第一部分：反向传播算法

代码基本架构

Requirements: 除了常见的numpy，matplotlib等库之外，本项目额外使用了：

- tqdm来显示训练的进度
- pydantic作为类的解析验证库

设计了Layer，Nonlinearity，Loss等类作为基础；使用Model类组装这些线性层形成MLP。为了训练，设计了一个BasicRoutinizer的类。BasicRoutinizer接受一个Model类的对象，并进行训练。所有与模型训练相关的超参数都应该在BasicRoutinizer中调整；Model中调整的是模型本身的结构。

为了能够进行反向传播算法，每一个Layer，Nonlinearity，Loss类中都至少应该有forward和backward两种方法：前者正向地进行推理，后者是实现反向传播的基础：由于我们的MLP并不涉及到residual连接等额外的结构，梯度事实上线性地在模型中从后向前地传播。由于矩阵求导的知识，很容易知道：例如在某一线性层（$Y = WX + B$）处损失函数对权重的导数 $\frac{\partial L}{\partial W}$ 就是损失函数对结果的导数 $\frac{\partial L}{\partial Y}$ 同结果对矩阵的导数 $\frac{\partial Y}{\partial W} = X$ 之积。只要矩阵的形状能够符合运算规则，就能得到预期的导数。对偏差的导数类似。为此，反向传播算法事实上需要记录：

- Upstream derivative（$\frac{\partial L}{\partial Y}$），应该作为参数传入
- Local tensor（X），应该被每一个基础类自己在forward的过程中保存

一旦在模型的每一处都获得了Upstream derivative和Local tensor，就可以计算得到所有权重和偏差的导数。

在BasicRoutinizer中，利用优化器可以对模型进行优化。我们设计了SGD和Adam两种优化方法。SGD利用导数的信息直接进行更新；Adam则利用到了momentum的思想，能够在一定程度上自适应地更改lr的值。更加常用的是AdamW，即加入了weight decay的Adam，但这里没有实现。

下面是更具体的代码说明。

1. Layer

每一实例中存储了大量的数据，例如bias，weight，in_tensor，d_weight，d_bias，d_in_tensor等权重和导数相关的矩阵，saved_weight，saved_bias等用于保存模型的矩阵，momentum1，momentum2等会被Adam实时更新的值也被储存。此外，一些例如d_in，d_out，random_seed之类的基础信息也会被储存。

```
1 # mlp.py
2 class Layer(BaseModel):
```

```
    ...
    class Config:
        arbitrary_types_allowed = True

    def initialize(self):
        ...

    def forward(self,tensor:np.ndarray,train:bool=True):
        # The input tensor should be shape like (Features,Nums)
        if train is True:
            self.in_tensor = tensor
        out = np.dot(self.weight,tensor) + self.bias
        if train is True:
            self.out = out
        return out

    def back_propagate(self,upstream_derivative:np.ndarray):
        self.d_weight = np.dot(upstream_derivative, self.in_tensor.T)
        self.d_bias =
np.dot(upstream_derivative,np.ones((np.size(upstream_derivative,1),1)))
        self.d_in_tensor = np.dot(self.weight.T, upstream_derivative)
        return self.d_in_tensor
```

加入train的布尔值，是由于我们在训练中希望能够实时地看到在validation集上的表现，而若同时进行，则训练和推理的表现不同：推理时不能记录Local_tensor，否则会影响反向传播。

2. Nonlinearity

我们只实现了ReLU的激活函数，因为ReLU是目前最广泛被运用的激活函数，其求导方便，而且不会出现梯度爆炸现象。同样加入train的布尔值，在推理时不记录Local tensor，在训练时记录。

```
# mlp.py
class Nonlinearity(BaseModel):
    type:str = 'ReLU'
    in_tensor:Optional[np.ndarray] = None

    class Config:
        arbitrary_types_allowed = True

    def forward(self,tensor:np.ndarray,train:bool=True):
        if train is True:
            self.in_tensor = tensor
        if self.type == 'ReLU':
            tensor = np.maximum(tensor, 0)
            return tensor
```

```
        else:
            print(f"Don't support nonlinearity type:{self.type}\n")

    def back_propagate(self,upstream_derivative:np.ndarray,in_tensor:np.ndarray):
        # We only inplement the ReLU nonlinearity
        # f(x) = max(0,x)
        if self.type == 'ReLU':
            downstream_derivative = upstream_derivative * np.where(in_tensor >= 0,1,0)
        else:
            print(f"Don't support nonlinearity type:{self.type}\n")

        return downstream_derivative
```

3. Loss

实现了MSE和CrossEntropy两种。需要注意的是，CrossEntropy实现时必须注意数值稳定性，由于算出的logits的值往往较大，执行指数运算时会出现NaN；此外，在计算导数时，分母也会出现过小的情况。必须进行处理。

事实上，我们可以证明 $\text{softmax}(v) == \text{softmax}(v - \text{np.max}(v))$，这样就不会出现过大的指数运算；此外，对于分母过小的问题，我们可以将softmax和crossentropy的运算结合到一起进行求导，就可以规避这个问题。

```
# mlp.py
class Loss(BaseModel):
    type:Optional[str] = 'MSE'    # 0.5(in-res)^2; in-res
    input:np.ndarray
    result:np.ndarray
    probas:np.ndarray = None
    class Config:
        arbitrary_types_allowed = True

    def forward(self):
        if self.type == 'MSE':
            # Here, self.result should be a (C,N) vector
            result = 0.5 * np.square(np.subtract(self.input, self.result)).mean()
            return result

        elif self.type == 'CrossEntropy':
            N = np.size(self.input,1)
            max_x = np.max(self.input,0)
            x = self.input - max_x
```

```
            exp_logits = np.exp(x)
            sum_exp = np.sum(exp_logits,0)
            self.probas = exp_logits/sum_exp
            modified_x = x - np.log(sum_exp)
            t = np.c_[self.result.T,range(N)]
            result = -np.mean(modified_x[t[:,0],t[:,1]])
            del max_x,x,exp_logits,sum_exp,modified_x,t
            return result

    def back_propagate(self):
        if self.type == 'MSE':
            result = np.subtract(self.input, self.result)
            return result

        elif self.type == 'CrossEntropy':
            # We use the cross-entropy softmax loss, although Model.final_ans is logit, not probas
            # Still, let us use one-hot matrix as it is very useful
            N = np.size(self.input,1)
            onehot = np.zeros_like(self.probas)
            t = np.c_[self.result.T,range(N)]
            onehot[t[:,0],t[:,1]] = 1
            result = self.probas - onehot
            del onehot,t
            return result
        pass
```

4. Model

可以看到，在下面的代码里，额外有一些关于dropout层的逻辑，关于具体dropout的实现，将在后文描述。

Model的一个实例中保存了自己的layer列表，dropout列表和nonlinearity类型（由于我们的模型只支持MLP，就没有将线性层，激活层等作为平等的layer保存到一个列表中，相反，在model中手动地调用线性层和激活层），lr和dropout等会用于update的参数，loss列表用于记录loss。

此外，也实现了保存ckpt的方法。

```
# mlp.py
class Model(BaseModel):
    ...
    class Config:
        arbitrary_types_allowed = True

    def initialize(self):
        ...
```

```
 9
10      def resume_from_ckpt(self,o_path:str="./ckpt"):
11          ...
12
13      def forward(self,tensor:np.ndarray,train:bool=True ):
14          if self.debug is True:
15              print("A new iter:\n")
16          for m in range(len(self.layer)):
17              l =self.layer[m]
18              tensor = l.forward(tensor=tensor,train=train)
19              if m<=len(self.layer)-2:
20                  tensor = self.nonlinearity.forward(tensor=tensor,train=train)
21              if train is True:
22                  try:
23                      tensor = self.dropout_layer[m].forward(tensor)
24                  except:
25                      pass
26          if train is False:
27              return tensor
28          self.final_ans = tensor
29          return tensor
30
31      def get_final_ans(self,type:str = 'softmax'):
32          ...
33
34      def
    cal_loss(self,real_data:np.ndarray,type:str='MSE',train:bool=True,val_input=Non
    e):
35          if train is True:
36              self.loss = Loss(type=type,input=self.final_ans,result=real_data)
37              return self.loss.forward()
38          else:
39              return Loss(type=type,input = val_input,
    result=real_data).forward()
40
41
42      def back_propagate(self):
43          upstream_derivative = self.loss.back_propagate()
44          for i in range(len(self.layer_size)-1):
45              # The current layer
46              current_layer = self.layer[len(self.layer_size)-2-i]
47              if i > 0:
48                  if self.dropout is not None:
49                      c_dropout_layer =
    self.dropout_layer[len(self.layer_size)-2-i]
50                      upstream_derivative  =
    c_dropout_layer.back_propagate(upstream_derivative=upstream_derivative)
```

```
                upstream_derivative =
self.nonlinearity.back_propagate(upstream_derivative=upstream_derivative,in_ten
sor=current_layer.out)

upstream_derivative=current_layer.back_propagate(upstream_derivative)

    def
update(self,method:str='Adam',num_iter:int=0,beta1=0.9,beta2=0.999,debug=False)
:
        if method == 'SGD':
            for i in self.layer:
                i.weight = i.weight - self.lr * i.d_weight
                i.bias = i.bias - self.lr * i.d_bias

                if (self.debug is True):
                    print(f"[debug]:weight shape: {i.weight.shape}\n")
            if (self.debug is True):
                    print(f"[debug]:weight shape: {self.layer[-1].weight}\n")
        elif method == 'Adam':
            num_iter += 1
            for i in self.layer:
                i.momentum1 = beta1 * i.momentum1 + (1-beta1) * i.d_weight
                i.momentum2 = beta2 * i.momentum2 + (1-beta2) * i.d_weight *
i.d_weight
                momentum1_unbias = i.momentum1 / (1 - beta1 ** num_iter)
                momentum2_unbias  = i.momentum2 / (1 - beta2 ** num_iter)
                i.weight -= self.lr * momentum1_unbias /
(np.sqrt(momentum2_unbias) + 1e-8)

    def save_current_weight(self):
        ...

    def save_to_ckpt(self,o_path:str="./ckpt"):
        ...
```

5. BasicRoutinizer

可以用于设置训练相关的超参数，并进行具体的训练。

```
# mlp.py
class BasicRoutinizer(BaseModel):
    ...

    class Config:
        arbitrary_types_allowed = True
```

```
7
8       # Should save a best model!
9       def run(self,m:Model):
10          m.lr = self.lr
11          reach_min_loss = 0
12          val = False
13          if self.val_feature_data is not None and self.val_label_data is not
    None:
14              val = True
15
16          if self.dynamic_show is True:
17              fig = plt.figure()
18              plt.ion()
19          for i in tqdm(range(self.max_iter)):
20              rot = 0
21              if self.batchsize == None:
22                  rotN = 1
23              else:
24                  rotN = np.ceil(np.size(self.feature_data,1) / self.batchsize)
25
26              for j in range(int(rotN)):
27                  if self.batchsize == None:
28                      feature_data = self.feature_data
29                      label_data = self.label_data
30                  else:
31                      feature_data =
    self.feature_data.T[rot:rot+self.batchsize].T
32                      label_data = self.label_data.T[rot:rot+self.batchsize].T
33                      rot = rot+self.batchsize
34
35                  m.forward(feature_data)
36                  if val is True:
37                      val_predict = m.forward(self.val_feature_data,train=False)
38
39                  loss = m.cal_loss(label_data,type=self.type)
40
41
42                  m.back_propagate()
43
    m.update(method=self.method,num_iter=i,debug=True,beta1=self.beta1,beta2=self.b
    eta2)
44
45
46                  self.loss.append(loss)
47                  if val is True:
48                      val_loss =
    m.cal_loss(self.val_label_data,type=self.type,train=False,val_input=val_predict
```

```
    )
                    self.val_loss.append(val_loss)
                if self.save_ckpt is True:
                    if val_loss < self.min_loss:
                        reach_min_loss = 1
                        self.min_loss = val_loss
                        m.save_current_weight()

            if self.dynamic_show is False and i % self.dynamic_show_res == 0:
                print(f"{i}th iteration: Loss is {loss}")
            if self.dynamic_show is True and i % self.dynamic_show_res == 0:
                # Plot
                # ...

            self.final_ans.append(m.final_ans.ravel().tolist())

        if self.dynamic_show is True:
            plt.ioff()
            plt.show()
        if self.save_ckpt is True:
            if reach_min_loss == 0:
                print(f'Never reach at expected loss {self.min_loss}\n')
                m.save_current_weight()

            m.save_to_ckpt(o_path=self.path)

    def print_loss(self):
        # Plot
        # ...
```

数据预处理和Dropout

1. 数据预处理

我们采取最简单的数据预处理办法，即使其分布近似标准正态分布。这往往能使训练更加顺利。不同于torch.vision中提供的逐通道的归一化处理，这里就简单地使用逐像素的归一化处理。

事实上，一些额外的数据预处理应该能极大地提升表现。例如向四周平移一个像素点（这就能将train set的规模扩大5倍）；由于MLP对这种平移是敏感的，这种简单的数据增强就能起到很好的作用。（与之相对，CNN对于平移这种简单的增强就不是敏感的）；此外，翻折，旋转等都是很有可能生效的数据增强手段。但由于这不是本项目的重点，没有实施。

2. Dropout

我们使用标准的暂退法，即在训练过程中，按概率丢弃一些节点。（需要乘以一个系数以保证期望不变），在推理过程中，dropout实际上没有影响。

dropout是一个正则化方法，能显著地减少过拟合的现象。其原因在于，训练过程中随机丢弃了一些节点，能防止神经网络过于依赖其中的某些节点，从而降低泛化性能。MLP由于并不稀疏，dropout往往有用；另一方面，类似CNN的结构，由于本身具有一定的稀疏性，再加入dropout可能没有明显的效果。

```python
# mlp.py
class DropoutLayer(BaseModel):
    dropout:float = 0.5
    mask:Optional[np.ndarray] = None

    class Config:
        arbitrary_types_allowed = True

    def forward(self,in_tensor:np.ndarray):
        # in_tensor.shape = (f,N)
        self.mask = np.random.uniform(0,1,in_tensor.shape)
        self.mask = np.where(self.mask>self.dropout,1/(1-self.dropout),0)
        out = in_tensor * self.mask
        return out

    def back_propagate(self,upstream_derivative:np.ndarray):
        derivative = upstream_derivative * self.mask
        return derivative
```

3. Shuffle

如果按照 `[1,1,...,1,2,2,...,2,3,3,...,12,12...12]` 的标签进行训练，有可能神经网络会记住这种空间结构，从而导致泛化能力下降。使用 `np.random.shuffle` 可以打乱。

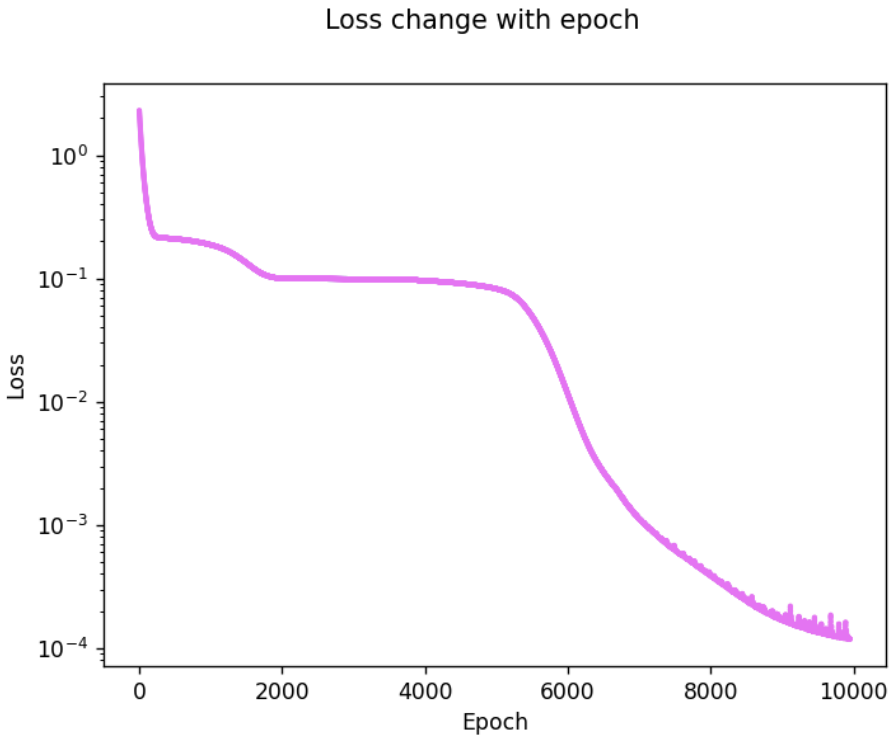
网络结构、参数的实验

1. 拟合三角函数

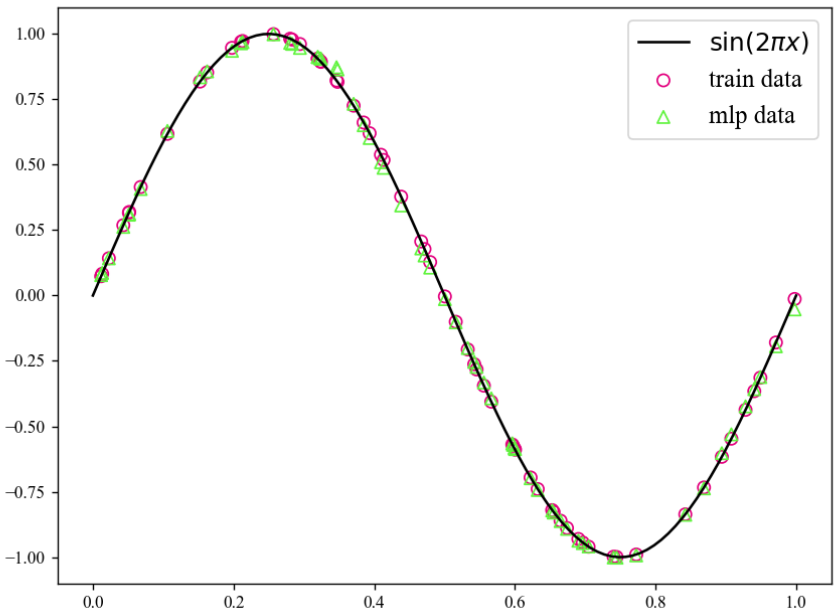
直接测试双层MLP，网络大小为100，训练代码如下：

```python
# train.py
from mlp import Model
from mlp import BasicRoutinizer
import numpy as np
from tqdm import tqdm

```

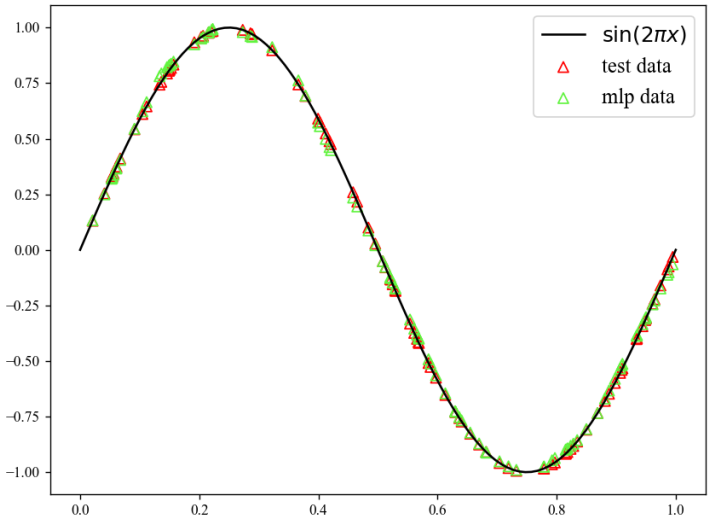
```
import matplotlib.pyplot as plt
from matplotlib import rc

test = np.random.random([9,20])
# test  =[-3,-5,-5,-6,-1,-64,-12,-123,-3]
# test = np.array(test).T

# Borrow toy data code from project in FDU PRML 2023 Lab 0
"""Create toy data"""
import math
# sin
def sin(x):
    y = np.sin(2 * math.pi * x)
    return y

def create_toy_data(func, interval, sample_num, noise = 0.0, add_outlier = False, outlier_ratio = 0.001):
    # ...

func = sin
interval = (0,1)
train_num = 64
test_num = 10
noise = 0
X_train, y_train = create_toy_data(func=func, interval=interval, sample_num=train_num, noise=noise)
X_test, y_test = create_toy_data(func=func, interval=interval, sample_num=test_num, noise=noise)
X_underlying = np.linspace(interval[0],interval[1],num=100)
y_underlying = sin(X_underlying)

max_iter = 10000

m = Model(layer_size=[1,100,100,1],debug=False)
m.initialize()
run = BasicRoutinizer(max_iter=max_iter,feature_data=X_train.T,label_data=y_train.T,lr=3e-4,dynamic_show=True,dynamic_show_res=50)
run.run(m=m)


# plot
# ...
```



loss的变化，可以看到一段平台期



训练集上的表现，loss已经降低到1e-4数量级



测试集上的表现，可以看到比较好地拟合了sin函数

可以看到，符合预期，实验成功。

2. 手写汉字分类

训练代码如下：

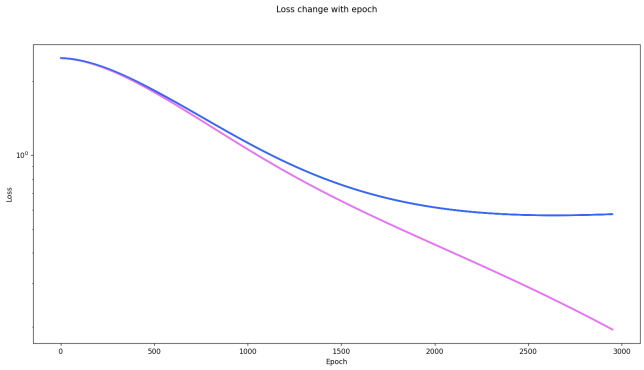
```
# bmptest.py
import matplotlib.pyplot as plt
from PIL import Image
from mlp import Model
from mlp import BasicRoutinizer
from preprocess import Data
import numpy as np
from tqdm import tqdm
import os
from matplotlib import rc

source_folder = './train'
val_source_folder='./validation'
input_matrix = []
label_vector = []
val_input_matrix = []
val_label_vector = []

for root, dirs, files in tqdm(os.walk(source_folder)):
    for file in files:
        file_path = os.path.join(root, file)
        digit = int(root.lstrip(source_folder + '\\'))
        I = Image.open(file_path,mode="r")
        a = np.asarray(I, dtype=int)
        a = a.reshape(-1,1)
```

```
        input_matrix.append(a)
        label_vector.append(digit-1)

for root, dirs, files in tqdm(os.walk(val_source_folder)):
    for file in files:
        file_path = os.path.join(root, file)
        digit = int(root.lstrip(val_source_folder + '\\'))
        I = Image.open(file_path,mode="r")
        a = np.asarray(I, dtype=int)
        a = a.reshape(-1,1)
        val_input_matrix.append(a)
        val_label_vector.append(digit-1)

input_matrix = np.array(input_matrix)
label_vector = np.array(label_vector).reshape(1,-1)
input_matrix = input_matrix.squeeze(2).T

val_input_matrix = np.array(val_input_matrix)
val_label_vector = np.array(val_label_vector).reshape(1,-1)
val_input_matrix = val_input_matrix.squeeze(2).T

d = Data(y_train=label_vector,X_train=input_matrix,y_val=val_label_vector,X_val=val_input_matrix)
d.shuffle()
d.normalize()

max_iter = 100
m = Model(layer_size=[784,512,512,512,12],debug=False,initial_scalar=0.01,dropout=0.5)
m.initialize()
#m.resume_from_ckpt()
run = BasicRoutinizer(max_iter=max_iter,feature_data=d.X_train,label_data=d.y_train,lr=3e-4,dynamic_show=True,dynamic_show_res=50,type='CrossEntropy',val_feature_data=d.X_val,val_label_data=d.y_val,min_loss=2.5,batchsize=256)
run.run(m=m)

```

2.1 首先实验[784,500,12]的单层MLP

在研究loss曲线前，我们应该首先明白这些loss的物理意义：在完全随机的情况下，probs中的每一个元素应该接近 $\frac{1}{12}$ ，由此算出的Cross Entropy损失为 $\ln(12) \approx 2.485$ ，我们的loss曲线应该每次都

从这个值附近开始下降。倘若某一时刻的loss为 l ，我们也能大致估计出大概的正确率约为 $\mathrm{Acc} \approx e^{-l}$ ，但实际的准确率并不一定会符合该公式。



单层MLP，未做标准化，没有dropout

此时的loss在0.6附近。

- 加入shuffle

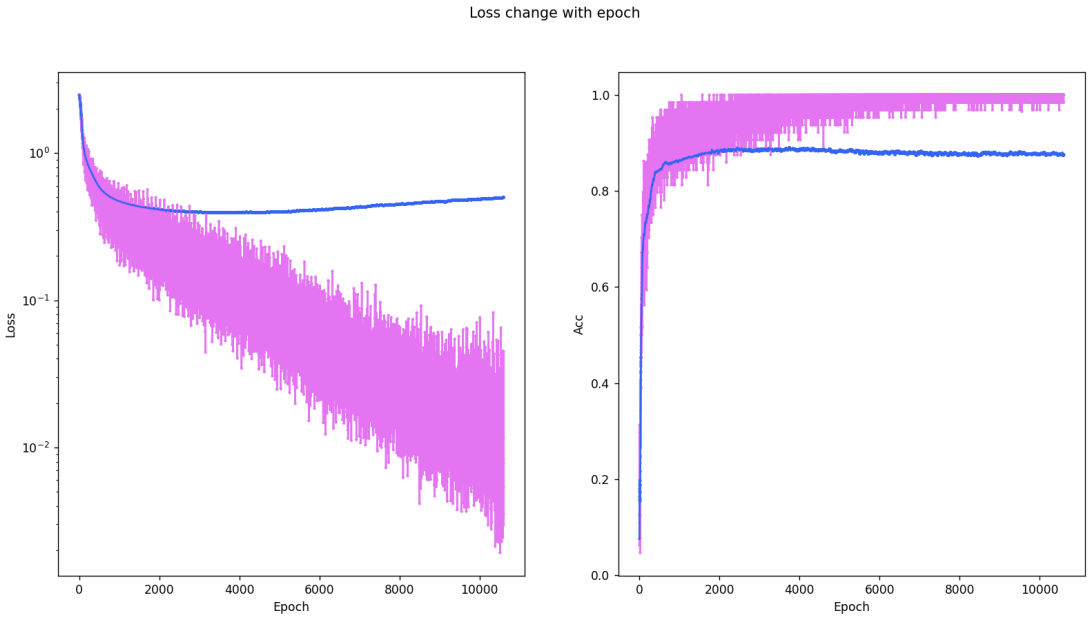
val_loss最低可以达到0.48412

- 加入normalization & dropout

val_loss在0.451232

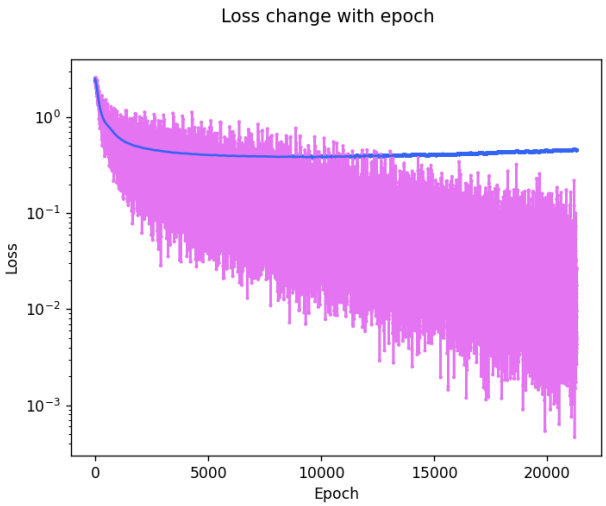
- 更改batchsize从全空间至64

val_loss在0.448左右，此时最高的准确率为89%

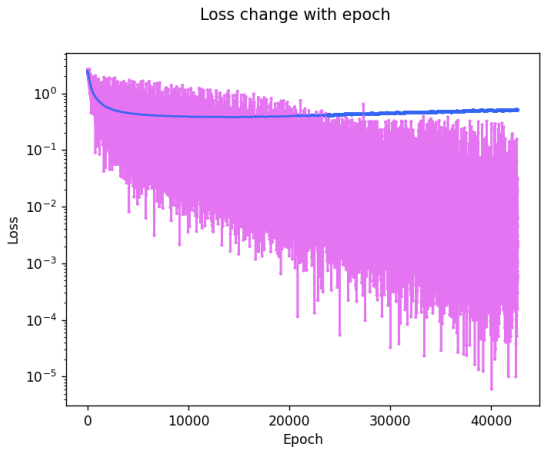


batchsize = 64，[784,500,12]

(继续更改batchsize可能会有更细微的提升)

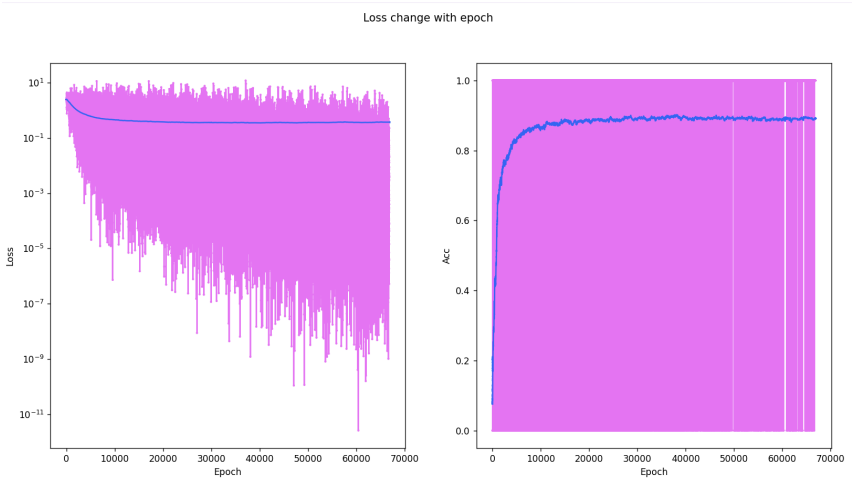


batchsize=16，min_val_loss = 0.382



batchsize=8,min_val_loss = 0.379

事实上，最为彻底的batchsize=1会导致最好的结果（尽管要花费更多的时间）我们就用batchsize=1进行一次实验，以得到最优的模型：



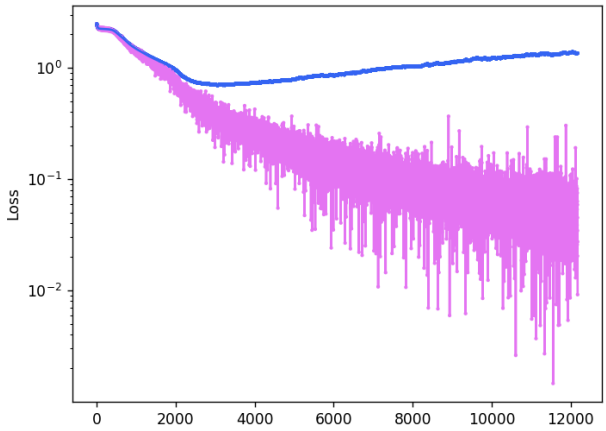
batchsize=1,min_val_loss = 0.353，max_val_acc = 90.2

即此时，**在验证集上有90.2%的准确率**

2.2 考察其它的网络结构

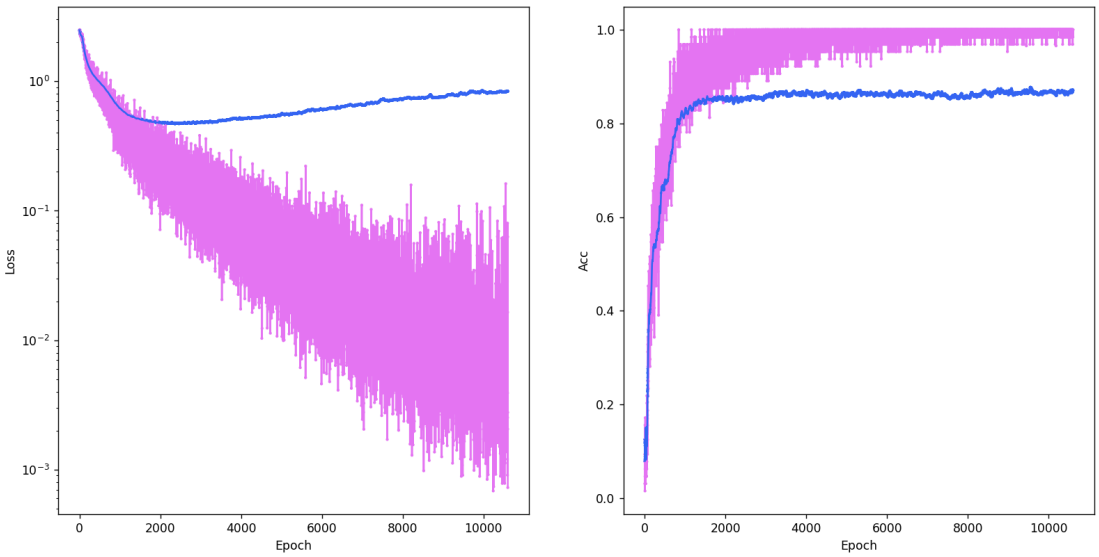
这里一律按shuffle & dropout & normalization & batchsize = 64进行处理

- 单层MLP：[784,256,23] loss 0.47 ； [784,64,12] loss 0.51
- 多层MLP：
  - [784,128,128,12] loss 0.7



[784,128,128,12]

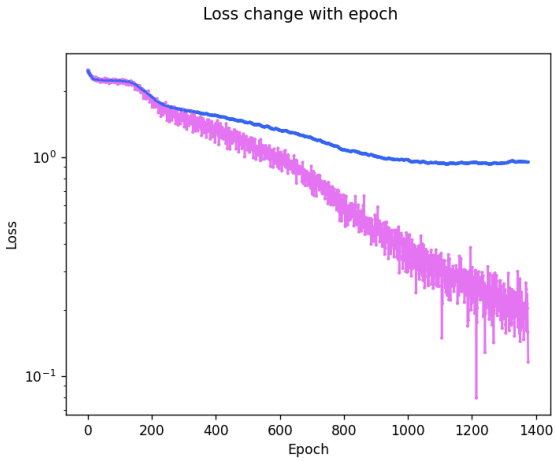
  - [784,512,512,12] loss 0.505



[784,512,521,12]，最大acc为87.7

可以看到，尽管并不是一一对应，但一般来说，更低的loss确实会产生更高的accuracy。

  - [784,512,512,512,12] loss 0.9



[784,512,512,512,12]

经过一些简单的尝试，没有找到更好的模型结构。如果模型更深，则训练难度会大大加大；同时泛化能力不保证提升。因此，决定的最好模型即为[784,500,12]，shuffle & normalization & dropout &batch_size = 1.

对反向传播算法的理解

在成熟的深度学习框架中，反向传播由计算图的形式计算，从而十分灵活。在我们自己实现的MLP中，只有单向的梯度流通，即计算图退化为一根链。此时实现起来就更加方便。

反向传播本质上是一种链式求导的方法，而在优化中，如何利用我们获得的一阶导数信息来进行优化，是更加关键的。最朴实也使用最多的SGD方法收敛的比较慢，但是它能够达到一个不错的精准度，因此直到今日，仍然被较为广泛地使用；除了SGD之外，还有许多方法想要更好地利用这种一阶导数信息。例如：SGD with Momentum，Nesterov Momentum，AdamGrad，RMSProp等。本项目使用的是较为广泛的Adam，基本上，其原理就是RMSProp + Momentum。Adam可以在一定程度上自适应地调整lr，从而加快收敛。

另一方面，对于激活函数的选取也十分有讲究。本项目直接采用了最为常见的ReLU激活函数（此外，还有一系列例如Leaky ReLU，SeLU，GeLU等相似的函数），而没有采用已经被淘汰的sigmoid激活函数和tanh激活函数。它们的计算开销比较大；此外，会出现所谓saturated gradient的现象，即在数值较大（小）时，梯度为0。ReLU系列的激活函数，计算开销小。然而，ReLU也会有Dead ReLU现象，即一旦小于0，则该神经元对应的权重就不会被更新。因此，如果使用SGD更新ReLU的神经网络，比较将学习率设置的很小，否则会有大批神经元死亡。训练十分困难。改用Adam后，训练得以继续。