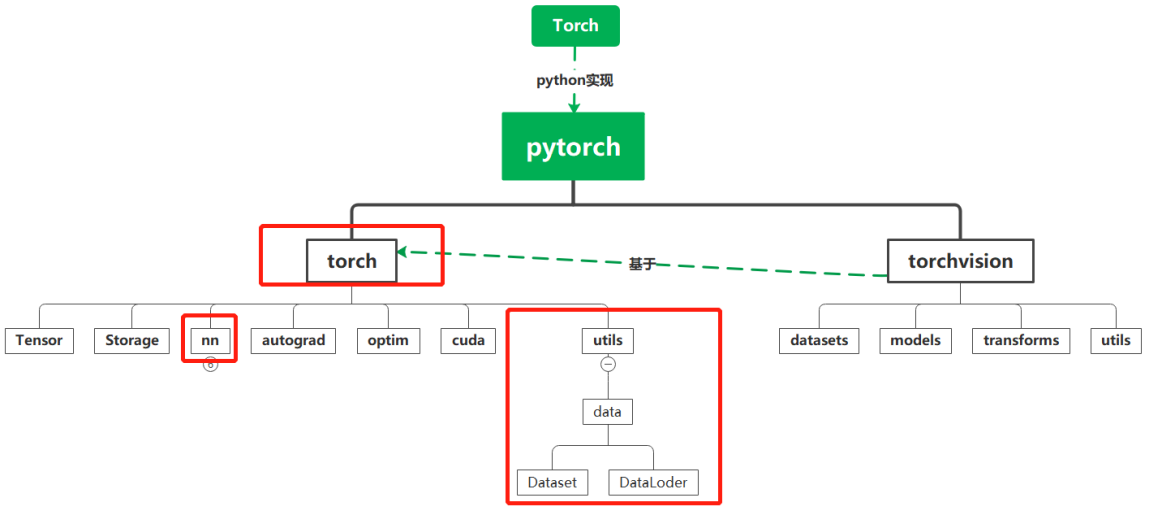
PJ1 Part1实验报告

代码架构

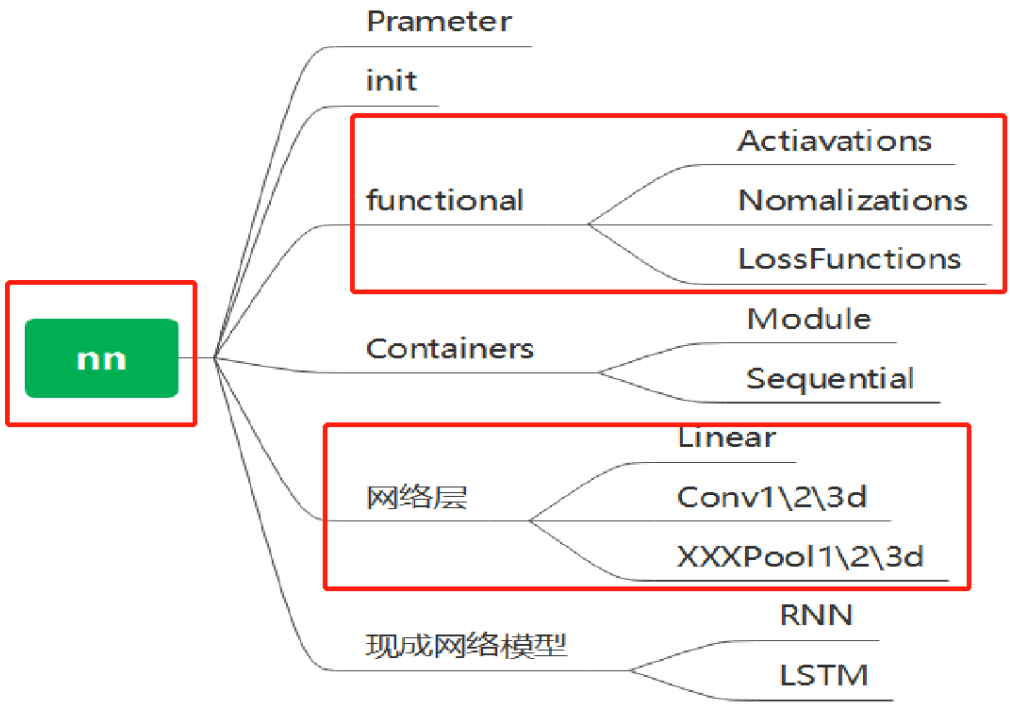
宏观架构说明：

由于实验要求设计可伸缩易调整的网络结构，可以灵活设置层数、神经元个数、学习率等，所以我采用面向对象的方法写了一些类，架构方面参考pytorch的架构，如下图：



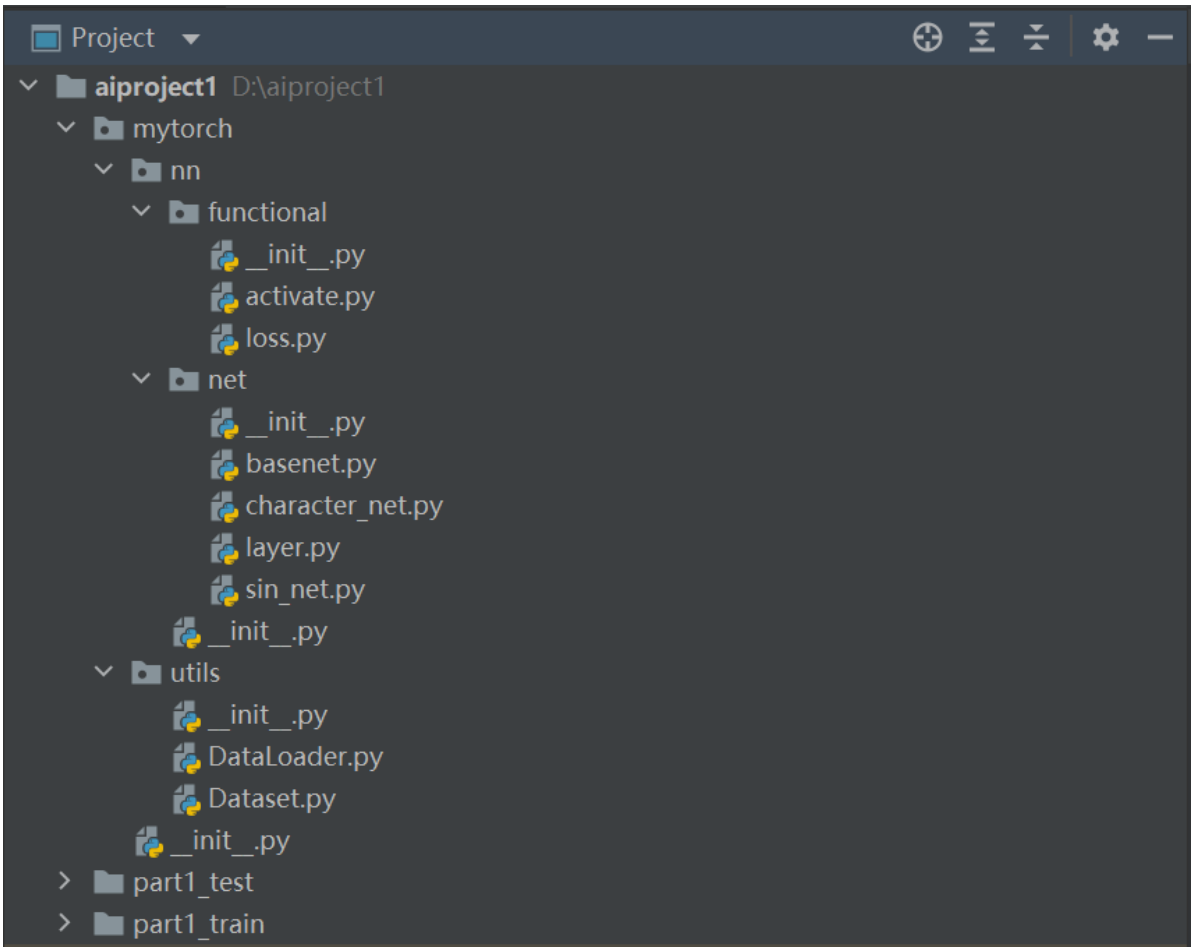
这里我只实现了torch的nn部分和utils部分，即图中红线框出来的部分。

对于nn部分，布局如下，我实现的部分也用红线框出来：



架构的参考文章为：(184条消息) PyTorch结构、架构分析*pytorch架构*windrisess的博客-CSDN博客

我的代码的布局情况如下：



接下来对主要代码进行讲解：

- mytorch.nn.functional.activate

activate.py中的类将作为bp网络每个layer的一个对象，用于为前向计算和反向传播提供方便。

activate.py文件中写了激活函数的类，一共写了**Activate**、**Tanx**、**Sigmoid**、**Softmax**四个类，以**Sigmoid**为例进行说明，里面定义了一个activate函数用于对输入值计算Sigmoid激活值，一个derivate函数用于Sigmoid函数进行求导，返回求导后的雅可比矩阵，用于反向传播时计算梯度。同理 **Softmax** 类同理，只是它的activate和derivate都是针对Softmax函数进行的。Sigmoid类的代码如下：

```
class Sigmoid:
    def activate(self, x) -> float:
        return 1 / (1 + np.exp(-x))

    def derivate(self, z, a) -> float:
        # f(x)' = f(x) * (1 - f(x))
        size = a.size
        Jacobi = np.zeros((size,size))
        for i in range(0,size):
            Jacobi[i][i] = a[i] * (1 - a[i])
        return Jacobi
```

需要强调一下的是**Activate**类中的activate函数不做任何处理直接返回输入值，derivate函数也是并不求导直接返回单位矩阵，这样设计的原因在于：对于拟合sin函数的任务，最后一层并不需要激活函数，但是如果不给最后一层赋值一个激活函数的对象的话，反向传播的时候就会因为最后一层没有激活函数的求导而和其他层不一致，为了保持计算的一致性，设置**Activate**类专门用于sin函数拟合任务的网络的最

后一层。**Activate**的代码如下：

```
class Activate:
    def __init__(self):
        pass
    def activate(self,X):
        return X
    def derivate(self, z, a):
        """
        返回该激活函数的Jacobi矩阵
        :param z: 输入变量
        :param a: 输入遍量经过原激活函数运算后的结果，用来减少计算量
        :return: 返回单位矩阵
        """
        size = z.size
        Jacobi = np.zeros((size, size))
        for i in range(size):
            Jacobi[i, i] = 1
        return Jacobi
```

- mytorch.nn.functional.loss

loss.py里面定义了损失函数的类，包括：**MSELoss**、**CELoss**

MSELoss即均方误差（Mean Squared Error），公式如下：

$$MSE = \frac{\sum_i^n (y_i - y_i^p)^2}{n}$$

我们的MESLoss中定义函数loss_calculate计算误差，定义函数backprop用于反向传播，逐层修改每个网络层的参数。MSELoss的loss_calculate定义如下：

```
    def loss_calculate(self,net,y):
        self.label = y
        self.net = net
        last_layer = self.net.last_layer
        difference = last_layer.a - self.label
        # loss = 1/2sum(y_i - y)^2 for all y_i
        loss = np.dot(difference.T,difference).sum() /2
        return loss
```

MSELoss的backprop代码如下(这一部分由于我的求导是算的Jacobi矩阵，所以计算梯度的过程中存在矩阵数据对齐的问题，花了好长时间来调)：

```
    def backprop(self):
        """
        反向传播函数，根据loss和求导，逐层修改参数w和b
        参考助教给的知乎上讲解反向传播的文章
        梯度计算使用链式法则
        先修改最后一层的权重，再逐层向前修改
        :return: None
        """
        last_layer = self.net.last_layer
        error = last_layer.a - self.label    # 预测值和真实值之间的差值
```

```
        delta =
np.dot(last_layer.function.derivate(last_layer.z,last_layer.a).T, error)
        # delta是对w和b求导的公共部分，可以共用

        dw = np.dot(last_layer.X,delta.T) * self.net.lr

        db = delta * self.net.lr
        # 接下来更新权重w和偏移bias
        last_layer.backward(dw.T, db)
        last_layer = last_layer.pre
        while last_layer:
            delta_z = np.dot(last_layer.next.weight,
last_layer.function.derivate(last_layer.z,last_layer.a))
            # delta_z = np.dot(last_layer.function.derivate(last_layer.z,
last_layer.a),last_layer.next.weight.T).T
            delta = np.dot(delta_z.T,delta)
            # 此时delta是这一层对w和b求导的公共部分，可共用
            dw = np.dot(last_layer.X,delta.T) * self.net.lr
            db = delta * self.net.lr
            last_layer.backward(dw.T,db)
            last_layer = last_layer.pre
```

对于MSE函数：

(1) 输出层的梯度计算公式

$$\begin{aligned}\frac{\partial C(W,b)}{\partial w^{(l)}} &= \frac{\partial C(W,b)}{\partial a^{(l)}}\frac{\partial a^{(l)}}{\partial z^{(l)}}\frac{\partial z^{(l)}}{\partial w^{(l)}} \\ &= (a^{(l)}-y)\odot\sigma'(z^{(l)})a^{(l-1)}\end{aligned}$$

$$\begin{aligned}\frac{\partial C(W,b)}{\partial b^{(l)}} &= \frac{\partial C(W,b)}{\partial a^{(l)}}\frac{\partial a^{(l)}}{\partial z^{(l)}}\frac{\partial z^{(l)}}{\partial b^{(l)}} \\ &= (a^{(l)}-y)\odot\sigma'(z^{(l)})\end{aligned}$$

(2) 隐藏层的梯度计算公式

$$\delta^{(l)} = \frac{\partial C(W,b)}{\partial z^{(l+1)}}\frac{\partial z^{(l+1)}}{\partial z^{(l)}} = \delta^{(l+1)}\frac{\partial z^{(l+1)}}{\partial z^{(l)}}$$

而$z^{(l+1)}$和$z^{(l)}$的关系如下：

$$z^{(l+1)} = W^{(l+1)}a^{(l)} + b^{(l+1)} = W^{(l+1)}\sigma(z^{(l)}) + b^{(l+1)}$$

$$\frac{\partial z^{(l+1)}}{\partial z^{(l)}} = (W^{(l+1)})^T \odot \sigma'(z^{(l)})$$

$$\delta^{(l)} = (W^{(l+1)})^T \delta^{(l+1)} \odot \sigma'(z^{(l)})$$

现在我们得到了 $\delta^{(l)}$ 的递推关系式，只要求出了某一层的$\delta^{(l)}$，求解$w^{(l)}$，$b^{(l)}$的对应梯度就很简单：

$$\frac{\partial C(W,b)}{\partial w^{(l)}} = \delta^{(l)} a^{(l-1)}$$

$$\frac{\partial C(W,b)}{\partial b^{(l)}} = \delta^{(l)}$$

Softmax类同理，代码不再展示。

在完成反向传播这一块的时候，我参考了文章：https://zhuanlan.zhihu.com/p/71892752

上面函数backprop中的对梯度的计算也是按照这篇文章中的讲解来写的，反向传播就是我们有一个用来计算预测值和真值之间误差大小的损失函数，这里在拟合sin函数里用的MSE损失函数，在汉字识别任务里用的CE损失函数，我们训练的目的是让预测值和真值的差距变小，即让损失函数的值变小，我们可以修改的是网络中一些参数的值，比如weight和bias，自然可以想到用**损失函数对weight和bias求导**来找到使得损失函数值变小的weight和bias修改的方向，然后乘上我们的学习率得到对参数修改的量从而更新参数。

我们通过正向计算得到一个预测值，进而算出损失函数，然后损失函数对参数进行求导得到梯度，再根据梯度和学习率对参数进行更新，重复迭代这个过程，我们就能得到一个损失函数较小的一组参数，这也是反向传播的目的。

- mytorch.nn.net.layer

layer.py文件中定义了bp网络中“层”的类Linear，类Linear包含了一层中的输入参数个数、输出参数个数、输入数据、加权计算后的值、激活后的值、激活函数等参数。并采用setup函数定义对输入计算输出的权重weight和偏移bias（用于sin拟合任务）。w_b_setup用于汉字识别任务的权重初始化。同时定义了forward函数用于前向计算，backward函数用于根据反向传播算出来的dW和db来更新自己的权重weight和bias。代码展示如下，该类的参数方便修改，使得网络结构易伸缩调整：

```
class Linear:
    """
    全连接层的类，用于后续连接成网络net
    参数包括：
    1、输入节点数量,可方便调整
    2、输出节点数量,可方便调整
    3、输入到输出的权重矩阵weight
    4、偏移量bias
    """
    def __init__(self,input_num,output_num,f):
        self.input_num = input_num
        self.output_num = output_num
        self.X = None            # 上一层的输出，这一层的输入
        self.z = None            # 存储激活前的值
        self.a = None            # 存储激活后的值
        self.function = f
        self.pre = None         # 用于和其他层进行连接
        self.next = None        # 用于和其他层进行连接
        self.islast = False
        self.setup()
    def setup(self):
```

```
        self.weight = np.random.uniform(-1, 1, (self.output_num,
self.input_num))
        self.bias = np.random.uniform(-1, 1, (self.output_num, 1))
    def w_b_setup(self):
        """
        使用nguyen-widrow算法初始化
        """
        self.overlap = 0.7
        if self.islast:
            self.w = np.random.uniform(-1, 1, (self.n_in, self.n_out))
            self.b = np.random.uniform(-1, 1, (self.n_out, 1))
            return
        self.w = np.random.uniform(-1, 1, (self.n_in, self.n_out))
        self.s = np.abs(self.w).sum(axis=0)
        self.w = self.overlap * np.power(self.n_out, 1 / self.n_in) * self.w

        self.b = np.random.uniform(-1, 1, (self.n_out, 1))
        self.g = self.overlap * np.power(self.n_in, (1 / (self.n_out - 1)))
        self.k = np.array(range(0, self.n_out)).reshape(self.n_out, 1)
        self.b = ((2 * self.k) / self.n_out - 1) * self.g * self.b
    def forward(self,x):
        self.x = x
        self.z = np.matmul(self.weight,self.x) + self.bias
        self.a = self.function.activate(self.z)
        return self.a
    def backward(self,dw,db):
        self.weight -= dw
        self.bias   -= db
```

- mytorch.nn.net.basenet

basenet是一个网络基类，里面只定义了一个connect函数用于连接网络中的每一层。网络的setup函数和forward函数在子类中要重写。

- mytorch.nn.net.sin_net

sin_net.py中定义了类sin_net是针对sin函数拟合任务的类，继承自basenet，里面具体定义了有几个层，设置网络的每个层的setup函数（重写），函数forward调用每一层对象的forward函数用于前向传播（重写），学习率可以通过类中的setup函数方便地进行修改，代码如下：

```
class sin_net(basenet):
    def __init__(self):
        self.lr = 0.005
        self.Layer1 = None
        self.Layer2 = None
        self.Layer3 = None
    def setup(self,lr,layer1,layer2,layer3):
        self.lr = lr
        self.Layer1 = layer1
        self.Layer2 = layer2
        self.Layer3 = layer3
        self.connect(self.Layer1,self.Layer2,self.Layer3)
    def forward(self, x):
        x = self.Layer1.forward(x)
        x = self.Layer2.forward(x)
        x = self.Layer3.forward(x)
```

```
        return X
```

- mytorch.nn.net.character_net

同样继承自basenet，针对汉字识别任务的类，类似sin_net，具体细节不同。

- mytorch.utils.DataLoder

用于读练数据集并划分训练集和验证集，每次划分的时候要对数据进行打乱，防止每次的训练集和验证集都是一样的。在导入数据时对数据进行预处理，使用了**高斯模糊**的手段，不仅提高了验证集上的准确率，也加快了收敛速度。

训练和验证部分

在Part1_train文件夹下有文件sin_train.py用于训练拟合sin函数的bp网络，sin_test.py用于检测训练好的网络在验证机上的准确率，同理有character_train.py用于训练汉字识别的网络，character_test.py用于检测网络的识别准确率。以sin_train.py为例，代码如下：

```
from mytorch.nn.functional import Tanx,Sigmoid,Softmax,Activate
from mytorch.nn.net import Linear,sin_net
import numpy as np
from mytorch.nn.functional import MSELoss
import pickle
from tqdm import tqdm,trange

if __name__ == "__main__":
    sinNet = sin_net()                  #生成sin_net的对象
    layer1 = Linear(1,7, Sigmoid())     #生成第一层的对象
    layer2 = Linear(7,5,Sigmoid())      #生成第二层的对象
    layer3 = Linear(5,1,activate())     #生成第三层的对象

    sinNet.setup(0.001,layer1,layer2,layer3)  #利用setup函数搭建起来一个三层的网络

    n_feature = 1
    n_count = 4000
    epoch = 500

    x = (2 * np.random.rand(n_count, n_feature) - 1) * np.pi #生成一个（4000，1）的
数组，元素大小在正负π之间
    x_list = []
    for i in x:
        x_list.append(np.reshape(i, (n_feature, 1)))
    x_list = np.array(x_list)
    y_list = np.sin(x_list)  # label
    loss = MSELoss()
    # print(x_list.shape)
    # exit()

    for i in tqdm(range(epoch)):
        ls = 0
        for j in range(n_count):
            X = x_list[j]
            y = y_list[j]
```

```
            y_pred = sinNet.forward(X)
            ls += loss.loss_calculate(sinNet,y)

            loss.backprop()
        average_loss = ls / n_count
        print("general loss = {}, average loss = {}".format(ls,average_loss))

    f = open('models/sinNet', 'wb')
    pickle.dump(sinNet, f)
    f.close()
    print("模型存储完毕")
```

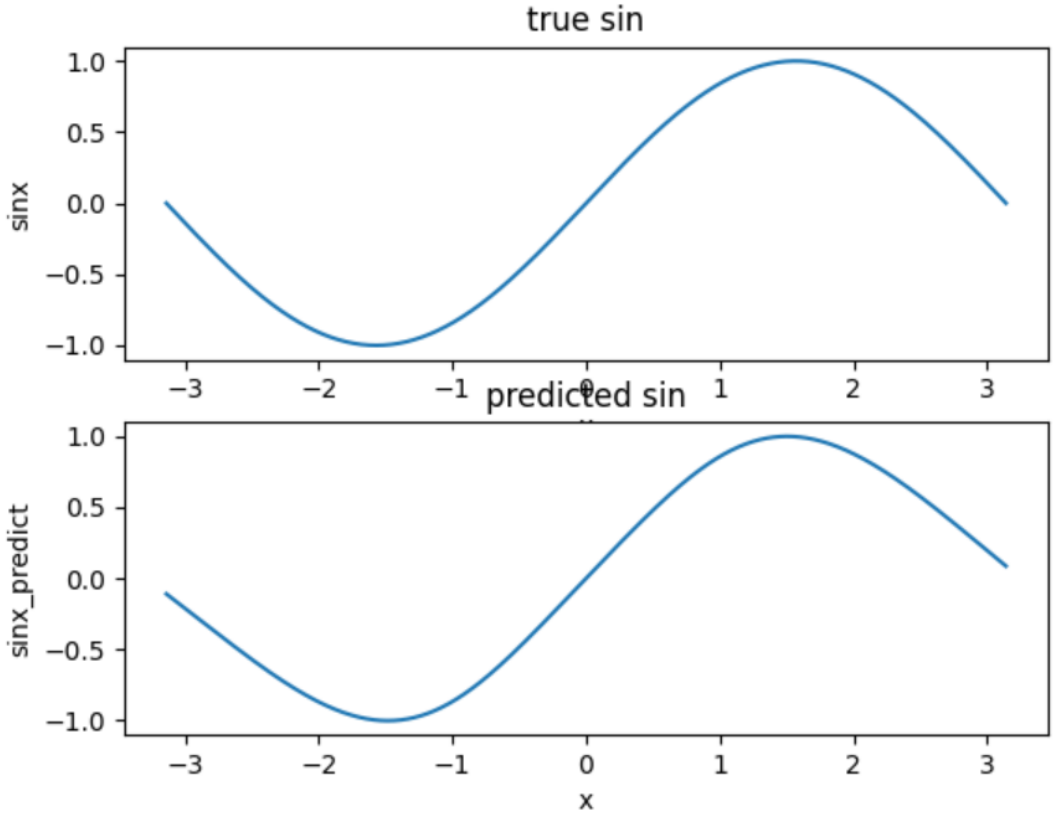
训练结果

拟合sin函数

在sin_train.py中训练，500个epoch，每个epoch4000个数据，训练后将平均误差大致在0.001左右，满足平均误差小于0.01的要求，如下：

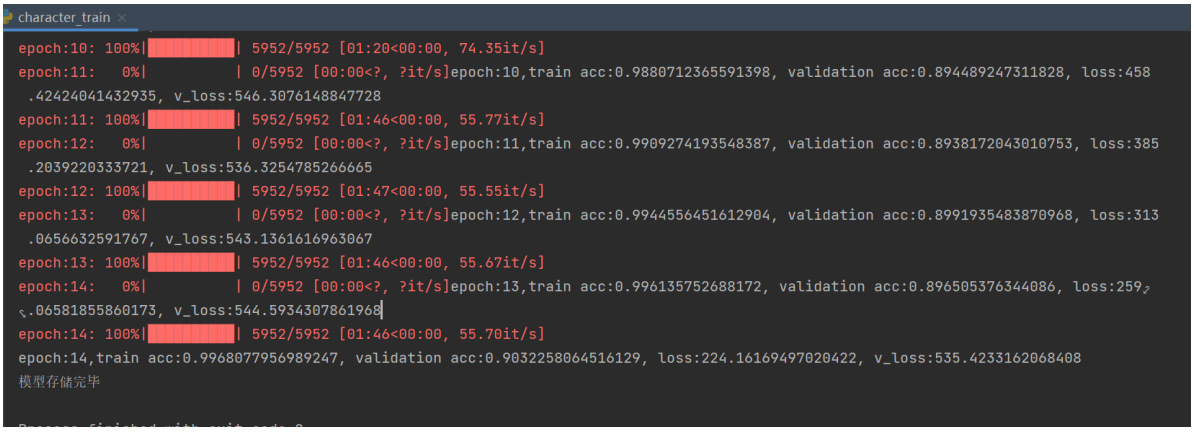
```
 99%|█████████▉| 493/500 [02:36<00:02,  2.97it/s]general loss = 6.750680639365242, average loss = 0.0016876701598413104
general loss = 6.7176674853352, average loss = 0.0016794168713338
 99%|█████████▉| 495/500 [02:37<00:01,  2.96it/s]general loss = 6.684754365995137, average loss = 0.0016711885914987843
 99%|█████████▉| 496/500 [02:37<00:01,  2.92it/s]general loss = 6.651940450284011, average loss = 0.0016629851125710027
 99%|█████████▉| 497/500 [02:37<00:01,  2.95it/s]general loss = 6.619224919078079, average loss = 0.0016548062297695198
100%|█████████▉| 498/500 [02:38<00:00,  2.99it/s]general loss = 6.586606965167677, average loss = 0.0016466517412919192
general loss = 6.554085793236344, average loss = 0.001638521448309086
100%|██████████| 500/500 [02:38<00:00,  3.15it/s]
general loss = 6.521660619840643, average loss = 0.0016304151549601607
模型存储完毕
```

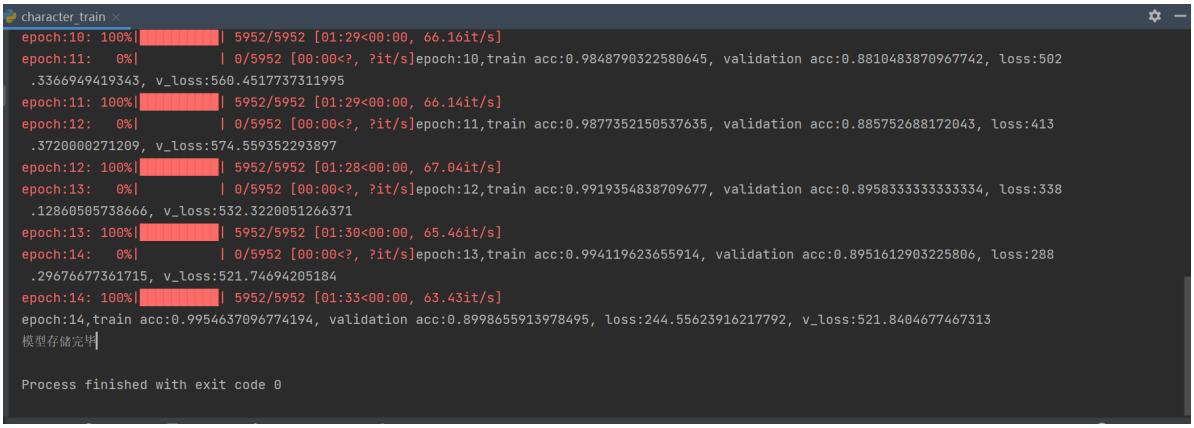
在sin_test.py中进行检测，画出预测的sin函数图像和真实的sin函数图像如下：



汉字识别

在character_train.py中进行训练，训练15个epoch，每个epoch会打乱给定的train的数据，采用其中的80%作为训练数据，剩下20%作为验证数据，经过多组网络参数的调整，在训练集上的准确率都能达到99%这样特别高的水平，但验证集上的准确率大致为90%。如下：





拟合sin函数的任务比较简单，网络一共三层，很容易就达到了很高的准确率，但是汉字识别的任务比较复杂，网络设置了四层，对于每层的参数设置为多少，我尝试了很多组，下面列一下部分组参数和对应的验证集准确率：

第1层输入、输出	第2层输入、输出	第3层输入、输出	第4层输入、输出	准确率
28×28,32×16	32×16,16×16	16×16,16×8	16×8,12	89.1%
28×28, 28×16	28×16,16×16	16×16,16×8	16×8,12	89.9%
28×28, 24×20	24×20,16×16	16×16, 14×12	14×12,12	88.5%
28×28,22×20	22×20,20×16	20×16,12×8	12×8,12	90.3%
28×28,24×24	24×24,24×24	24×24,16×16	16×16,12	88.9%

第四组参数准确率较高，达到了90%以上。

思考

1、反向传播那一块的梯度计算转化为代码稍稍有些复杂，尤其使用了矩阵乘法，需要注意数据对齐的问题

2、参数设置的初始化比较重要，好的参数能更快达到收敛，且准确率更高

3、我训练的网络在训练集上准确率接近100%，但是验证集上准备率始终在90左右上不去，感觉还是过拟合了，有什么办法可以再提高一些?

4、网络的架构很重要，好的架构使得网络扩展性高，参数也好调整，按照pytorch的架构去写，等于写了一些通用的模块，搭建其他网络也比较方便