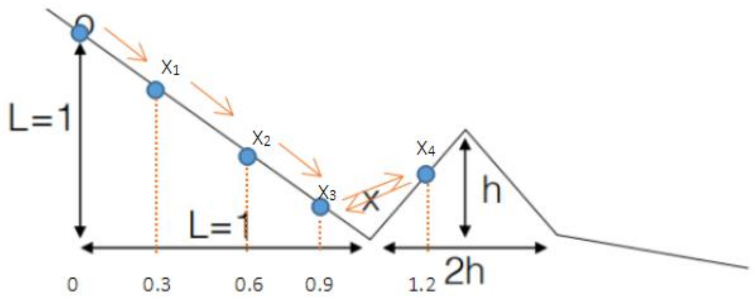
BS 6207 ASSIGNMENT 2 - *Tian Siqi, G2101015G*

Github link: https://github.com/SiqiT/Assigmnment

Question 1

(1) Standard gradient descend: $x^{t} = x^{t-1} - a\nabla f(x^{t-1})$



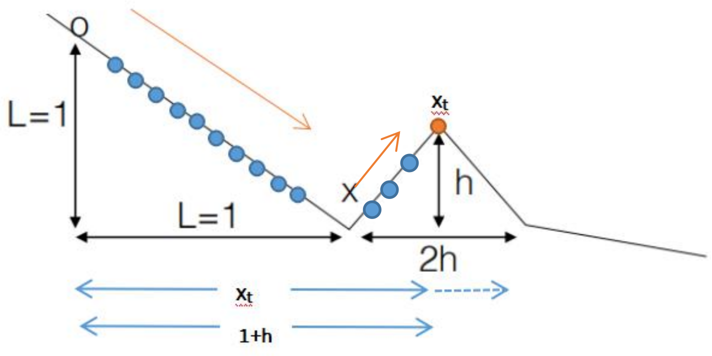
$\nabla f(x)$ for the first line is -1, learning rate a = 0.3

Start from point 0, $x_1 = x_0 + 0.3*1$, x1 is the point move right 0.3 from point 0;

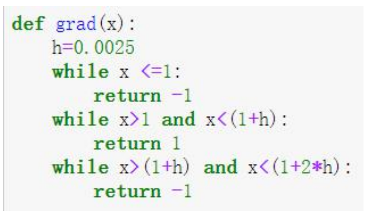
The same, we can get x_2,x_3;

About x_4, **since h>a=0.3, x_4 is on the second line,** $\nabla f(x)$ is 1, $x_5 = x_4 - 0.3*1$, x_5 is is the point

move left 0.3 from x_4;(move to the direction that the gradient is descending),
Since x_5 is the same point as x_3, then x would **vibrate** between point x_3 and x_4, The gradient descends can't pass the 'h hill', it will be around the local minimum forever.
(2)In order to escape the local minimum, we need to find **the last point x(the highest point)**when the x is **climbing** the 'h hill' during the gradient descend search, and let $x_t \geq 1 + h$.

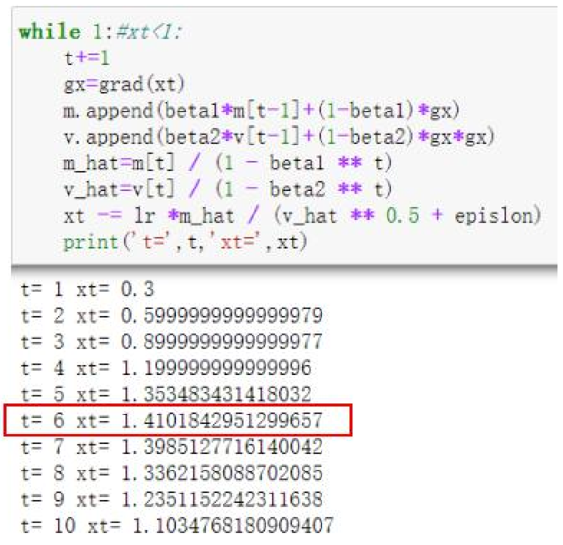


I write a script to find the x_t, and then use x_t to find the max h. There are 2 main parts in this script. The first part is about the $\nabla f(x)$ for each line.



```
def grad(x):
    h=0.0025
    while x <=1:
        return -1
    while x>1 and x<(1+h):
        return 1
    while x>(1+h) and x<(1+2*h):
        return -1
```

The next part is to find the **last point x**(the highest point) x_t.
(set the original parameters: h=0.5, lr=0.3, beta1=0.9, beta2=0.999, epislon=0, m=[0], v = [0], t = 0, xt=0)



```
while 1:#xt<1:
    t+=1
    gx=grad(xt)
    m.append(beta1*m[t-1]+(1-beta1)*gx)
    v.append(beta2*v[t-1]+(1-beta2)*gx*gx)
    m_hat=m[t] / (1 - beta1 ** t)
    v_hat=v[t] / (1 - beta2 ** t)
    xt -= lr *m_hat / (v_hat ** 0.5 + epislon)
    print('t=',t,'xt=',xt)
```

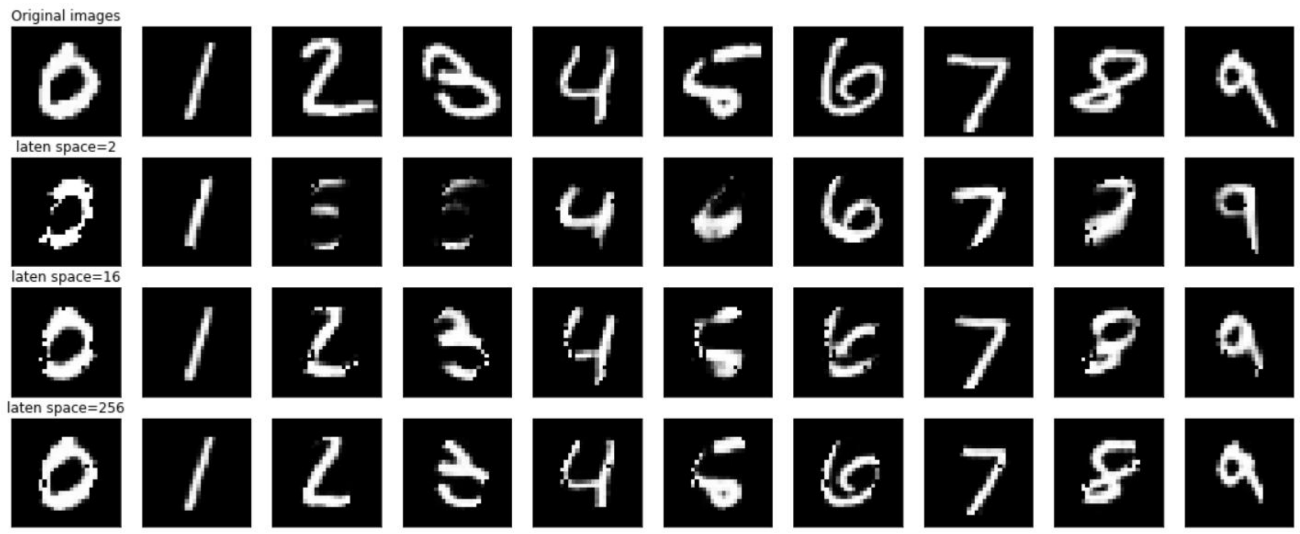
```
t= 1 xt= 0.3
t= 2 xt= 0.5999999999999979
t= 3 xt= 0.8999999999999977
t= 4 xt= 1.199999999999996
t= 5 xt= 1.353483431418032
t= 6 xt= 1.4101842951299657
t= 7 xt= 1.3985127716140042
t= 8 xt= 1.3362158088702085
t= 9 xt= 1.2351152242311638
t= 10 xt= 1.1034768180909407
```

After 6 times iteration, we get x_t≈1.41, $x_t \geq 1 + h$, thus the max **h = 0.41**

Now the gradient descend can escape from the 'h hill' local minimum problem.

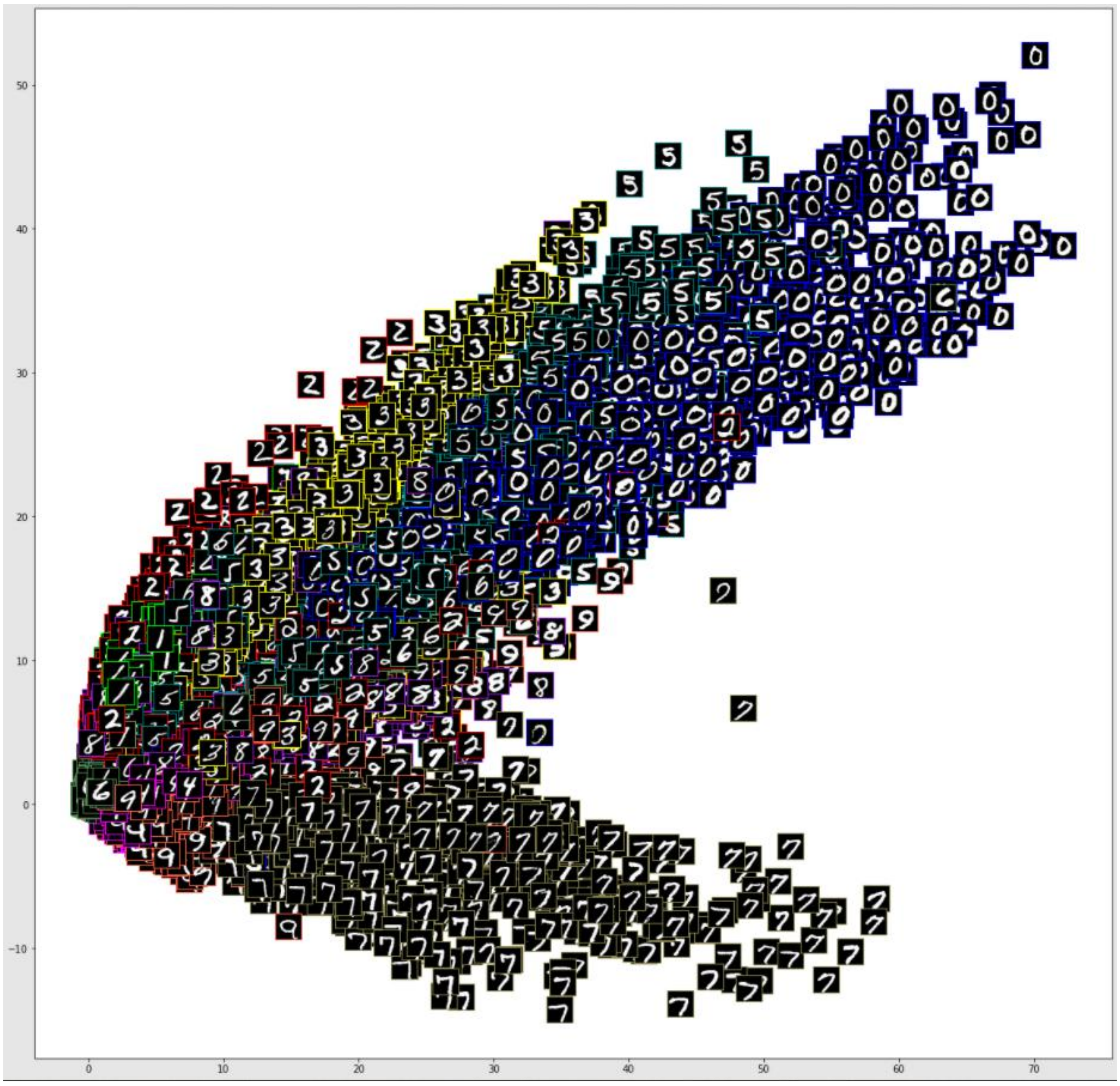
Question 2

(a) In this section, I first design 3 auto encoders all using dense layer, each has a latent space dimension of 2, 16, 256, each auto encoder is combined from 2 separate models, one is the **encoder**, one is the **decoder** to reconstruct the image, in this way I can check the latent space situation.
The loss function when compile the model is '*mae*', the L1 loss function.
(1) Reconstructed images for different latent spaces:



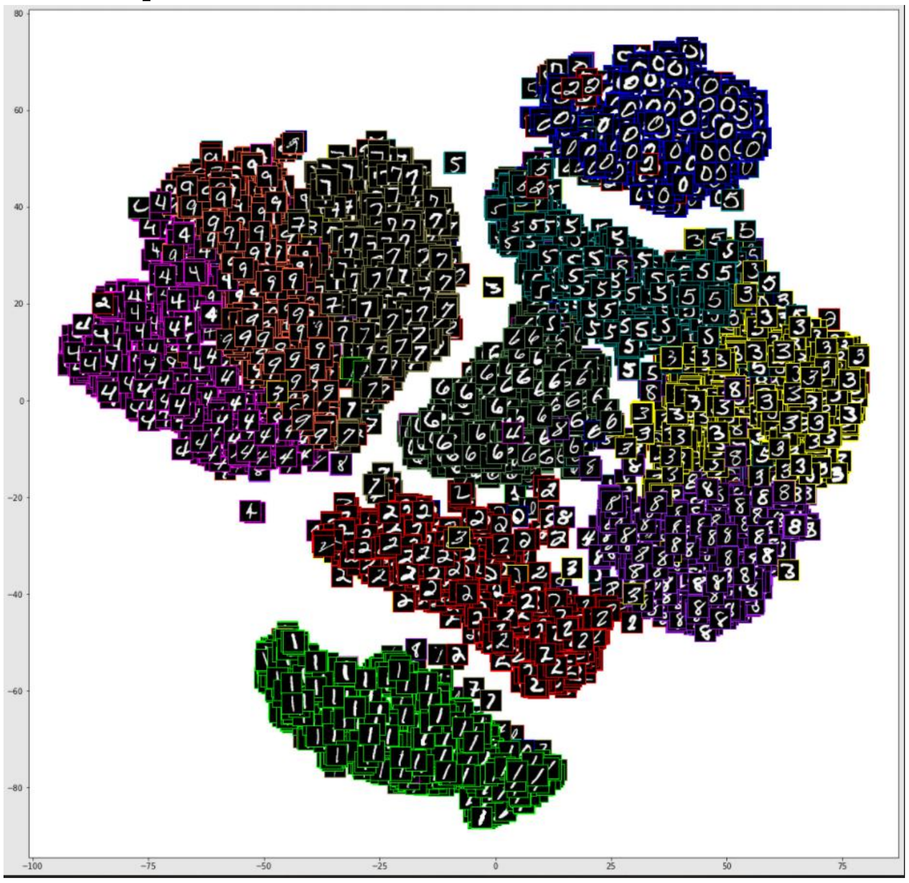
A **higher** dimension of latent space would catch more **details** from the original images. For a **simple** image, such as digit '0''1''7, the dimension of 2,16,256 would reconstruct the image with no big difference, however, when the image becomes more **complex**, such like '2''3''5''6''9', a higher dimension(16, 256) would construct more details, so the output would be more like the original image, the low latent dimension(2) would miss some important features and reconstruct the image to the wrong digit.

(2)2D plot of the latent space for different digits for latent space of 2D

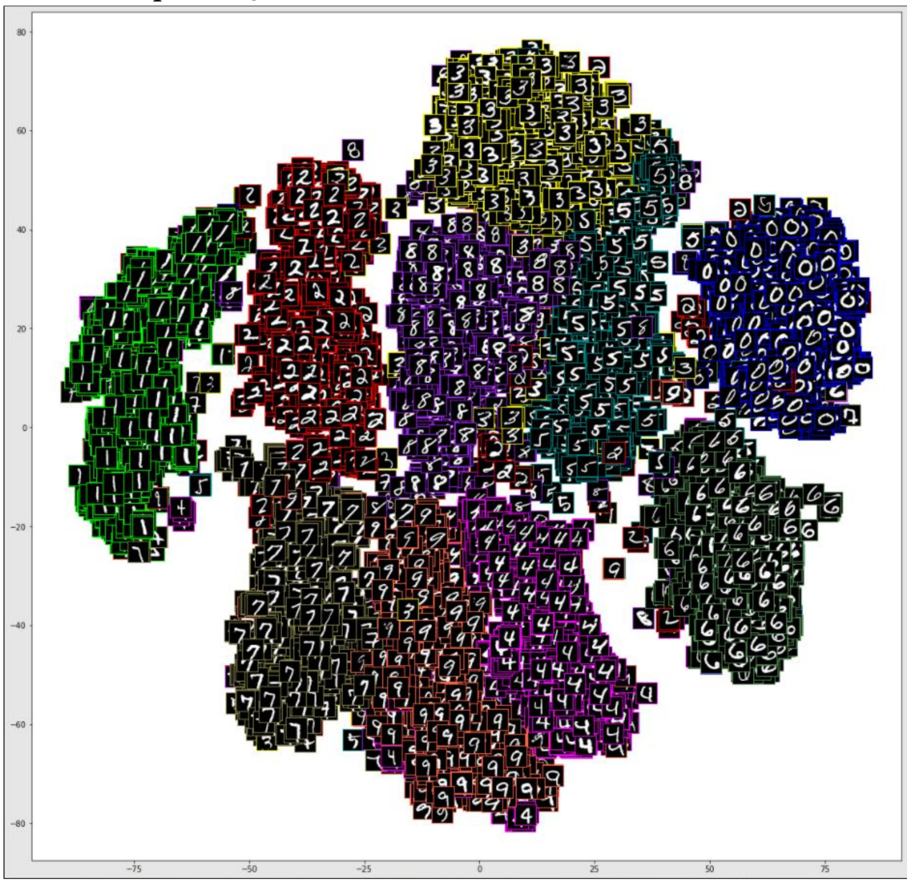


From this 2D latent space, we can see the same digits are close to each other, also the similar digits are likely to be **close** with each other too.
(3)K-Mean clustering visualization
Here I m using the images from the latent space with the dimension of 16 and 256 to do the clustering and then use **t-SNE** to visualize it in a 2d plot.
The K-mean model would cluster all images in 10 clustering, then the colorful border of each original image represent 10 different digit(True label).

(i)Latent Space=16

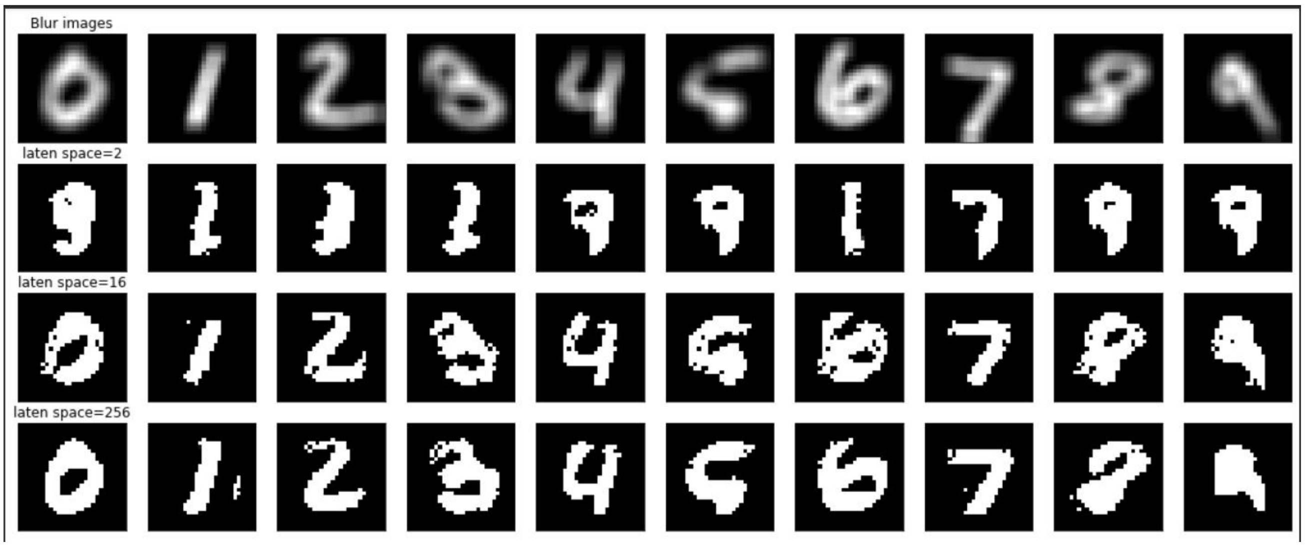


(2) Latent Space=256



In all, the k mean with latent space dimensions 16 and 256 can overall cluster 10 digits to different clusters. The result is not perfect, but compared to the original dataset, the result for each cluster is **denser**.
Dimension of 16 and 256 can both cluster '1' and '0' good, but for the dimension of 16, '4''9''7' and '3''8''5''6''2' are not very distinguished; For the dimension of 256,'2''6' can be clustered more separately with other digits, '4''9''7' and '3''8''5' clusters are still close to each other.

(b)Reconstructed images for different latent spaces for blur images:



Here I use the blurred image as input and use the clear image as the label to train this encoder, using L1 loss as the loss function.
From this result, we can see a **higher** latent space can get a more **clear** reconstructed image. Compared with the reconstructed image from the clear image, those new reconstructed images from blurred images have **thicker** lines, the reconstructed image is **coarsening**. In this case, if the accuracy is good enough, a lower latent space seems can give a reconstructed image with clearer edges.
In the end, I use the dis_net to check whether my reconstructed images are clear, the prediction result says they are. (But I don't think that's very accurate..)

Some failed approaches:
(1) Design a **binary** classification network to classify blur images and clear images, there are **labels 0,1** for training this network. However, I can't find a good way to **define the dis_loss** from this network. If using the **prediction** result as loss, for all the blur images, the loss would be **1**, that's meaningless.
Next step I would try to build a **CNN** network, and use the **probability** for the prediction of a blur as the loss value.
(2)Define a **customer loss function** that get
loss=L1_Loss(pred_image,clear_image) +L1_Loss(pred_image,blur_image),
However since I build the network using TensorFlow, it doesn't allow me to **interpret with the KerasTensor input**(blur_image). I need to find a way to solve this problem.