Reproduction and Promotion of Single-cell RNA-seq denoising using a deep count autoencoder

Experiment Description

Denoise single-cell RNA-seq data. Single-cell RNA's RNA-seq is measured in biology experiments to analyse gene expression. Some RNA-seq cannot be accurately measured and is recorded as 0. However, there are also some genes actually do not express and their real value is 0. The work done in the original paper is using DCA to denoise single-cell RNA-seq data, that is, to judge whether the record 0 is generated from measurement or lack of expression, and to estimate the real value and fill in the approrariate position.

Experiment Requirements

- 基本要求

  将test_data.csv,test_truedata.csv分为测试集和验证集。实现任意补插算法来对数据集data.csv进行补插。使用测试集确定一个较好的模型，并使用验证集验证。针对你的模型和实验写一份报告，并提交源码(说明运行环境)和可执行文件。(建议使用神经网络)

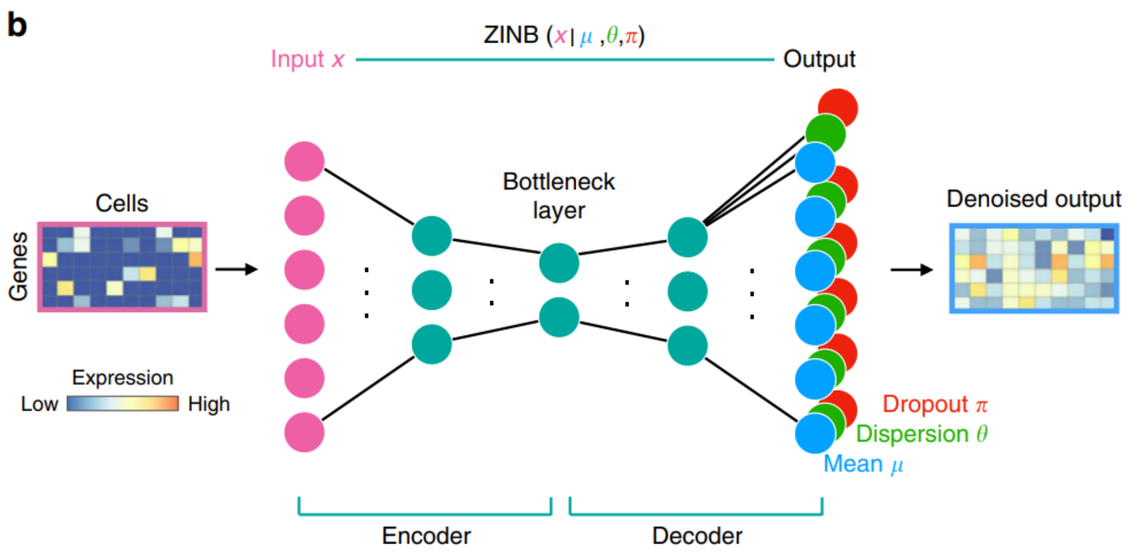
- 中级要求

  在test_data.csv上获得一定效果(dropout率变小，表达差异变小) 。在别人理论的基础上进行一些自己的修改。

- 高级要求

  对于data.csv的补插和生成data.csv的模拟真实数据集相比获得较好效果(dropout率变小，表达差异变小) 。

Experiment Principle



Model Construction

```
class AutoEncoder(nn.Module):
```

$$
\begin{aligned}
\boldsymbol{E} &= \operatorname{ReLU}(\bar{\boldsymbol{X}}\boldsymbol{W}_E) \\
\boldsymbol{B} &= \operatorname{ReLU}(\boldsymbol{E}\boldsymbol{W}_B) \\
\boldsymbol{D} &= \operatorname{ReLU}(\boldsymbol{B}\boldsymbol{W}_D) \\
\bar{\boldsymbol{M}} &= \exp\left(\boldsymbol{D}\boldsymbol{W}_\mu\right) \\
\Pi &= \operatorname{sigmoid}(\boldsymbol{D}\boldsymbol{W}_\pi) \\
\boldsymbol{\Theta} &= \exp(\boldsymbol{D}\boldsymbol{W}_\theta),
\end{aligned}
$$

- Input: count matrix A

  A_{ij} represents the expression the i gene in the j cell.

- Hidden Layer:

  E, B, D in the picture above reprensent encoding layers, bottleneck layer and decoding layers.

  All hidden layers except the bottleneck layer includes 64 neurons, and the bottleneck layer has 32 neurons.

- Output: The three parameters of every gene (μ, θ, π), that is (mean, dispersion, the probability of dropout), and is also the three parameters of ZINB distribution.

Specify the numbers of neurons per layer

```
def __init__(self, input_size=None, hasBN=False):
        """
        该autoencoder还可以改进:
        dropout层

        :param input_size:
        """
        super().__init__()
        self.intput_size = input_size
        self.hasBN=hasBN
        self.input = Linear(input_size, 64)
        self.bn1 = BatchNorm1d(64)
        self.encode = Linear(64, 32)
        self.bn2 = BatchNorm1d(32)
        self.decode = Linear(32, 64)
        self.bn3 = BatchNorm1d(64)
        self.out_put_PI = Linear(64, input_size)
        self.out_put_M = Linear(64, input_size)
        self.out_put_THETA = Linear(64, input_size)
        self.reLu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()
```

Specify the activation function

The activation function of THETA and M are both exp, this can make sure the output is legitimate because they are both non-negative values.

Use Sigmoid as the activation function, because it is the estimated dropout probability and it should be between 0 and 1.

```
def forward(self, x):
        x = self.input(x)
        if self.hasBN:x = self.bn1(x)
        x = self.reLu(x)
        x = self.encode(x)
        if self.hasBN:x = self.bn2(x)
        x = self.reLu(x)
        x = self.decode(x)
        if self.hasBN:x = self.bn3(x)
        x = self.reLu(x)
        PI = self.sigmoid(self.out_put_PI(x))
        M = torch.exp(self.out_put_M(x))
        THETA = torch.exp(self.out_put_THETA(x))
        return PI, M, THETA
```

Define the loss function of the model

$$\begin{aligned}\hat{\Pi}, \hat{\boldsymbol{M}}, \hat{\Theta} &= \operatorname{argmin}_{\Pi,\boldsymbol{M},\boldsymbol{\Theta}} \mathrm{NLL}_{\mathrm{ZINB}}(\boldsymbol{X}; \Pi, \boldsymbol{M}, \boldsymbol{\Theta}) + \lambda \|\Pi\|_F^2 \\ &= \operatorname{argmin}_{\Pi,\boldsymbol{M},\boldsymbol{\Theta}} \sum_{i=1}^{n} \sum_{j=1}^{p} \mathrm{NLL}_{\mathrm{ZINB}}(x_{ij}; \pi_{ij}, \mu_{ij}, \theta_{ij}) + \lambda \pi_{ij}^2,\end{aligned}$$

NLL is the negative logarithmic likelihood, and the ZINB formula is as below:

$$\mathrm{NB}(x; \mu, \theta) = \frac{\Gamma(x+\theta)}{\Gamma(\theta)} \left(\frac{\theta}{\theta+\mu}\right)^{\theta} \left(\frac{\mu}{\theta+\mu}\right)^{x}$$

$$\mathrm{ZINB}(x; \pi, \mu, \theta) = \pi\delta_0(x) + (1-\pi)\mathrm{NB}(x; \mu, \theta)$$

The ZINB distribution is chosen is because scRNA-seq data has many 0 values, and whose negative logarithmic likelihood is used as the loss function. When the loss function is minimum, the probability and the liklihood of ZINB are greatest. The training to be conducted is to reduce the loss and get the best estimation of ZINB parameters. Then use the estimated ZINB function to generate interpolation values.

Loss function definiton:

```
eps = torch.tensor(1e-10)
        THETA = torch.minimum(THETA, torch.tensor(1e6))
        t1 = torch.lgamma(THETA + eps) + torch.lgamma(X + 1.0) -
torch.lgamma(X + THETA + eps)
        t2 = (THETA + X) * torch.log(1.0 + (M / (THETA + eps))) + (X *
(torch.log(THETA + eps) - torch.log(M + eps)))
        nb = t1 + t2
        nb = torch.where(torch.isnan(nb), torch.zeros_like(nb) + max1, nb)
        nb_case = nb - torch.log(1.0 - PI + eps)
        zero_nb = torch.pow(THETA / (THETA + M + eps), THETA)
        zero_case = -torch.log(PI + ((1.0 - PI) * zero_nb) + eps)
        res = torch.where(torch.less(X, 1e-8), zero_case, nb_case)
        res = torch.where(torch.isnan(res), torch.zeros_like(res) + max1,
res)
        return torch.mean(res)
```

Input

The input function is as shown below. X is original counting matrix, and S_i is the factor of proportion of every cell.

$$\overline{X} = zscore(log(diag(s_i)^{-1}X + 1))$$

Because NLL estimation has been used, we need to preprocess infinite and invalid values in original dataset. Implement z-score normalization, that is, subtract the mean from the data and divide it by its standard deviation. The outcome of the normalization is that for every attribute, all data is clustered around 0 and is with a deviation of 1. The formula is: $x^* = \frac{x-\overline{x}}{\sigma}$

```
def preprocess_data(data: Tensor):
    gene_num = data.shape[1]
    s = torch.kthvalue(data,gene_num//2,1)
    s = 1/s.values
    norm_data = torch.matmul(torch.diag(s), data) + 1
    norm_data = torch.log(norm_data)
    norm_data = (norm_data - norm_data.mean()) / norm_data.std()
    return norm_data
```

Training

Use autoencoder for training, use batchsize = 32, that is, use the data of 32 cells for training each time.

```
def train(EPOCH_NUM=100, print_batchloss=False, autoencoder=None,
loader=None, startEpoch=0):
    """

    :param print_batchloss: if print train infor, False by default
    """
    opt = Adam(autoencoder.parameters(), lr=LR, betas=(BETA1, BETA2),
eps=EPS, weight_decay=WEIGHT_DECAY)
    # opt = SGD(autoencoder.parameters(), lr=1e-2, momentum=0.8)
    mean_loss=0
    for epoch in range(EPOCH_NUM+1):
        epoch_loss = 0

        for batch, batch_data in enumerate(loader):
            # batchsize = 32

            opt.zero_grad()
            train_batch = batch_data[0]

            # d: original matrix
            d = train_batch[:, :, 0]
            # norm_d: proessed data
            norm_d = train_batch[:, :, 1]
```

Forward and calculate loss value.

Follow the sequence of input to output, calculate and store intermediate variables of the model to train the next layer.

```
# forward
            PI, M, THETA = autoencoder(norm_d)
            templ = lzinbloss(d, PI, M, THETA)
            epoch_loss += templ
            if print_batchloss:
                print(f'epoch:{epoch+startEpoch},batch:{batch},batch loss:
{templ},(batch size {BATCHSIZE})')
                f.write(f'epoch:{epoch+startEpoch},batch:{batch},batch loss:
{templ},(batch size {BATCHSIZE})\n')

            # gradient descent
            opt.step()
        mean_loss+=epoch_loss
        if epoch % 100 ==0:
            mean_loss=mean_loss/100
            print(f'epoch:{epoch+startEpoch},epoch loss:{mean_loss}')
            f.write(f'epoch:{epoch+startEpoch},epoch loss:{mean_loss}\n')
            mean_loss=0
        # if epoch % EVER_SAVING == 0 and epoch!=0:
torch.save(autoencoder.state_dict(), open(f'0113epoch{epoch+startEpoch}.pkl',
'wb'))
        if epoch % EVER_SAVING == 0 and epoch!= 0 :
torch.save(autoencoder.state_dict(),
open(f'0113epoch{epoch+startEpoch}withoutBN.pkl', 'wb'))
```

Compute gradients with backpropagation

```
            templ.backward()
            clip_grad_norm_(autoencoder.parameters(), max_norm=5,
norm_type=2)
```

Considering the output PI,M,THETA, if the gene's corresponding PI>0.5, fill with corresponding M value as below:

```
 autoencoder.load_state_dict(torch.load(STATE_DICT_FILE))
    print(autoencoder)
    PI, M, THETA = autoencoder(norm_data)
    iszero = data == 0
    predict_dropout_of_all = PI>0.5
    # dropout_predict = torch.where(predict_mask, M, torch.zeros_like(PI))
    # print("after",after)

    true_drop_out_mask = iszero*((truedata - data)!=0)
    predict_dropout_mask = iszero*predict_dropout_of_all
    after = torch.floor(torch.where(predict_dropout_mask,M,data))
    zero_num = iszero.sum()
    true_dropout_num = true_drop_out_mask.sum()
    predict_dropout_num = predict_dropout_mask.sum()
    print("predict_dropout_num:",predict_dropout_num,
          "\ntrue_dropout_num:", true_dropout_num,
          "\nzero_num:",zero_num,
          "\npredict out of true dropout rate:",
(predict_dropout_mask*true_drop_out_mask).sum()/true_dropout_num)

    dif_after =  truedata - after
    dif_true = truedata - data
    # print(dif_after)
    # print(dif_true)
    print("predict distance:", torch.sqrt(torch.square(truedata -
after).sum()).data,
          "origin distance:", torch.sqrt(torch.square(truedata -
data).sum()).data)
```

Analysis and Result Display

Split *test_data.csv,test_truedata.csv* into test set and validation set in a ratio of 7:3.

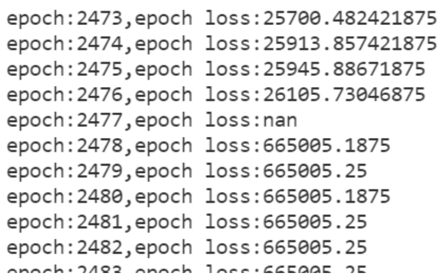
Test four models on test dataset

hasBN: whether has BatchNorm1d layer

zinb_new: whether use the improved zinb function

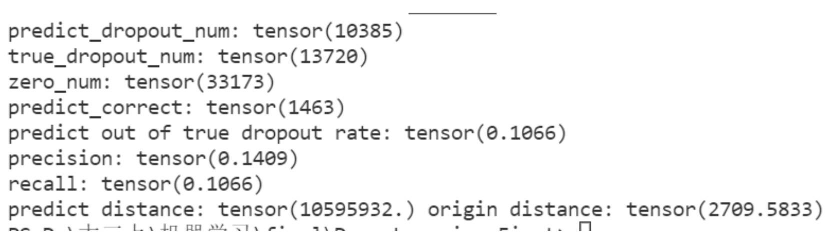
Model I：hasBN=False zinb_new=False

Gradient explosion occurred in the later stage of training.



```
epoch:2473,epoch loss:25700.482421875
epoch:2474,epoch loss:25913.857421875
epoch:2475,epoch loss:25945.88671875
epoch:2476,epoch loss:26105.73046875
epoch:2477,epoch loss:nan
epoch:2478,epoch loss:665005.1875
epoch:2479,epoch loss:665005.25
epoch:2480,epoch loss:665005.1875
epoch:2481,epoch loss:665005.25
epoch:2482,epoch loss:665005.25
```

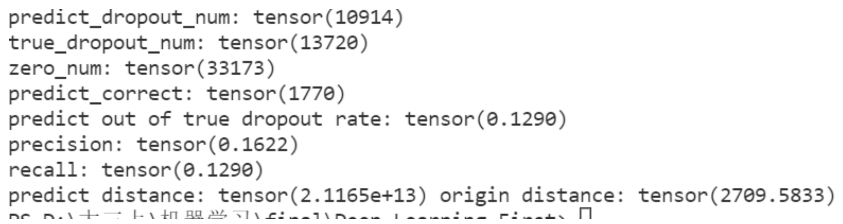
1000 epochs:



```
predict_dropout_num: tensor(10385)
true_dropout_num: tensor(13720)
zero_num: tensor(33173)
predict_correct: tensor(1463)
predict out of true dropout rate: tensor(0.1066)
precision: tensor(0.1409)
recall: tensor(0.1066)
predict distance: tensor(10595932.) origin distance: tensor(2709.5833)
```

Use Euclidean distance to represent the difference of predict value and real value of data. The difference is tremendous, and it is clearly underfitting.

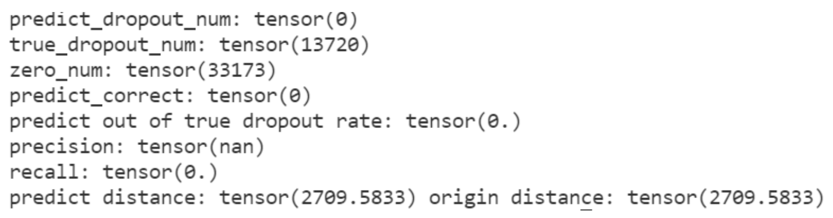
2000 epochs:



```
predict_dropout_num: tensor(10914)
true_dropout_num: tensor(13720)
zero_num: tensor(33173)
predict_correct: tensor(1770)
predict out of true dropout rate: tensor(0.1290)
precision: tensor(0.1622)
recall: tensor(0.1290)
predict distance: tensor(2.1165e+13) origin distance: tensor(2709.5833)
```

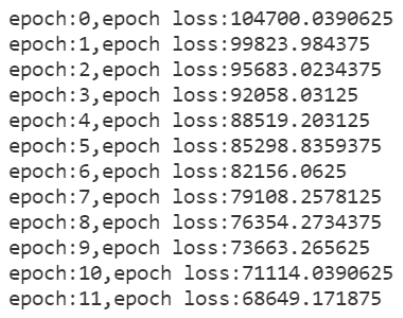
Still underfitting.

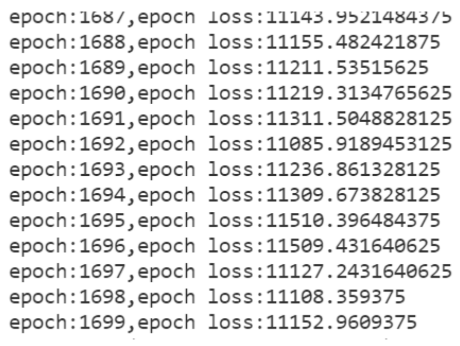
3000 epochs:


```
predict_dropout_num: tensor(0)
true_dropout_num: tensor(13720)
zero_num: tensor(33173)
predict_correct: tensor(0)
predict out of true dropout rate: tensor(0.)
precision: tensor(nan)
recall: tensor(0.)
predict distance: tensor(2709.5833) origin distance: tensor(2709.5833)
```

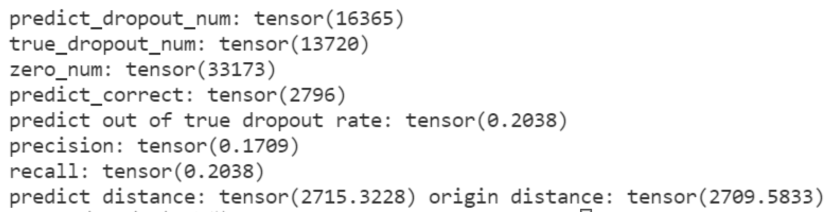
Gradient explosion.

Model II：hasBN=True zinb_new=False

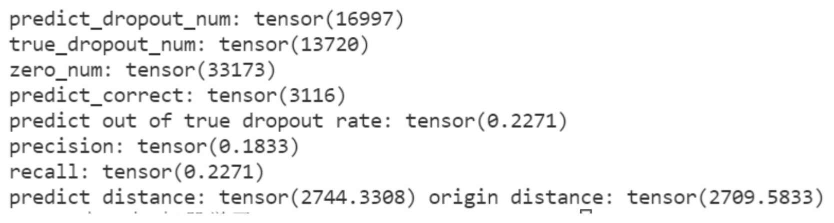

```
epoch:0,epoch loss:104700.0390625
epoch:1,epoch loss:99823.984375
epoch:2,epoch loss:95683.0234375
epoch:3,epoch loss:92058.03125
epoch:4,epoch loss:88519.203125
epoch:5,epoch loss:85298.8359375
epoch:6,epoch loss:82156.0625
epoch:7,epoch loss:79108.2578125
epoch:8,epoch loss:76354.2734375
epoch:9,epoch loss:73663.265625
epoch:10,epoch loss:71114.0390625
epoch:11,epoch loss:68649.171875
```


```
epoch:1687,epoch loss:11143.9521484375
epoch:1688,epoch loss:11155.482421875
epoch:1689,epoch loss:11211.53515625
epoch:1690,epoch loss:11219.3134765625
epoch:1691,epoch loss:11311.5048828125
epoch:1692,epoch loss:11085.9189453125
epoch:1693,epoch loss:11236.861328125
epoch:1694,epoch loss:11309.673828125
epoch:1695,epoch loss:11510.396484375
epoch:1696,epoch loss:11509.431640625
epoch:1697,epoch loss:11127.2431640625
epoch:1698,epoch loss:11108.359375
epoch:1699,epoch loss:11152.9609375
```

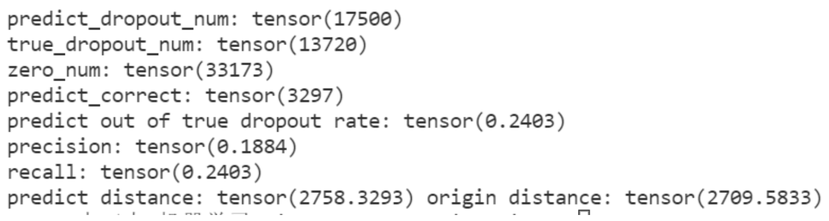
1000 epochs:


```
predict_dropout_num: tensor(16365)
true_dropout_num: tensor(13720)
zero_num: tensor(33173)
predict_correct: tensor(2796)
predict out of true dropout rate: tensor(0.2038)
precision: tensor(0.1709)
recall: tensor(0.2038)
predict distance: tensor(2715.3228) origin distance: tensor(2709.5833)
```

2000 epochs:


```
predict_dropout_num: tensor(16997)
true_dropout_num: tensor(13720)
zero_num: tensor(33173)
predict_correct: tensor(3116)
predict out of true dropout rate: tensor(0.2271)
precision: tensor(0.1833)
recall: tensor(0.2271)
predict distance: tensor(2744.3308) origin distance: tensor(2709.5833)
```

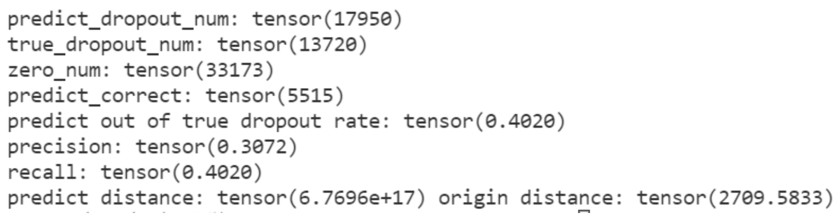
3000 epochs:



```
predict_dropout_num: tensor(17500)
true_dropout_num: tensor(13720)
zero_num: tensor(33173)
predict_correct: tensor(3297)
predict out of true dropout rate: tensor(0.2403)
precision: tensor(0.1884)
recall: tensor(0.2403)
predict distance: tensor(2758.3293) origin distance: tensor(2709.5833)
```

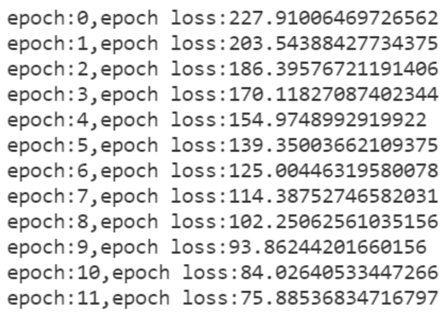
Model III：hasBN=False zinb_new=True

3000 epochs:

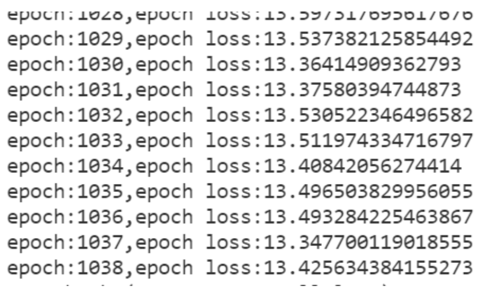


```
predict_dropout_num: tensor(17950)
true_dropout_num: tensor(13720)
zero_num: tensor(33173)
predict_correct: tensor(5515)
predict out of true dropout rate: tensor(0.4020)
precision: tensor(0.3072)
recall: tensor(0.4020)
predict distance: tensor(6.7696e+17) origin distance: tensor(2709.5833)
```

Model IV：hasBN=False zinb_new=True

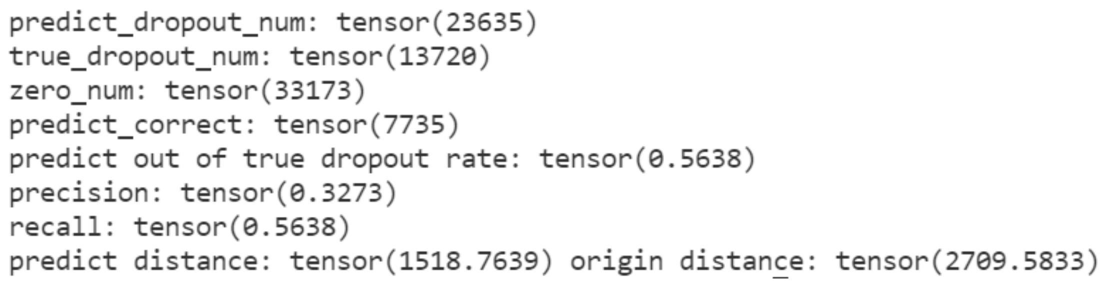


```
epoch:0,epoch loss:227.91006469726562
epoch:1,epoch loss:203.54388427734375
epoch:2,epoch loss:186.39576721191406
epoch:3,epoch loss:170.11827087402344
epoch:4,epoch loss:154.9748992919922
epoch:5,epoch loss:139.35003662109375
epoch:6,epoch loss:125.00446319580078
epoch:7,epoch loss:114.38752746582031
epoch:8,epoch loss:102.25062561035156
epoch:9,epoch loss:93.86244201660156
epoch:10,epoch loss:84.02640533447266
epoch:11,epoch loss:75.88536834716797
```



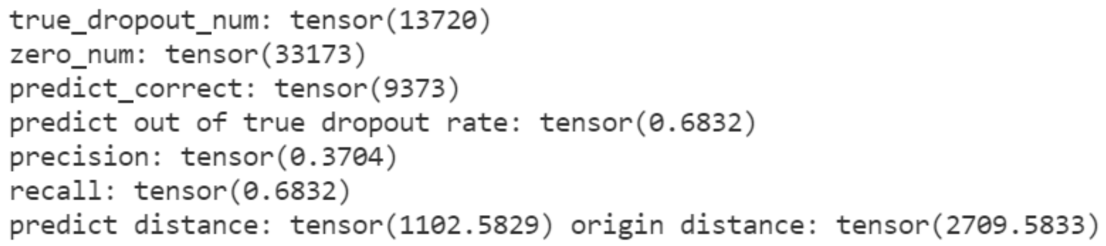
```
epoch:1028,epoch loss:13.597317695617676
epoch:1029,epoch loss:13.537382125854492
epoch:1030,epoch loss:13.36414909362793
epoch:1031,epoch loss:13.37580394744873
epoch:1032,epoch loss:13.530522346496582
epoch:1033,epoch loss:13.511974334716797
epoch:1034,epoch loss:13.40842056274414
epoch:1035,epoch loss:13.496503829956055
epoch:1036,epoch loss:13.493284225463867
epoch:1037,epoch loss:13.347700119018555
epoch:1038,epoch loss:13.425634384155273
```

1000 epochs:



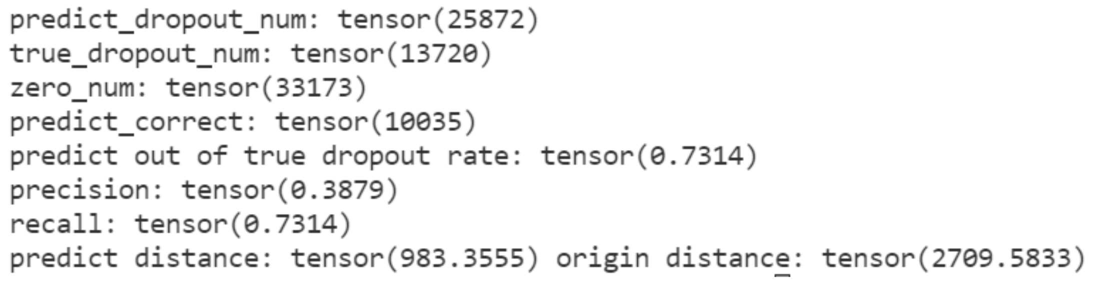
```
predict_dropout_num: tensor(23635)
true_dropout_num: tensor(13720)
zero_num: tensor(33173)
predict_correct: tensor(7735)
predict out of true dropout rate: tensor(0.5638)
precision: tensor(0.3273)
recall: tensor(0.5638)
predict distance: tensor(1518.7639) origin distance: tensor(2709.5833)
```

2000 epochs:



```
true_dropout_num: tensor(13720)
zero_num: tensor(33173)
predict_correct: tensor(9373)
predict out of true dropout rate: tensor(0.6832)
precision: tensor(0.3704)
recall: tensor(0.6832)
predict distance: tensor(1102.5829) origin distance: tensor(2709.5833)
```

3000 epochs:



```
predict_dropout_num: tensor(25872)
true_dropout_num: tensor(13720)
zero_num: tensor(33173)
predict_correct: tensor(10035)
predict out of true dropout rate: tensor(0.7314)
precision: tensor(0.3879)
recall: tensor(0.7314)
predict distance: tensor(983.3555) origin distance: tensor(2709.5833)
```

Conclusion: Model IV has the best performance. The new zinb loss function is better, and using batchNorm1d layer is better than not using it.

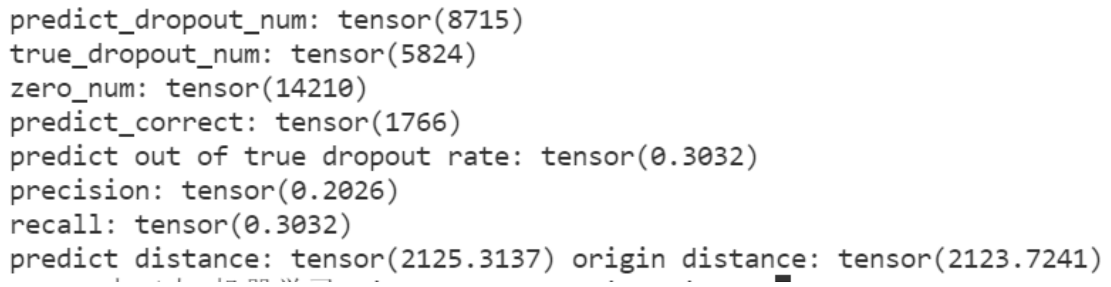
Validate on the validation set

Predict on the validation set using 3000 epoch model.

Model I:

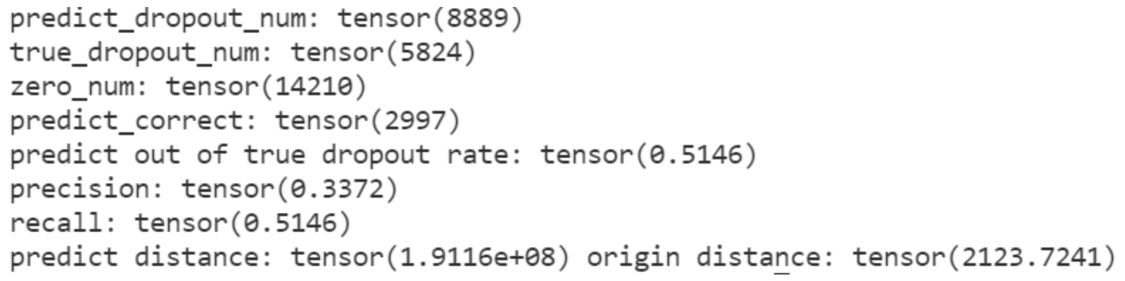
Model I has poor performance, and gradient explode after 3000 epoches, so it is not comparable. The outcome is not listed here.

Model II:



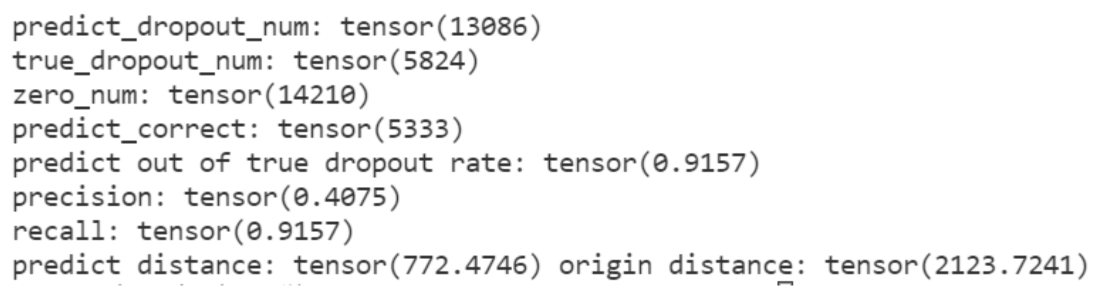
```
predict_dropout_num: tensor(8715)
true_dropout_num: tensor(5824)
zero_num: tensor(14210)
predict_correct: tensor(1766)
predict out of true dropout rate: tensor(0.3032)
precision: tensor(0.2026)
recall: tensor(0.3032)
predict distance: tensor(2125.3137) origin distance: tensor(2123.7241)
```

Model III:



```
predict_dropout_num: tensor(8889)
true_dropout_num: tensor(5824)
zero_num: tensor(14210)
predict_correct: tensor(2997)
predict out of true dropout rate: tensor(0.5146)
precision: tensor(0.3372)
recall: tensor(0.5146)
predict distance: tensor(1.9116e+08) origin distance: tensor(2123.7241)
```

Model IV:



```
predict_dropout_num: tensor(13086)
true_dropout_num: tensor(5824)
zero_num: tensor(14210)
predict_correct: tensor(5333)
predict out of true dropout rate: tensor(0.9157)
precision: tensor(0.4075)
recall: tensor(0.9157)
predict distance: tensor(772.4746) origin distance: tensor(2123.7241)
```

It can be seen that model IV has the best performance, with high precision and recall, and the lowest predict distance.

Analyzation

Model I is the original model reproduced as described in the paper. However, the performance of Model I is undesirable, and the speed of its gradient descent is slow. Besides, after certain epoches, the loss is NA, which represents calculation error or numerical anormaly. This can be caused by gradient explosion, which makes the parameters to be extreme and leads to overflow, making the value be inf, and result in NaN eventually. Thus the batch normalization layer is introduced into the neural network, that is, set `autoencoder = AutoEncoder(1000,hasBN=True)` , and test its performance. It can be seen that the denoise of data.csv has a fair performance compared with the simulated real dataset that generated data.csv, with less dropout and less difference.

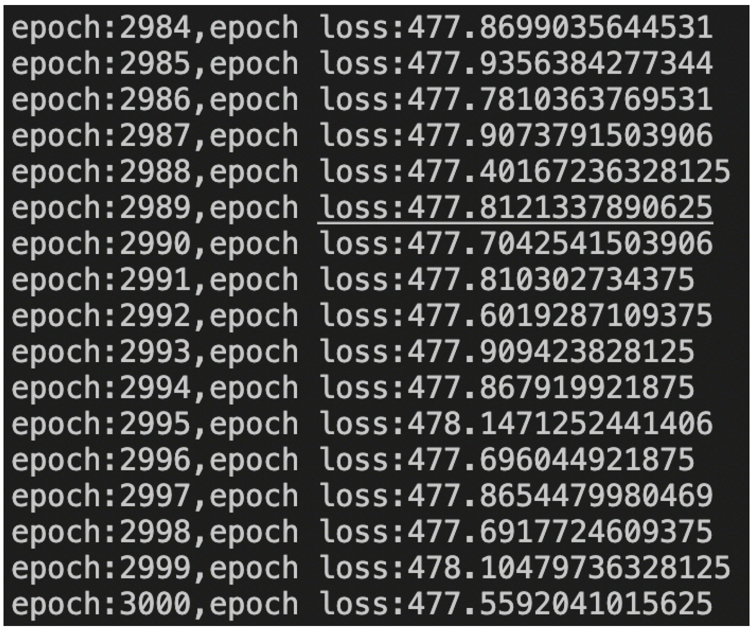
After reading the whole paper, it is found that some experiment result cannot be reproduced according to the method described in the paper. Besides, the source code is different with the text description. Thus, re-implement loss calculation and training technique as below:

```
class LZINBLoss(nn.Module):
    def __init__(self, eps=1e-6):
        super().__init__()
        self.eps = eps

    def forward(self, X: Tensor, PI: Tensor = None, M: Tensor = None, THETA:
Tensor = None):
        # 防止出现除0, log(0) log (负数) 等等等
        eps = self.eps
        # deal with inf
        max1 = max(THETA.max(), M.max())
        if THETA.isinf().sum() != 0:
            THETA = torch.where(THETA.isinf(), torch.full_like(THETA, max1),
THETA)
        if M.isinf().sum() != 0:
            M = torch.where(M.isinf(), torch.full_like(M, max1), M)

        if PI.isnan().sum() != 0:
            PI = torch.where(PI.isnan(), torch.full_like(PI, eps), PI)
        if THETA.isnan().sum() != 0:
            THETA = torch.where(THETA.isnan(), torch.full_like(THETA, eps),
THETA)
        if M.isnan().sum() != 0:
            M = torch.where(M.isnan(), torch.full_like(M, eps), M)

        eps = torch.tensor(1e-10)
        THETA = torch.minimum(THETA, torch.tensor(1e6))
        t1 = torch.lgamma(THETA + eps) + torch.lgamma(X + 1.0) -
torch.lgamma(X + THETA + eps)
        t2 = (THETA + X) * torch.log(1.0 + (M / (THETA + eps))) + (X *
(torch.log(THETA + eps) - torch.log(M + eps)))
        nb = t1 + t2
        nb = torch.where(torch.isnan(nb), torch.zeros_like(nb) + max1, nb)
        nb_case = nb - torch.log(1.0 - PI + eps)
        zero_nb = torch.pow(THETA / (THETA + M + eps), THETA)
        zero_case = -torch.log(PI + ((1.0 - PI) * zero_nb) + eps)
        res = torch.where(torch.less(X, 1e-8), zero_case, nb_case)
        res = torch.where(torch.isnan(res), torch.zeros_like(res) + max1,
res)
        return torch.mean(res)
```

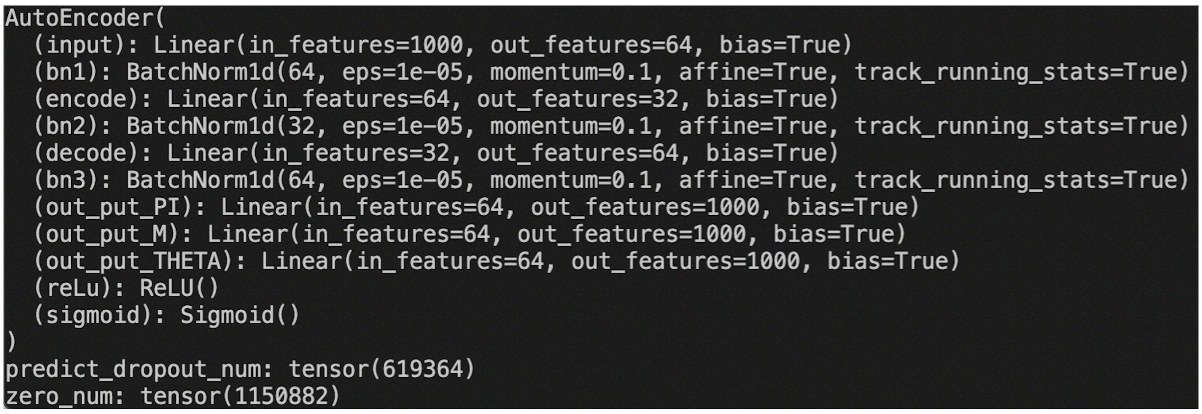
```
def train(EPOCH_NUM=100, print_batchloss=False, autoencoder=None,
loader=None, startEpoch=0):
    opt = Adam(autoencoder.parameters(), lr=LR, betas=(BETA1, BETA2),
eps=EPS, weight_decay=WEIGHT_DECAY)
    # opt = SGD(autoencoder.parameters(), lr=1e-2, momentum=0.8)
    mean_loss=0
    for epoch in range(EPOCH_NUM+1):
        epoch_loss = 0

        for batch, batch_data in enumerate(loader):
            opt.zero_grad()
            train_batch = batch_data[0]
            d = train_batch
            PI, M, THETA = autoencoder(d)
            templ = lzinbloss(d, PI, M, THETA)
            epoch_loss += templ
            if print_batchloss:
                print(f'epoch:{epoch+startEpoch},batch:{batch},batch loss:
{templ},(batch size {BATCHSIZE})')
```

```
                f.write(f'epoch:{epoch+startEpoch},batch:{batch},batch loss:
{templ},(batch size {BATCHSIZE})\n')
            # backpropagtion
            templ.backward()
            clip_grad_norm_(autoencoder.parameters(), max_norm=5,
norm_type=2)
            # gra descent
            opt.step()
        print(f'epoch:{epoch+startEpoch},epoch loss:{epoch_loss}')
        f.write(f'epoch:{epoch+startEpoch},epoch loss:{mean_loss}\n')
        # mean_loss=0
        if epoch % EVER_SAVING == 0 and epoch!=0:
torch.save(autoencoder.state_dict(), open(f'0113epoch{epoch+startEpoch}.pkl',
'wb'))
        # if epoch % EVER_SAVING == 0 and epoch!= 0 :
torch.save(autoencoder.state_dict(),
open(f'0113epoch{epoch+startEpoch}withoutBN.pkl', 'wb'))
```

After the improvement, test on the test set, the result is shown as below, and the denoised data is saved in result.csv.



```
epoch:2984,epoch loss:477.8699035644531
epoch:2985,epoch loss:477.9356384277344
epoch:2986,epoch loss:477.7810363769531
epoch:2987,epoch loss:477.9073791503906
epoch:2988,epoch loss:477.40167236328125
epoch:2989,epoch loss:477.8121337890625
epoch:2990,epoch loss:477.7042541503906
epoch:2991,epoch loss:477.810302734375
epoch:2992,epoch loss:477.6019287109375
epoch:2993,epoch loss:477.909423828125
epoch:2994,epoch loss:477.867919921875
epoch:2995,epoch loss:478.1471252441406
epoch:2996,epoch loss:477.696044921875
epoch:2997,epoch loss:477.8654479980469
epoch:2998,epoch loss:477.6917724609375
epoch:2999,epoch loss:478.10479736328125
epoch:3000,epoch loss:477.5592041015625
```



```
AutoEncoder(
  (input): Linear(in_features=1000, out_features=64, bias=True)
  (bn1): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (encode): Linear(in_features=64, out_features=32, bias=True)
  (bn2): BatchNorm1d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (decode): Linear(in_features=32, out_features=64, bias=True)
  (bn3): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (out_put_PI): Linear(in_features=64, out_features=1000, bias=True)
  (out_put_M): Linear(in_features=64, out_features=1000, bias=True)
  (out_put_THETA): Linear(in_features=64, out_features=1000, bias=True)
  (reLu): ReLU()
  (sigmoid): Sigmoid()
)
predict_dropout_num: tensor(619364)
zero_num: tensor(1150882)
```