.shape[2], -1)
        dx = col2im(dcol, self.x.shape, self.pool_h, self.pool_w, self.stride, self.pad)

        return dx
```

10.3 pool 单元测试

测试的数据如下：



im2col 以后的数据：

	input_data			
R	3	0	4	2
	6	5	4	3
	3	0	2	3
	1	0	3	1
G	1	2	0	1
	3	0	2	4
	1	0	3	2
	4	3	0	1
B	4	2	0	1
	1	2	0	4
	3	0	4	2
	6	2	4	5

Maxpool 以后的数据：

	input_data						
R	3	0	4	2			
	6	5	4	3			
	3	0	2	3			output
	1	0	3	1			
G	1	2	0	1	**max_pool**	6	4
	3	0	2	4	`--->	3	3
	1	0	3	2	stride=2	3	4
	4	3	0	1	padding=0	4	3
B	4	2	0	1		4	4
	1	2	0	4		6	5
	3	0	4	2			
	6	2	4	5			

测试程序：

```
# -*- coding: utf-8 -*-
# @File  : test_im2col.py
# @Author: lizhen
# @Date  : 2020/2/14
# @Desc  : 测试 im2col
import numpy as np

from src.common.util import im2col,col2im
from src.common.layers import Convolution,Pooling


if __name__ == '__main__':
    raw_data = [3, 0, 4, 2,
                6, 5, 4, 3,
                3, 0, 2, 3,
                1, 0, 3, 1,

                1, 2, 0, 1,
                3, 0, 2, 4,
                1, 0, 3, 2,
                4, 3, 0, 1,

                4, 2, 0, 1,
                1, 2, 0, 4,
                3, 0, 4, 2,
                6, 2, 4, 5
    ]

    raw_filter=[
        1,    1,    1,    1,    1,    1,
        1,    1,    1,    1,    1,    1,
        2,    2,    2,    2,    2,    2,
```

```
    2,    2,    2,    2,    2,    2,
]


input_data = np.array(raw_data)
filter_data = np.array(raw_filter)

x = input_data.reshape(1,3,4,4)#NCHW
W = filter_data.reshape(2,3,2,2) #NHWC
b = np.zeros(2)
# b = b.reshape((2,1))
# col1 = im2col(input_data=x,filter_h=2,filter_w=2,stride=1,pad=0)#input_data, filter_h, filter_w, stride=1, pad=0
# print(col1)

# print("input_data.shape=%s"%str(input_data.shape))
# print("W.shape=%s"%str(W.shape))
# print("b.shape=%s"%str(b.shape))
# conv = Convolution(W,b) # def __init__(self, W, b, stride=1, pad=0)
# out = conv.forward(x)
# print("bout.shape=%s"%str(out.shape))
# print(out)

print("============")
pool=Pooling( pool_h=2, pool_w=2, stride=2, pad=0)
out = pool.forward(x)
print(out.shape)
print(out)
```

对应输出：

```
====================
(1, 3, 2, 2)
[[[[6. 4.]
   [3. 3.]]

  [[3. 4.]
   [4. 3.]]

  [[4. 4.]
   [6. 5.]]]]
```

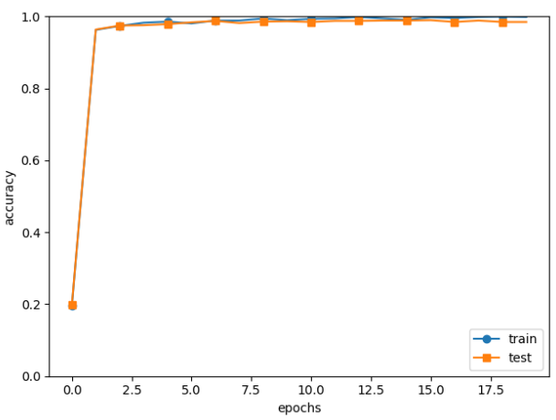
十一、卷积神经网络

11.1 卷积神经网络的实现

程序的输出：

==========Final Test Accuracy ==========

test acc:0.9882



-----到此为止2020 年 2 月 14 日星期五

11.2 神经网络的可视化

11.3 深度神经网络