MLP and BP implemented with Numpy Only

iiGray

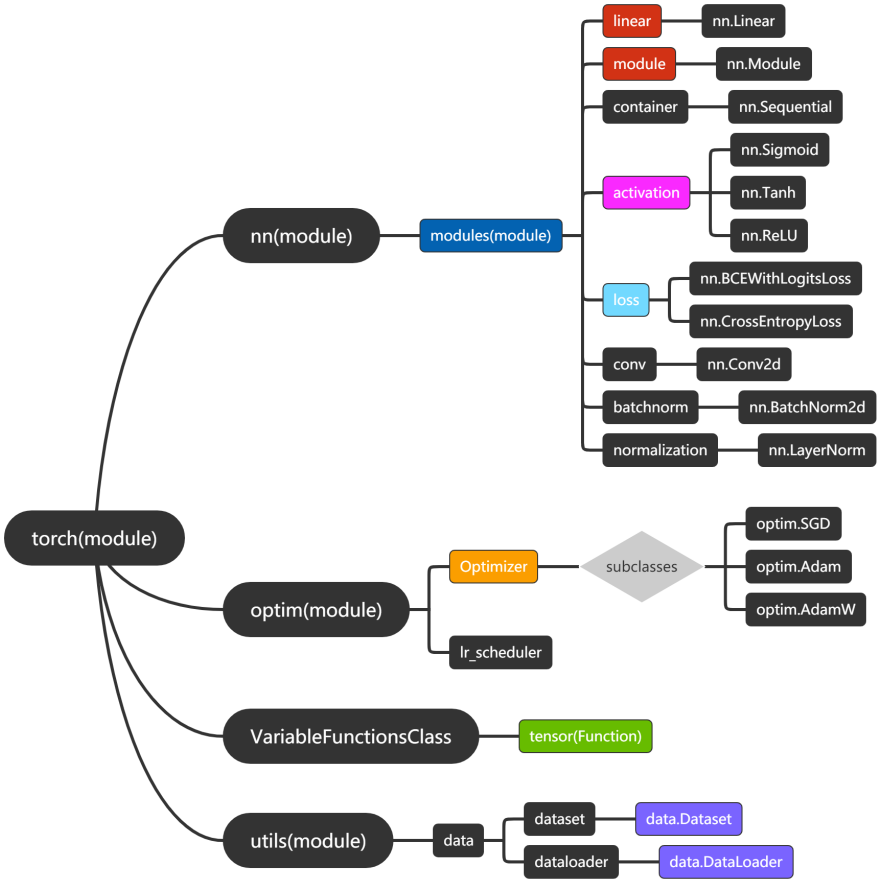
Preface

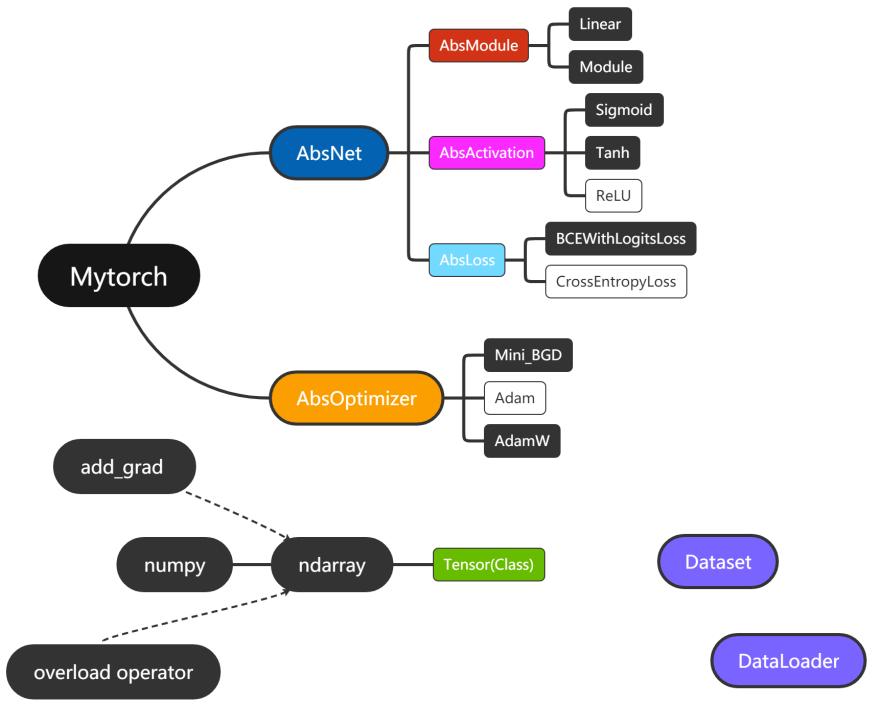
1. 本篇拟仅用 *numpy* 库和 *python* 内置函数和内置库手搓一个小型的神经网络系统, 对 *torch* 最核心的几个模块进行了 *python* 实现, 本篇通过实现最基本的多层感知机 (*Multilayer Perceptron*), 神经网络的最基本训练方式 (*BackPropagation and Gradient Descent*), 希望能够理清 *torch* 的主干框架结构, 以便加深对神经网络的理解。

2. 本篇所有模块全部采用类来实现, 避免多文件带来的阅读不便，*Mytorch* 尽可能地贴合了 *torch* 的体系, 是使用 *numpy* 对 *torch* 体系主干部分的简单实现, 包括:
网络自动求导
前向、反向传播机制
Linear, *Module* 模块,
Sigmoid, *Tanh* 激活函数
BCEWithLogitsLoss 损失函数
Mini_BGD, *AdamW* 梯度下降算法
Dataset, *DataLoader* 数据集和数据迭代器

3. 未实现:
普通 *tensor* 的求导机制 (所以 *backward* 会和 *torch* 体系有一定的差异, 比如 *detach* 和 *is_leaf* 等, 但基本原理相同)
CNN, *RNN* 等网络
ReLU, *GLU* 等激活函数
CrossEntropyLoss, *BCELoss* 等损失函数
Sequential, *ModuleList* 等辅助容器类
SGD, *Adam* 等梯度下降算法 , *StepLR* 等学习率调度器
等等等等.....

[补充]
1.*torch* 是一个含有多个模块的庞大的深度学习框架，其底层主要由 $C/C++$ 实现, 本篇全部采用 *python*，其主要目的在于理清 *torch* 的主干结构，而不是复现 *torch*.
2. 下图是 *torch* 主干框架 (的仅仅一小部分) 和本篇将实现的框架结构，相同颜色表示相互对应，可以看出 *Mytorch* 和 *torch* 的架构并不一样 (只是抽取了主干)，白框部分留给感兴趣的读者实现.
3. 建议从第 4 节开始阅读，同时对比 *torch* 版本的代码查看，并回溯到前面的具体实现.


torch(module)
nn(module)
modules(module)
linear
nn.Linear
module
nn.Module
container
nn.Sequential
activation
nn.Sigmoid
nn.Tanh
nn.ReLU
loss
nn.BCEWithLogitsLoss
nn.CrossEntropyLoss
conv
nn.Conv2d
batchnorm
nn.BatchNorm2d
normalization
nn.LayerNorm
optim(module)
Optimizer
subclasses
optim.SGD
optim.Adam
optim.AdamW
lr_scheduler
VariableFunctionsClass
tensor(Function)
utils(module)
data
dataset
data.Dataset
dataloader
data.DataLoader


Mytorch
AbsNet
AbsModule
Linear
Module
AbsActivation
Sigmoid
Tanh
ReLU
AbsLoss
BCEWithLogitsLoss
CrossEntropyLoss
AbsOptimizer
Mini_BGD
Adam
AdamW
add_grad
numpy
ndarray
Tensor(Class)
overload operator
Dataset
DataLoader

原理

– –

本节主要讲述反向传播原理，给出本篇实现反向传播代码需要依赖的基本公式

[注]　因为**导数要放在形状和原矩阵一样的矩阵中** (这句话很重要)，故矩阵偏微分采用分母布局 (所以和雅可比不同, 雅可比是分子布局), 以下公式均以分母布局给出 (即偏导数矩阵和被求的偏导数的原矩阵形状相同)

元素级别函数求导 (函数作用于矩阵等价于作用于矩阵中每个元素):

$$\frac{\partial f(W)}{\partial W} = \begin{bmatrix} \frac{\partial f(w_{11})}{\partial w_{11}} & \cdots & \frac{\partial f(w_{11})}{\partial w_{1n}} \\ \cdots & \cdots & \cdots \\ \frac{\partial f(w_{n1})}{\partial w_{n1}} & \cdots & \frac{\partial f(w_{nn})}{\partial w_{nn}} \end{bmatrix}$$

标量对向量求导及链式法则:

$$\frac{\partial y}{\partial X} = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \frac{\partial y}{\partial x_2} & \cdots & \frac{\partial y}{\partial x_n} \end{bmatrix}^T, \ where \ y \in R \ , \ X \in R^{n \times 1}$$

$$\frac{\partial z}{\partial X} = \frac{\partial Y}{\partial X} \cdot \frac{\partial z}{\partial Y} = J_X(Y)^T \cdot \frac{\partial z}{\partial Y} ,$$

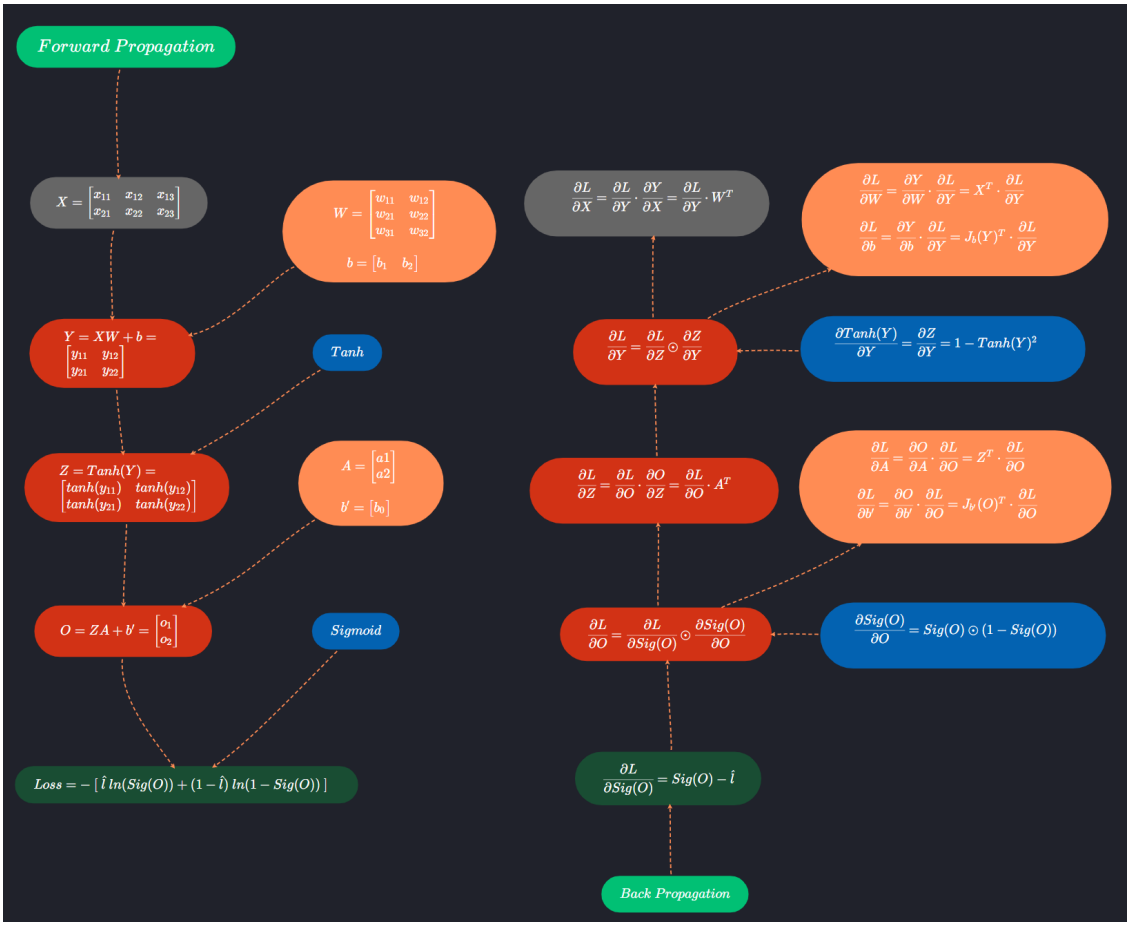
$$where \ X \in R^{n \times 1} \ , \ Y = F(X) \in R^{m \times 1} \ , \ z = G(X) \in R$$

推论：

$$\frac{\partial z}{\partial X} = A^T \cdot \frac{\partial z}{\partial (AX)}, \ where \ z = f(AX) \in R, \ A \in R^{m \times n}, \ X \in R^{n \times p}$$

$$\frac{\partial z}{\partial X} = \frac{\partial z}{\partial (XB)} \cdot B^T, \ where \ z = f(XB) \in R, \ B \in R^{p \times m} \ , \ X \in R^{n \times p}$$

有了上述基础, 就可以来搭建整个网络体系了, 这里给出一个具体例子, 本篇将以与该例相同的模式搭建 (请注意, 以下运算当形状不同时向量会进行广播机制处理):



[1]:
```
''' 以下是全部所需库，pickle 用于载入数据集，关于数据集，请参考 get_data 文件'''
import numpy as np
import math,pickle,time
import matplotlib.pyplot as plt
from collections import defaultdict
from abc import ABC,abstractmethod,abstractproperty
```

1　继承并重写 np.ndarray, 为其加上导数

— — — — — — — — — — — — — — — — — — — —

装饰在前向和反向传播中需要用到的成员方法，使参数在传递过程中始终拥有导数

[2]:
```
''' 对于拷贝传递返回值的成员方法，导数也拷贝生成'''
def add_grad(func):
    def inner(self,*args,**kwargs):
        ret=func(self,*args,**kwargs)
        ret.detach=False
        ret.grad=np.zeros(ret.shape)
        return ret
    return inner
```

```
''' 对于引用传递返回值的成员方法，导数也取引用'''
def add_grad_inplace(func):
    def inner(self,*args,**kwargs):
        ret=func(self,*args,**kwargs)
        if isinstance(ret,np.ndarray):
            ret.__class__=Tensor
            ret.detach=False
            ret.grad=func(self,*args,**kwargs)
        return ret
    return inner

class Tensor(np.ndarray):
    detach=False
    ''' 传入元组会生成元组形状的随机矩阵，其余参数仅进行拷贝封装'''
    def __new__(cls,input_array,requires_grad=True):
        obj=np.asarray(input_array).view(cls) \
        if type(input_array) in (list,Tensor,np.ndarray) \
        else np.random.randn(*input_array).view(cls)
        obj.grad=np.zeros(obj.shape)
        return obj

    ''' 可能有些方法本篇未用到，选择需要的进行装饰即可'''
    @add_grad
    def mean(self,*args,**kwargs):
        return super().mean(*args,**kwargs)
    @add_grad
    def std(self,*args,**kwargs):
        return super().std(*args,**kwargs)
    @add_grad
    def sum(self,*args,**kwargs):
        return super().sum(*args,**kwargs)
    @add_grad
    def __add__(self,*args,**kwargs):
        return super().__add__(*args,**kwargs)
    @add_grad
    def __radd__(self,*args,**kwargs):
        return super().__radd__(*args,**kwargs)
    @add_grad
    def __sub__(self,*args,**kwargs):
        return super().__sub__(*args,**kwargs)
    @add_grad
    def __rsub__(self,*args,**kwargs):
        return super().__rsub__(*args,**kwargs)
    @add_grad
    def __mul__(self,*args,**kwargs):
        return super().__mul__(*args,**kwargs)
    @add_grad
    def __rmul__(self,*args,**kwargs):
        return super().__rmul__(*args,**kwargs)
    @add_grad
    def __pow__(self,*args,**kwargs):
        return super().__pow__(*args,**kwargs)
```

```
    @add_grad
    def __rtruediv__(self,*args,**kwargs):
        return super().__rtruediv__(*args,**kwargs)
    @add_grad
    def __truediv__(self,*args,**kwargs):
        return super().__truediv__(*args,**kwargs)
    @add_grad
    def __matmul__(self,*args,**kwargs):
        return super().__matmul__(*args,**kwargs)
    @add_grad
    def __rmatmul__(self,*args,**kwargs):
        return super().__rmatmul__(*args,**kwargs)

    @add_grad_inplace
    def reshape(self,*args,**kwargs):
        return super().reshape(*args,**kwargs)
    @add_grad_inplace
    def __getitem__(self,*args,**kwargs):
        return super().__getitem__(*args,**kwargs)

    @property
    def zero_grad_(self):
        self.grad=np.zeros(self.grad.shape)
    @property
    def grad_fn(self):
        return "Leaf" if self.detach else "Node"

    def detach_(self,whether=True):
        self.detach=whether

''' 定义网络所需的函数'''
def exp(x):
    if hasattr(x,"__len__"):
        return Tensor([exp(i) for i in x])
    return math.exp(x)
def log(x):
    if hasattr(x,"__len__"):
        return Tensor([log(i) for i in x])
    return math.log(x)
```

2 定义体系所需基本抽象模块

– –

定义一个 $MyTorch$ 抽象类，充当该体系一切类的基类

$AbsNet$ 表示网络主体模块，一切在前向传播中需要经历的模块都继承自该模块

$AbsOptimizer$ 表示求解算法模块，一切基于梯度下降的求解算法都继承自该模块

AbsActivation 是激活函数模块，激活函数无需要更新的参数，因此需要和网络参数区分开

AbsModule 是网络中含有需要更新参数的模块，一切含参模块继承自该模块

AbsLoss 是损失函数模块，一切损失函数继承自改模块

各个模块的继承关系如 *P*2

[3]:
```
class MyTorch(ABC):pass
class AbsNet(MyTorch):
    @abstractmethod
    def __init__(self,*args,**kwargs):pass
    @abstractmethod
    def __call__(self,*args,**kwargs):pass
    @abstractmethod
    def forward(self,*args,**kwargs):pass
    @abstractmethod
    def backward(self,*args,**kwargs):pass

class AbsActivation(AbsNet):
    def __init__(self,*args,**kwargs):pass
    @abstractmethod
    def function(self,*args,**kwargs):pass
    def __call__(self,x):
        self.input=x
        self.output=self.forward(x)
        return self.output
    @property
    def zero_grad_(self):
        if "input" in self.__dict__.keys():
            self.input.zero_grad_

class AbsOptimizer(MyTorch):
    @abstractmethod
    def __init__(self,*args,**kwargs):pass
    @abstractmethod
    def step(self,*args,**kwargs):pass
    def zero_grad(self):
        self.parameters.zero_grad_

class AbsModule(AbsNet):
    @abstractproperty
    def zero_grad_(self):pass
    @abstractproperty
    def __repr__(self):pass

class AbsLoss(AbsNet):
    @abstractproperty
    def outgrad(self):pass
    def backward(self):
        cgrad=self.outgrad
        for block_name,block in reversed(self.net.__dict__.items()):
```

```
            if type(block).__base__ not in (AbsActivation,AbsModule,Module,):
                continue
            cgrad=block.backward(cgrad)
```

3 定义内置模块

— — — — — — — — — — — — — — — — — —

不同类型的模块需继承各自相应的抽象模块

定义内置模块需要继承 *AbsModule*，并实现 *__repr__* 方法方便打印

定义内置激活函数需要继承 *AbsActivation*，并需要重载 *__init__*，*forward* 和 *backward*

定义内置损失函数需继承 *AbsLoss*，不用重载 *backward*，但要实现 *outgrad* 方法

定义内置梯度下降算法需继承 *AbsOptimizer*，并实现 *step* 方法

自定义网络只需继承 *Module*，并只需实现 *__init__* 和 *forward* 即可

[4]:
```
''' 定义线性层'''
class Linear(AbsModule):
    def __init__(self,in_channels,out_channels,bias=True):
        self.in_channels=in_channels
        self.out_channels=out_channels
        self.bias=bias
        ''' 使用和 torch.nn.Linear 中一样参数初始化:参数 a=sqrt(5),mode='fan_in' 的 kaiming_uniform_ 初始化'''
        bound=1/math.sqrt(in_channels)

        self.parameters={"weights":Tensor((np.random.rand(in_channels,out_channels)-0.5)*2*bound)}
        if bias:
            self.parameters["bias"]=Tensor((np.random.rand(1,out_channels)-0.5)*2*bound)

    def __call__(self,x):
        self.input=x
        self.output=self.forward(x)
        return self.output

    def forward(self,x):
        out=x @ self.parameters["weights"]
        if self.bias:
            out+=self.parameters["bias"]
        return out

    def backward(self,cgrad):
        try:
            self.input.grad= cgrad @ self.parameters["weights"].T
        except AttributeError:
            raise AttributeError("The layer: "+self.__repr__()+" absent from FP!")
```

```
        self.parameters["weights"].grad+= self.input.T @ cgrad
        if self.bias:
            self.parameters["bias"].grad+= cgrad.sum(0,keepdims=True)
        return self.input.grad.copy()

    def __repr__(self):
        return f"Linear(in_features={self.in_channels}, "+\
        f"out_features={self.out_channels}, bias={self.bias})"

    @property
    def zero_grad_(self):
        if "input" in self.__dict__.keys():
            self.input.zero_grad_
        self.parameters["weights"].zero_grad_
        if self.bias:
            self.parameters["bias"].zero_grad_

''' 定义激活函数层，类似于 nn.Sigmoid 和 nn.Tanh'''
class Sigmoid(AbsActivation):
    def function(self,x):
        return 1/(1+exp(-x))

    def forward(self,x):
        return self.function(x)

    def backward(self,cgrad):
        assert self.output.shape==cgrad.shape,"Activation Sigmoid BP Error!"
        try:
            self.input.grad=(self.output*(1-self.output))*cgrad
        except (AttributeError):
            raise AttributeError("Layer: " +self.__repr__()+" absent from FP!")
        return self.input.grad

    def __repr__(self):
        return "Sigmoid()"

class Tanh(AbsActivation):
    def function(self,x):
        return (1-exp(-2*x))/(1+exp(-2*x))

    def forward(self,x):
        return self.function(x)

    def backward(self,cgrad):
        assert self.output.shape==cgrad.shape,"Activation Tanh BP Error!"
        try:
            self.input.grad=(1-self.output**2)*cgrad
        except (AttributeError):
            raise AttributeError("Layer: " +self.__repr__()+" absent from FP!")
        return self.input.grad
    def __repr__(self):
        return "Tanh()"
```

```
''' 定义模块层，该类类似于 torch.nn.Module'''
class Module(AbsModule):
    def __init__(self,*args,**kwargs):
        raise NotImplementedError("Class: \"Module\" has to be overrided!")

    def __call__(self,*args,**kwargs):
        return self.forward(*args,**kwargs)

    def forward(self,*args,**kwargs):
        raise NotImplementedError("Function: \"forward()\" has to be overloaded!")

    def backward(self,cgrad):
        for block_name,block in reversed(self.__dict__.items()):
            if type(block).__base__ not in (AbsActivation,AbsModule,Module):continue
            cgrad=block.backward(cgrad)
        return cgrad

    def __repr__(self):
                name="Net(\n"
                for block_name,block in self.__dict__.items():
                        if type(block).__base__ not in (AbsNet,AbsActivation,AbsModule,):
                                continue
                        name+="  ("+str(block_name)+"): "+block.__repr__()+"\n"
                return name+")"

    @property
    def zero_grad_(self):
        for block_name,block in self.__dict__.items():
            if type(block).__base__ not in (AbsActivation,AbsModule,Module):continue
            block.zero_grad_


''' 定义损失函数，这里使用二元交叉熵 + Sigmoid 损失函数'''
class BCEWithLogitsLoss(AbsLoss):
    def __init__(self,net,reduction="none"):
        self.net=net
        self.reduction=reduction
        self.function=Sigmoid()

    def __call__(self,y,y_hat):
        return self.forward(y,y_hat)

    def forward(self,y,y_hat):
        self.out=y
        self.hat=y_hat
        p=self.function(y)
        ret=-(y_hat*log(p)+(1-y_hat)*log(1-p))
        if self.reduction=="mean":return ret.mean()
        elif self.reduction=="sum":return ret.sum()
        return ret
```

```
    @property
    def outgrad(self):
        out=self.out
        hat=self.hat
        out.grad=(self.function(out)-hat)/out.shape[0]
        return out.grad


''' 定义梯度下降算法，这里使用最普通的小批量梯度下降算法，其实和 SGD 区别只在于遍历数据集的方式'''
class Mini_BGD(AbsOptimizer):
    def __init__(self,net,lr=0.001):
        self.parameters=net
        self.lr=lr
    def step(self):
        for block_name,block in reversed(self.parameters.__dict__.items()):
            if type(block).__base__!=AbsModule:continue
            for name,weight in block.parameters.items():
                weight-=self.lr*weight.grad


''' 定义另一个非常厉害的优化器:AdamW 优化算法，读者可查阅 AdamW 公式对照查看'''
class AdamW(AbsOptimizer):
    def __init__(self,net,lr=0.01,betas=(0.9,0.999),eps=1e-08,weight_decay=0.01):
        self.parameters=net
        self.lr=lr
        self.betas=betas
        self.eps=eps
        self.weight_decay=weight_decay

        ''' 初始化 t,mt,vt'''
        self.t=0
        self.mt=defaultdict(dict)
        self.vt=defaultdict(dict)
        for block_name,block in reversed(self.parameters.__dict__.items()):
            if type(block).__base__!=AbsModule:continue
            for name,weight in block.parameters.items():
                self.mt[block_name][name]=np.zeros_like(weight)
                self.vt[block_name][name]=np.zeros_like(weight)
    def step(self):
        beta1,beta2=self.betas
        self.t+=1
        for block_name,block in self.parameters.__dict__.items():
            if type(block).__base__!=AbsModule:continue
            for name,weight in block.parameters.items():
                gt=weight.grad
                mt=self.mt[block_name][name]
                vt=self.vt[block_name][name]

                weight-=self.lr*gt

                self.mt[block_name][name]=beta1*mt+(1-beta1)*gt
```

```
                self.vt[block_name][name]=beta2*vt+(1-beta2)*(gt*gt)

                mt=mt/(1-np.power(beta1,self.t))
                vt=vt/(1-np.power(beta2,self.t))

                weight-=self.lr*mt/(np.sqrt(vt)+self.eps)


''' 定义批量数据迭代器，使数据集可以按照小批量传入网络'''
class DataLoader:
    def __init__(self,dataset,batch_size):
        self.dataset=dataset
        self.batch_size=batch_size
        self.num=0
        self.stop=False
        self.final=False

    def __iter__(self):
        return self
    ''' 变量 self.final 使其可以反复迭代'''
    def __next__(self):
        if self.final==True:
            self.num=0
            self.final=False

        if not self.stop:
            bs=self.batch_size
            num=self.num

            self.num=min(self.num+bs,len(self.dataset))
            if self.num==len(self.dataset):self.stop=True
            return [Tensor(np.stack([self.dataset[i][j]
                                     for i in range(num,self.num)]))
                    for j in range(2)]

        self.stop=False
        self.final=True
        raise StopIteration
```

4 定义数据集和本问题的网络

— — — — — — — — — — — — — — — — — —

以下部分可以对比用 *torch* 训练的那一份文件, 几乎是一样的

[5]:
```
''' 定义数据集'''
class Dataset:
    def __init__(self,data):
        self.data=data if type(data)==list else pickle.load(open(data,"rb"))
```

```
    def __getitem__(self,i):
        return \
    np.array(list(self.data[i][0])),np.array([self.data[i][1]],dtype=np.int32)

    def __len__(self):
        return len(self.data)

''' 采用多层感知机，并使用 tanh 作为激活函数'''
class Net(Module):
    def __init__(self,in_dim):
        self.linear1=Linear(in_dim,5)
        self.tanh1=Tanh()

        self.linear2=Linear(5,3)
        self.tanh2=Tanh()

        self.linear3=Linear(3,1)

    def forward(self,x):
        out=self.linear1(x)
        out=self.tanh1(out)

        out=self.linear2(out)
        out=self.tanh2(out)

        out=self.linear3(out)

        return out
```

5 定义绘图函数, 训练函数, 预测函数

— — — — — — — — — — — — — — — — — —

draw 函数中参数 *data_path* 是 *.pkl* 文件路径, 详见 *get_data* 文件， *train* 和 *predict* 函数可对照 *torch* 版

[6]:
```
def draw(data_pth):
    dots=pickle.load(open(data_pth,"rb"))

    dots0=[[dot[0][0],dot[0][1]] for dot in dots if dot[-1]==0]
    dots0x=[k[0] for k in dots0]
    dots0y=[k[1] for k in dots0]

    dots1=[[dot[0][0],dot[0][1]] for dot in dots if dot[-1]==1]
    dots1x=[k[0] for k in dots1]
    dots1y=[k[1] for k in dots1]

    plt.scatter(dots0x,dots0y,c="g")
```

```
    plt.scatter(dots1x,dots1y,c="b")


def train(net,dataloader,epochs,lr,eps=1e-5):
    ''' 选择优化器 (两个都可以)'''
#     optimizer=Mini_BGD(net,lr=lr)
    optimizer=AdamW(net,lr=lr)
    ''' 选择损失函数'''
    l=BCEWithLogitsLoss(net,reduction="mean")

    loss_lst=[100]
    ''' 开始遍历 epochs 次数据集'''
    for _ in range(epochs):
        ''' 对每次遍历取出小批量 (这是必要的, 尤其在数据量很大的时候)'''
        for x,y in dataloader:
            ''' 前向传播'''
            y_pre=net(x)
            ''' 计算损失函数'''
            loss=l(y_pre,y)
            ''' 记录误差'''
            loss_lst+=[loss]
            ''' 反向传播求导数'''
            l.backward()
            ''' 更新参数'''
            optimizer.step()
            ''' 导数清零'''
            optimizer.zero_grad()

        ''' 误差足够低时退出'''
        if abs(loss_lst[-1]-loss_lst[-2])<eps:
            break

    ''' 输出训练信息'''
    print("Update times:",len(loss_lst))
    print("Final_Loss:",loss_lst[-1])

    plt.xlabel("update_times")
    plt.ylabel("loss")
    plt.plot(loss_lst)
    plt.show()

def predict(net,test_dataset,trn_path=False,eps=0.001):
    ''' 初始化真阳, 真阴, 假阳, 假阴'''
    TP,TN,FP,FN=0,0,0,0
    ALL=len(test_dataset)
    datas=[[[],[]],[[],[]]]

    for i in range(ALL):
        dot,l=test_dataset[i]
        dot=dot[None,:]
        l=l[None,:]
        out=net(dot)[0][0]
```

```
        pre=0 if out <0 else 1

        datas[pre][0]+=[dot[0][0]]
        datas[pre][1]+=[dot[0][1]]

        if l==1:
            if out<0:FP+=1
            else:TP+=1
        else:
            if out<0:TN+=1
            else: FN+=1
    if trn_path:
        draw(trn_path)

    ''' 绘出预测情况，黄色为预测负类，红色为预测正类'''
    plt.scatter(datas[0][0],datas[0][1],c="y")
    plt.scatter(datas[1][0],datas[1][1],c="r")
    plt.show()
    ''' 精确率'''
    print("Precision:\t",(TP+eps)/(TP+FP+eps))
    ''' 召回率'''
    print("Recall:\t",(TP+eps)/(TP+FN+eps))
    ''' 准确率'''
    print("Accuracy:\t",(TP+TN+eps)/(TP+TN+FP+FN+eps))
```

6 训练

[7]:
```
''' 载入训练集和测试集'''
trn_dataset=Dataset("trn_datas.pkl")
tst_dataset=Dataset("tst_datas.pkl")

''' 设置随机数种子以便复现'''
np.random.seed(0)
```

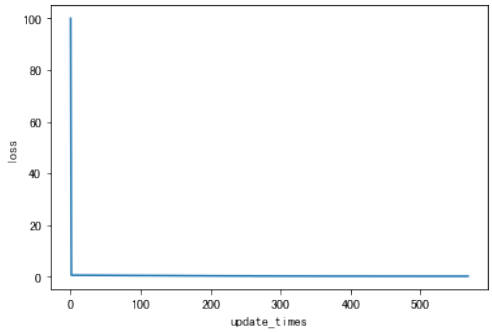
[8]:
```
net=Net(in_dim=2)
dataloader=DataLoader(trn_dataset,batch_size=50)
```

[9]:
```
''' 查看模型结构'''
net
```

[9]:
```
Net(
  (linear1): Linear(in_features=2, out_features=5, bias=True)
  (tanh1): Tanh()
  (linear2): Linear(in_features=5, out_features=3, bias=True)
  (tanh2): Tanh()
  (linear3): Linear(in_features=3, out_features=1, bias=True)
)
```

[10]:
```
train(net,dataloader,epochs=200,lr=0.001)
```

```
Update times: 569
Final_Loss: 0.28847184152795263
```



7 验证

— — — — — — — — — — — — — — — —

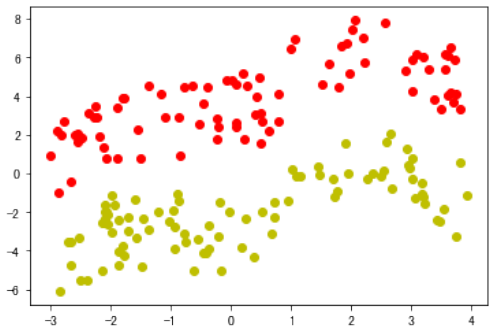
由于本任务较简单，采用训练集进行验证

(仅因以此来绘图，方便展示，实际上验证集不应与训练集和测试集相交)

— — — — — — — — — — — — — — — —

如下，红色为预测正类，黄色为预测负类:

[11]:
```
predict(net,trn_dataset)
```



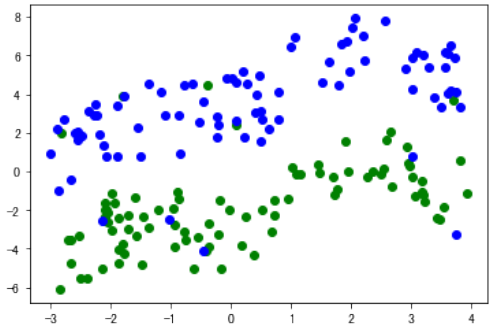

```
Precision:      0.9404768990845347
Recall:  0.9404768990845347
Accuracy:       0.9428574693858892
```

8 测试

– – – – – – – – – – – – – – – – – –

绘出原始训练数据集，蓝色为样本正类，绿色为样本负类

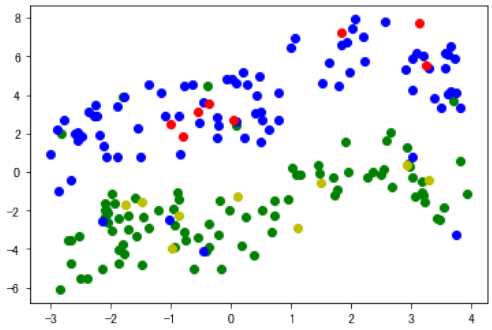
[12]:
```
draw("trn_datas.pkl")
```



– – – – – – – – – – – – – – – – – –

在测试集上进行预测，并与原始训练集绘在同一张图上

[13]:
```
predict(net,tst_dataset,trn_path="trn_datas.pkl")
```



```
Precision:      1.0
Recall:  1.0
Accuracy:       1.0
```

可见其全部预测正确

EndNote

该数据集是代码随机生成的，具体过程见 *generate_data* 文件，为了更专注于探索底层的逻辑，这份数据的训练很简单，基本可以预测全对，但要想让训练集准确率和召回率同时达到 93% 以上, 会稍有难度. 同时，本篇未考虑过拟合等问题.