# Part 1 反向传播算法

## 代码基本架构

## 总体结构

- 1. 所有有关神经网络的代码都放在NeuralNetwork文件夹下。
- 2. 部分函数、类参考了pytorch的设计方式
- 3. 采用面向对象设计,具体激活函数、网络层、损失函数都由对应基类派生而来。

#### 文件夹结构如下:

#### **NeuralNetwork**

- \_\_init\_\_.py
- ActFunc.py
- DataDealing.py
- Layer.py
- Loss.py

## 各文件说明

#### ActFunc.py

包含各激活函数的实现,如ReLU, Sigmoid, Softmax等。各激活函数继承于ActivationFunc基类,覆写运算和求导的两个虚函数。

DataDealing.py

包含有关数据处理的函数:载入图片、打乱数据、绘图、保存/加载模型的函数。

Layer.py

是实现神经网络核心文件。内部包含网络层以及网络的实现。全连接层FCLayer继承于Layer虚基类,覆写了正/反向传播、参数调整等函数。网络类Network,内含一个Layer数组以及网络训练的相关参数。

Loss.py

包含各个损失函数: Mean Absolute Error, Mean Square Error, CrossEntropyError。所有损失函数继承于虚基类Loss,覆写计算误差以及误差求导两个函数。

```
__init__.py
```

用于将以上四个py文件统筹到统一的命名空间下。

使用者通过 import NerualNetwork as nn 可以导入四个所有类。具体使用方式如下

```
import NerualNetwork as nn
nn.dl # 数据处理
nn.ls # 损失函数
nn.act # 激活函数
nn.* # 网络层和网络
```

## 构建网络 & 训练网络工作流

以汉字识别为例。

1. 导入网络的包

```
import NeuralNetwork as nn
```

2. 利用nn.dl导入训练、测试数据。

```
X_train, y_train, X_test, y_test = nn.dl.load_data(
    "train", num_train_samples=500, num_test_samples=120)
```

3. 制定好超参

```
# 分类问题自身的参数
pic_size = 28 * 28
char_class_num = 12

# 各层神经元的输入、输出数量
num1 = 128 #
num2 = 64
num3 = 32

batch_size = 1
epochs = 80
lr = 0.01
# 参数初始化的期望与方差
mean = 0
dev = 0.2
```

4. 创建网络层与构建网络

```
# 声明四个全连接层,最后一层Softmax
11 = nn.FCLayer(in_feature=pic_size, out_feature=num1,
                act_func=nn.act.ReLu, mean=mean, dev=dev)
12 = nn.FCLayer(in_feature=num1, out_feature=num2,
                 act_func=nn.act.ReLu, mean=mean, dev=dev)
13 = nn.FCLayer(in_feature=num2, out_feature=num3,
                 act_func=nn.act.ReLu, mean=mean, dev=dev)
14 = nn.FCLayer(in_feature=num1, out_feature=char_class_num,
                act_func=nn.act.Softmax, mean=mean, dev=dev)
# 构建网络
nw = nn.Network(loss_func=nn.ls.CE, batch_size=batch_size,
               lr=lr, epochs=epochs,)
nw.add(11)
nw.add(12)
nw.add(13)
nw.add(14)
```

5. 分类问题训练网络

#### # 训练结束会返回训练时间

total\_time = nw.classify\_train(X\_train, y\_train, X\_test, y\_test)

6. 使用nn.dl保存模型

```
save_name = "128-64-32_network"
nn.dl.save_model(nw,save_name)
```

7. 使用nn.dl读取模型

nw = nn.dl.load\_model(model\_name)

# 不同网络结构、超参对比

## Part 1: 拟合Sin的结构、超参对比

## 对比总述

所有结果记录在了Sin网络结构效果对比.xls当中。

					Size=1 Rol				194	結构:1-	30-20-1	BELCE_31		
	farris	Insim	heim	im-Inima	Instruction Pin	Tier	7		farin	In-in-	In-in-	andmin	Institution File	lier
	1700	1.3171000	6.001700			122			-	3.71301004	0. 00030100			71
1-1-10	7000	2.541905	0.075611	M		231	1:-4.	1-1-10	2000	s. namena	0.000000		5	231
	1	6. ETCVINI	6.000CW	Myn		es es			400	S. DOWNERS	0.00200		(M)	817
	_	6.5303508	0.0030314	M		1358				0.6138369	0.000707		M	1039
1-1-11		Ter-terr	Ingtion	in-form,	Instruyio	1100	14			Tor-Lane	Injeton	ian-Inimy	InstruMir	The
	>==	7.19777000	6.67/6797			xe			2000	0.7070303	8. 800017			329
	-	£ MATTHER	6.000075	N		Ess		16-1-11	400	6.51656751	0.00710704			629
	17000	1.018100	6.002002			661		1		2.56666071	0.07566601			401
				ALTERNA	DETT250		1		17000	3.776(869)	0.0075037	A TH	rf 7N	1583
	30000	3.77111961	6.0771118			1000	BC-CBT X (MCBC)	,	1300					1980
,	н			0-1 Batch	Size=1 Rol		を含っております。 をつか。近日ではあり ・で使える金を知らなる					Batch_Si	re=1 Signoi	
		1落結內:	1-20-1		Size=1 Rel	LU	まで、心臓を大工者の基金 からからはこのなか。 下便力の基金を取ります。		HS	(結构: 1-	20-10-1		re=1 Signoi	
		(集結為:	1-20-1			LT Har	京也・近日下下了原始を付 からか。かけ、大田田か ・可比大田本春七郎・中田		H2	(結构: 1-	20-10-1	im-Inion		d Her
le-1.10		路站荷: 23.00960	1-20-1			E E E E E E E E E E E E E E E E E E E	まで、必要するである場合である。 かっからからないません。 ではないないないない。 ではないないないないない。	1-4.411	H2	新海: 1 — Tex-Sess 14 etzen	20-10-1	im-Inion		ed There
tr-4.00		(基础内: 20.000000 8.000000	1-20-1			30 20 20 and	が、小田下下戸山内 で、小田下下戸山内 ・「地下の本生」の地方		H2	No. 100	20-10-1 10-3100 1-10-3100	5		No. 300
1-1.111	_	本統有: 23.000960 8.000000	1-29-1 11-22-1 1 HOUSE			13 to 20 do 40 do	Entrangle (State Control of State Contro		H2	A E R : 1 — TAY-TABLE IL ETIME IL ETIM	20-10-1 12-222 1 090301 1 000000	5		the see
1-1.111	_	本統有: 23.000960 8.000000	1-20-1 10-3334 6.003006 6.0003006			30 20 20 and	SC-CETT FACES	1-4-111	H2	14 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	20-10-1 10-3100 1-10-3100			700 700 300 601
	_	25. 002960 25. 002960 3. 004000 3. 004000	1-29-1 11-24-1 1 SENION 1 SENION 1 SENION 1 SENION			10 10 10 10 10 10 10 10 10 10 10 10 10 1	ST. CST. I PAGE Virtual (STEEL) VIRTUAL STEEL		- -	14 000000 14 000000 2 0000000 3 0000000	20-10-1 112-1112 1 000000 1 000000 1 1000000			700 NO COLUMN THE COLU
	_	23, 630,000,00 23, 630,000,00 26, 630,000,00 17, 800,000,00 14, 100,000,00	1-29-1  LUNCON  LUNCON  LUNCON  LUNCON  LUNCON  LUNCON  LUNCON			20 20 20 EEE EEE EEE EEE EEE EEE EEE EEE	Br. Jap I Jackson Street and Total Control of the C	1-4-111	——————————————————————————————————————	14 0000000 1. 00000000 2. 00000000 3. 0000000 3. 0000000	20-10-1 11-21-1 1-10-10-1 1-10-10-1 1-10-10-1 1-10-10-1 1-10-10-1 1-10-10-1 1-10-10-1 1-10-10-1			500 CO STATE OF STATE

总共对比了2种网络结构,

每种网络结构下又对比了ReLU和Sigmoid两种激活函数。

不同激活函数下对比了不同学习率、不同Epochs的学习效果。

## 对比结论

#### 网络结构对比

- 1. 两种网络结构1-30-20-1, 1-20-10-1都能出色的完成任务,使模型平均误差低至0.01一下(采用了MAE作为损失函数)。
- 2. 1-30-20-1 收敛速度比1-20-10-1 更慢,需要更多epochs才能使模型误差达标,但其优点在于收敛时的误差更小。这是符合直觉的,因为前者模型更复杂,需要更多训练,但也因此拟合效果更好,因而收敛时误差低。
- 3. 单层隐层的网络并没有写入记录表格中,但是在使用Sigmoid作为激活函数的情况下,也是可以较好拟合Sin的。

#### 激活函数对比

- 1. 从拟合结果图或者平均误差上可以非常明显的看出,**Sigmoid函数在拟合Sin的任务上远远好于 ReLU**,ReLU的拟合结果图甚至不是一个连续函数,而是一个分段函数。这可能是因为ReLU本身并 没有Sigmoid那样光滑,因而最终拟合出的函数出现不光滑甚至不连续的情况。
- 2. ReLU始终无法将平均误差降至0.01以下,最好情况也是大于0.03。Sigmoid可以非常轻松达成目标。
- 3. ReLU的平均误差降低过程十分"陡峭",Sigmoid函数降低过程十分平滑。
- 4. 同样的Epochs、结构下,ReLU训练速度比Sigmoid快大约10%~20%,虽然有ReLu比较简单的因素在,但也与我先前Sigmoid求导需先运算一遍函数结果有关(后来已经改正)。

#### 学习率对比

- 1. 学习率小, Loss曲线并**不一定**能更平滑。
- 2. 小学习率为了达到收敛花费了更多Epochs,但小学习率不一定能得到更好的拟合结果(在使用ReLU的网络中可以看出)

#### 训练次数对比

1. 一般来说Epochs越多误差越好,例如使用Sigmoid时的网络,但是在使用ReLU的网络中,Epochs 过多时反而有可能使Loss上升。

## Part 2: 汉字分类的结构、超参对比

### 对比总述

所有结果记录在了汉字识别全连接层分类.xlsx.当中。



总共比较了6种网络结构,包含单、双、三层隐层。

此外还比较了不同Batch\_Size,不同学习率,不同参数初始化的学习效果。

## 对比结论

#### 网络结构对比

- 1. 单层隐层、双层隐层、三层隐层的测试集上最终正确率均在85%至90%之间。表现最好的是'784-392-196-98-12',达到了90.1%的正确率。表现最差的是'784-128-12',正确率为85.97%。正确率提升了4%,但前者训练时间是后者训练时间的六倍左右。
- 2. Batch Size为1的情况下,所有网络的Is和cr曲线都非常的不平滑。

### Batch\_Size比较

- 1. bs =100 相比于bs =1 ls和cr曲线光滑的多。
- 2. bs更大训练速度会更快,(因为不用次次BP),但是收敛效果更差,因为采用了Batch内数据梯度的平均,并不能非常好的削减Loss。

#### 学习率比较

- 1. 同拟合Sin, 小学习率并不一定能带来更到的正确率, 但一定会付出更多的训练时间。
- 2. 在大学习率训练的末尾,采用小学习率,能获得略微更好的训练结果,约2%~5%,但始终很难达到90%正确率。

#### 参数初始化比较

本次全连接层参数都采用正态分布初始化,对比了(mean,var) = (0,0.2) / (0,0.1) / (0,0.01) 三组取值

1. 大部分网络结构var = 0.1 效果会比var = 0.2更好,但最高也只能高2%左右。 var=0.01会比var = 0.1略差1%。

在简单的网络里[784-128-12]下,var = 0.2更好。

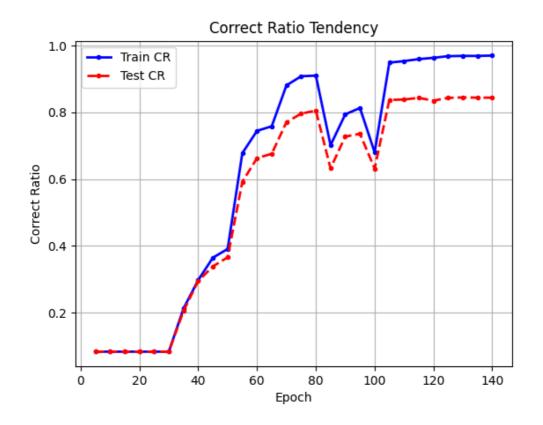
- 2. 其他更大var也有过测试:如果var接近1会直接无法收敛,var需要低于0.6才能正常的训练。
- 3. 更小的初始化值,最后用更小的Ir训练效果会更好。(见测试的两个两层网络),最后将Ir调至非常小(0.0001)正确率都能略微提升。
- 4. 三者的ls, cr曲线都非常的不平滑。

#### 激活函数

没有写在测试结果文件里,但是也有测试,比较下来也可以实现85%不到的正确率,相比于ReLU来说效果差些。并且在最初的Epochs里正确率会固定在8.3%。

此外相比于ReLU, Sigmoid更容易发生梯度消失的问题,不过好在目前测试的网络结构尚浅,问题不是很严重。对于梯度其更大影响的反而是参数初始化。

下面的图时Sigmoid作激活函数的cr图



## 反向传播算法理解

反向传播本质上基于链式求导法则,但相比于单纯运用链式求导,用到了前后层之间梯度的递推,这降低了求解的复杂度。

否则一个n层的网络,在第一个隐层的梯度需要求n-1个连续的Σ,这个复杂度是难以想象的。

下面手动推导一遍反向传播算法(先不考虑BatchSize)

#### 首先约定参数:

符号	意义
If	本层输入的数量(input feature)
Of	本层输出的数量,同时也是本层神经元数量(output feature)

符号	意义
Х	本层输入矩阵,大小为[lf,1]
ω,b	权重[If, of],偏置[of,1]
О	本层输出[of,1]
f	本层激活函数
OF	本层输出经过激活函数[of,1]
Е	损失函数

对于每层,前向传播时,有:
$$O=\omega^Tx+b$$
  $Ofunc=f(O)$  对于所有层,参数更新时都有: $\omega=\omega-lr\cdot 
abla \omega$   $b=b-lr\cdot 
abla b$ 

## 求解ω的梯度

$$egin{aligned} rac{\partial E}{\partial w} &= egin{bmatrix} rac{\partial E}{\partial w_{11}} & \cdots & rac{\partial E}{\partial w_{1Of}} \ dots & \ddots & dots \ rac{\partial E}{\partial w_{If1}} & \cdots & rac{\partial E}{\partial w_{IfOf}} \end{bmatrix} \ &= egin{bmatrix} rac{\partial E}{\partial w_{If}} & \cdots & rac{\partial E}{\partial w_{IfOf}} \end{bmatrix} \ &= egin{bmatrix} rac{\partial E}{\partial O_j} & rac{\partial O_j}{\partial w_{ij}} \end{bmatrix} \ &= egin{bmatrix} rac{\partial E}{\partial O_j} & x_i \end{bmatrix} \ &= m{x} \left[ rac{\partial E}{\partial O} \right] & (rac{\partial E}{\partial O}) \end{aligned}$$

所以只要求出E关于Output的梯度即可。而

$$\begin{split} \frac{\partial E}{\partial O_i} &= \frac{\partial E}{\partial OF_i} \frac{\partial OF_i}{\partial O_i} \\ \text{因此} \left[ \frac{\partial E}{\partial O_i} \right] &= \left[ \frac{\partial E}{\partial OF_i} \right] \cdot \left[ \frac{\partial OF_i}{\partial O_i} \right] \\ &= g\_OF \cdot g\_ActFunc \\ &= g\_O \end{split}$$

g表示gradient。g\_ActFunc是比较好求的,问题在于g\_OF。

OF是本层的输出,同时又作为下一层的输入,对下一层所有的神经元产生影响,最终作用到E上。于是g\_OF可以求了。先求对于本层第n个输出的偏导。

符号带prime的为下一层的数据。

$$\begin{split} \frac{\partial E}{\partial OF_n} &= \sum_{i=1}^{Of'} \frac{\partial E}{\partial O_i'} \cdot \frac{\partial O_i'}{\partial OF_n} \\ &= \sum_{i=1}^{Of'} \frac{\partial E}{\partial O_i'} \cdot \omega_{ni}' \\ &= \left[ \frac{\partial E}{\partial O_i'} \right] \cdot \omega_n' \end{split}$$

因此对于g\_OF矩阵:

$$g\_OF = \left[\frac{\partial E}{\partial OF_i}\right]$$
$$= \omega' \left[\frac{\partial E}{\partial O_i'}\right]$$
$$= \omega' q\_O'$$

现在求解ω梯度所有的问题都解决了, 连起来:

$$egin{aligned} 
abla \omega &= oldsymbol{x} \left[ rac{\partial E}{\partial O} 
ight] \quad ($$
矩阵乘法 $) \ &= oldsymbol{x} \left( g\_OF \cdot g\_ActFunc 
ight) \ &= oldsymbol{x} \left( \left( \omega' g\_O' 
ight) \cdot g\_ActFunc 
ight) \end{aligned}$ 

因此在写代码的时候,我们需要记录每一层的g\_O,借助下一层的g\_O可以计算本层的g\_O同时计算本层ω的梯度。

需要特别注意的是距离输出层的g\_O,因为它没有下一层了,不过此时g\_O已经可以借助损失函数E直接计算了。

特别的,在代码的实现上需要区分最后一层和其他层的反向传播。

#### 求解b的梯度

每一层的b相当于始终连着一个输入为1的神经元。因而上面的x可以直接替换成1。

因此

$$\nabla b = 1 \left[ \frac{\partial E}{\partial O} \right] \\
= g\_O$$

所以b的梯度可以非常轻松的通过g\_O得到。

### 总结

- 1. 每层记录本层关于输出的梯度的g\_O, ω和b的梯度由g\_O计算而来。
- 2. 上层g\_O由下层g\_O推导而来。输出层g\_O需要特别求解。

## 实验中遇到的问题及解决

## Softmax求导

由于Softmax和CrossEntropy搭配可以使得求导十分方便,最后导数直接是

$$g\_Loss = \hat{y} - y$$

但从代码实现的角度上来看,Softmax属于激活函数,CE属于Loss Function,两种是不同的类。导数应该分别计算。但这样的话就没法利用到上面这个优美的求导了。

人为规定Softmax求导恒定为1, CrossEntropy求导为上面的公式。

优点是利用了上面优美的式子,缺点是强制是的Softmax和CrossEntropy绑定,并且必须把Softmax放在最后一层。这个约束只能靠使用者自行遵守了。

## 汉字分类正确恒定1/12

在做汉字分类时,我发现无论怎么训练,在验证集上的正确率始终为8.3%,也就是 1/12 (见下图红框)。同时非常神奇的是在训练集上BP与不进行BP,计算出来的正确率是不同的(见下图黄框)。

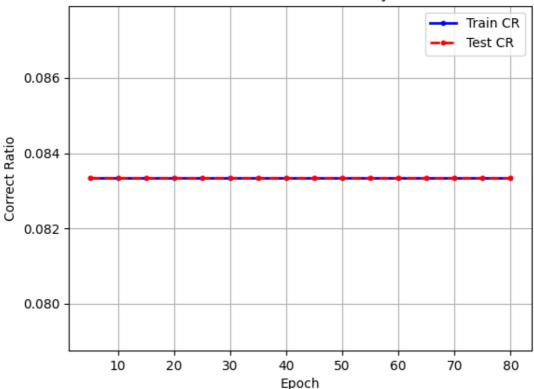
```
----epoch: 18----
0.07783333333333334
       epoch: 19
0.08366666666666667
Test Loss: 3579.9205081185537 Avg Loss: 2.486055908415662 Correct Ratio: 0.0833333333333333333 Progress: 20 / 80 Train Loss: 14916.33545049447 Avg Loss: 2.4860559084157448 Correct Ratio: 0.083333333333333333 Progress: 20 / 80
-----epoch: 20-----
0.0775
-----epoch: 21-----
0.0805
      --epoch: 22-----
0.0765
-----epoch: 23-----
0.08333333333333333
 -----epoch: 24---
0.07483333333333334
Test Loss: 3579.305894702096 Avg Loss: 2.485629093543122 Correct Ratio: 0.0833333333333333 Progress: 25 / 80
-----epoch: 25-----
```

可以看到Loss反复的震荡,事实上无论怎么训练,loss不会低于2.4。



无论是在训练集还是在验证集上,正确零雷打不动。

## Correct Ratio Tendency



#### 原因及解决方法

我打印了训练过程中所有的梯度以及输出,发现模型自初始化后,无论遇到什么图片,输出的都是同一个分类。这解决了为什么正确率是8.3%。

至于BP与不BP正确率不同,很有可能是BP的过程中改变了模型输出的那个分类,但仍然无法做到区分图片。改变之后,模型可能会判断对新的图片,但是对于绝大部分图片来说,仍然是错误的。这造成了训练时正确率不同于8.3%。

解决办法非常简单,这实际上是由于参数初始化过大造成的。由于我使用的是ReLU,当神经元输出小于 0时,激活函数的导数将会变成0,这导致该神经元不会更新。初始化参数在一个比较小的范围内使得更 多的神经元在训练过程种不会'死亡'。事实上(mean,dev)=(0,1)就已经会使得训练没有任何效果了(甚至 还会报错)。

### 连续训练-绘图连续性问题

得益于Pickle库的功能强大,能够使用Pickle库的dump函数将整个模型直接保存,也能直接读取整个模型,这省去了很多麻烦。保存模型使得模型可以复用,但也需要解决连续训练模型(紧接上一次模型训练结果接着训练)的问题。主要问题是绘制最后loss和correct\_ratio的图信息不完全,此外还有当前epoch打印、训练时间打印的问题。

#### 解决方法

将test\_loss和train\_loss, test\_cr和train\_cr都设为Network的成员变量,在验证过程当中,将相应信息 append进对应数组。由于这些重要数据已经成了Network的属性,因而能一并保存与读取。最后绘图也保留之前训练的结果。

至于epoch和训练时间的打印,需要额外多存储lastEpoch,lastTime两个属性,记录之前训练了多少epochs和时间,每次训练完之后进行更新。

上述两者结合,使得读取模型后,之前的相关训练信息也能得以保留,并最终可视化的展现在用户眼前。

# 目前对神经网络的看法

神经网络让人们避免了思考解决复杂问题的算法,或者说神经网络本身就是一种通用的算法。

但是从我自身训练的经历来看,只是从思考解决方法转变成了玄学测试超参。这也算是一种进步吧,毕竟解决问题的"算法空间",比神经网络的"超参空间"要来的大得多,也更难想。后者只不过是一堆数字罢了。