**Part1. Interesting discoveries in step1.1 and step1.2:**

In step1, it is required to have an access to GPU. While I don’t know how to get an access to GPU, I tried to directly run the program as a python project on my own computer. I guess I am running it with my CPU instead.

Google COLAB is about 10 times faster than my own laptop when training the model.

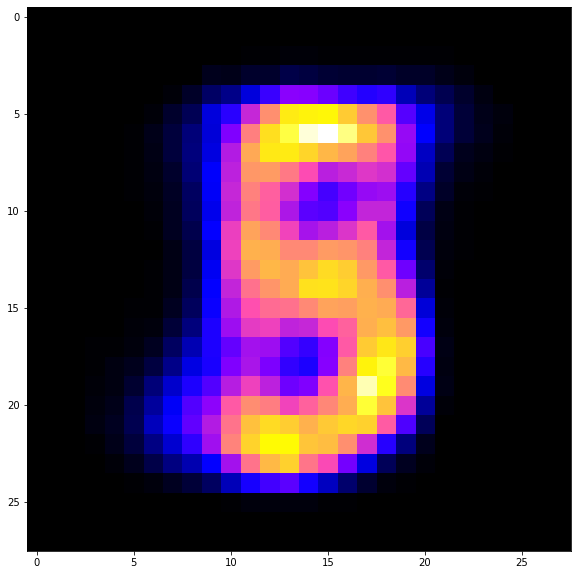
**Part2. Step1.3.1**

In this part, I basically just select some images and find their accordingly latent variable in latent space (z), decode them back to a matrix, then plot them out after fitting data set to the model. The images are listed as follows:

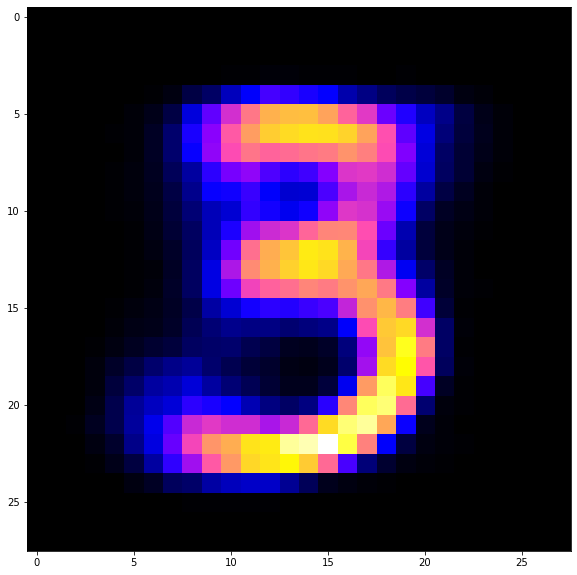
Original image:



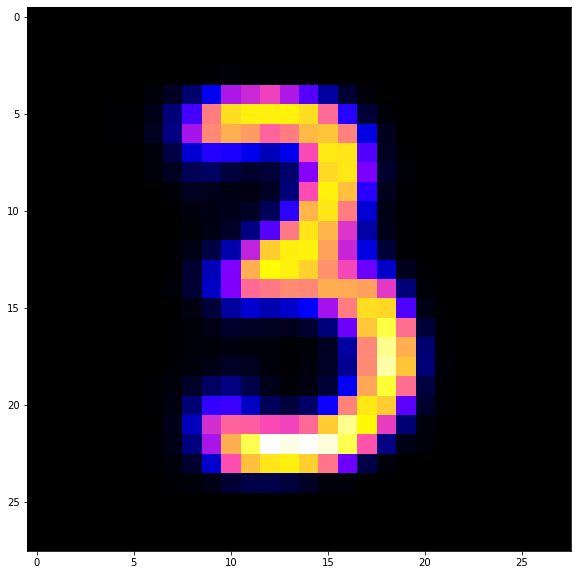
K = 1:



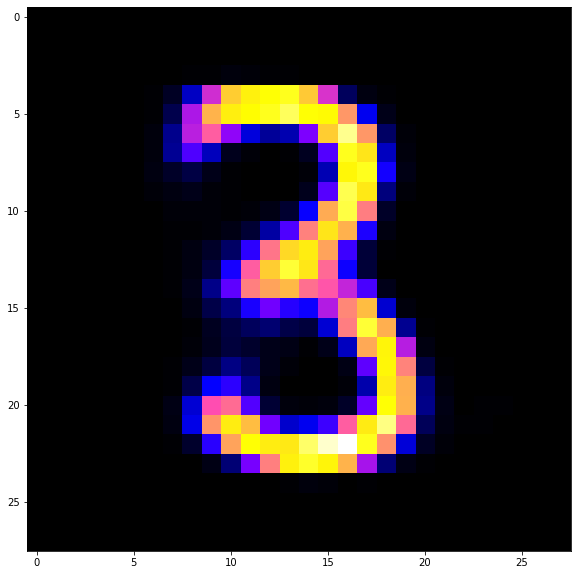
K = 2:



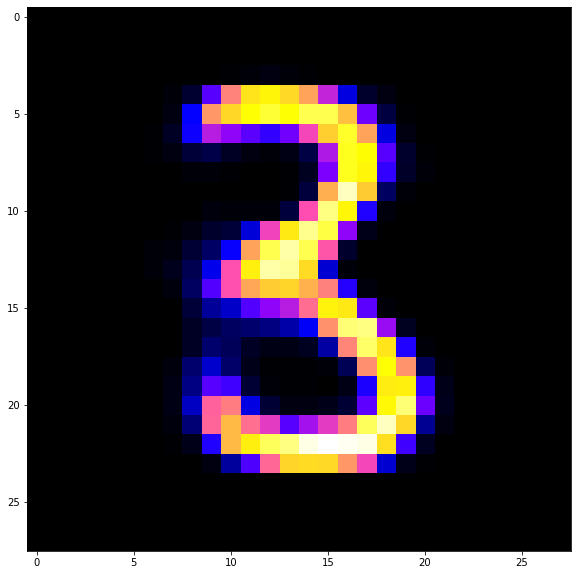
K = 4:



K = 6:



K=8:



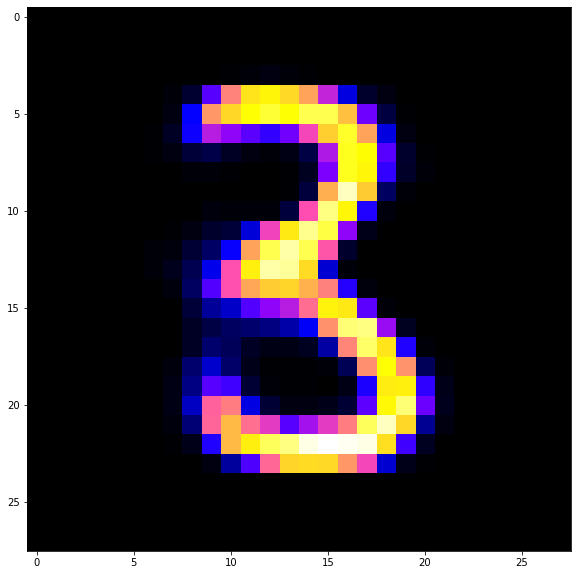
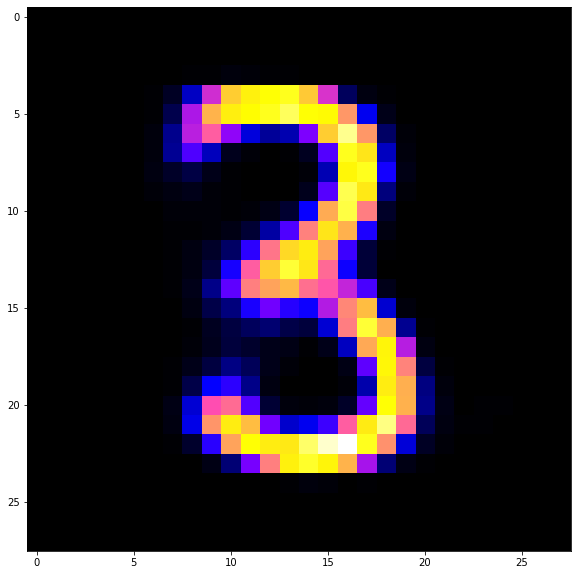
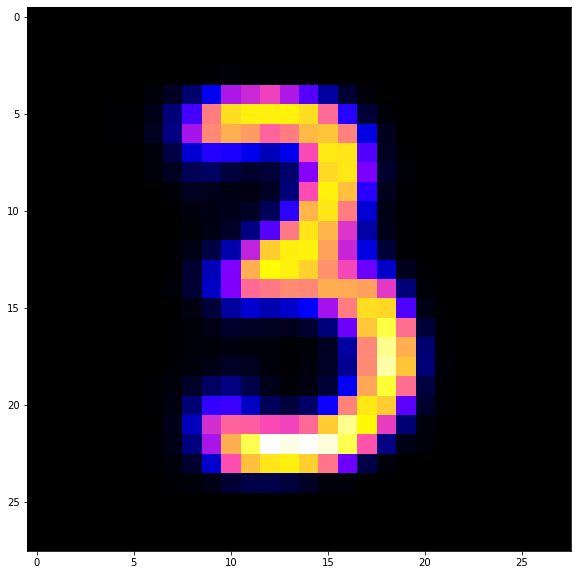
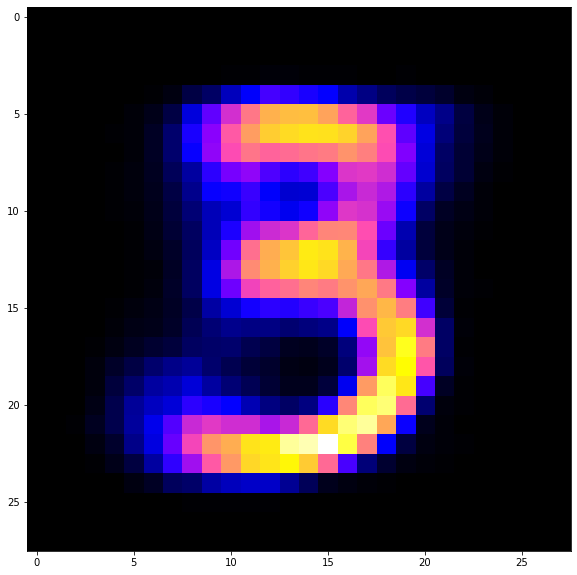
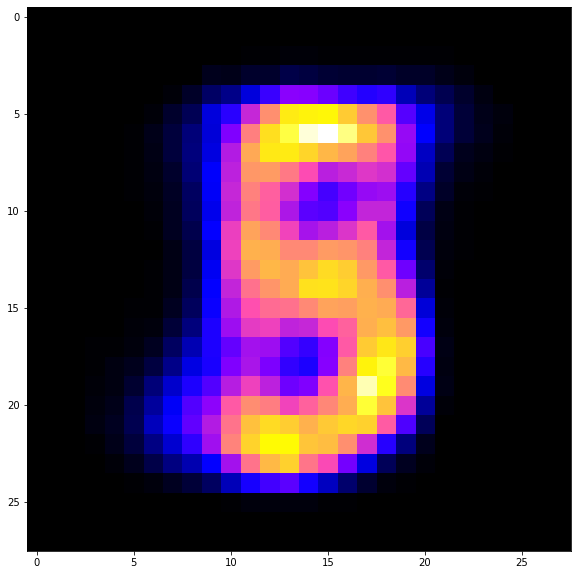
K = 10:



We could list them together as k changes to see what the qualities of images like:



The origin



K = 1 2 4 6 8 10

**Part3. Step1.3.2, similarity function:**

In this section, we are asked to tell the similarity of 2 images, which are represented as matrix with 28x28 pixels. I searched the Internet and found three different measures to tell the similarity of 2 matrices. They all have their own pros and cons. So I decided to combine them all and weight them by observation.

Here are all these functions and how I combine them:

The first function:

def mtx\_similar1(arr1:np.ndarray, arr2:np.ndarray) ->float:

farr1 = arr1.ravel()

farr2 = arr2.ravel()

len1 = len(farr1)

len2 = len(farr2)

if len1 > len2:

farr1 = farr1[:len2]

else:

farr2 = farr2[:len1]

numer = np.sum(farr1 \* farr2)

denom = np.sqrt(np.sum(farr1\*\*2) \* np.sum(farr2\*\*2))

similar = numer / denom

return (similar+1) / 2

This function is using a method to transfer the matrix into vectors, and then use their dot product dividing their length, which is actually the cosine value of the angle between these 2 vectors.

The second function:

def mtx\_similar2(arr1:np.ndarray, arr2:np.ndarray) ->float:

if arr1.shape != arr2.shape:

minx = min(arr1.shape[0],arr2.shape[0])

miny = min(arr1.shape[1],arr2.shape[1])

differ = arr1[:minx,:miny] - arr2[:minx,:miny]

else:

differ = arr1 - arr2

numera = np.sum(differ\*\*2)

denom = np.sum(arr1\*\*2)

similar = 1 - (numera / denom)

if similar <0:

return 0

else:

return similar

This function is calculating the square summary of each element’s difference at the same position. Since it should be 0 or similar to 0 if 2 images are considered as “similar”.

The third function:

def mtx\_similar3(arr1:np.ndarray, arr2:np.ndarray) ->float:

if arr1.shape != arr2.shape:

minx = min(arr1.shape[0],arr2.shape[0])

miny = min(arr1.shape[1],arr2.shape[1])

differ = arr1[:minx,:miny] - arr2[:minx,:miny]

else:

differ = arr1 - arr2

dist = np.linalg.norm(differ, ord='fro')

len1 = np.linalg.norm(arr1)

len2 = np.linalg.norm(arr2)

denom = (len1 + len2) / 2

similar = 1 - (dist / denom)

if similar <0:

return 0

else:

return similar

This function is based on Frobenius norm which is the square root of the squared sum of differences of all elements. In other words, reshape the matrices into vectors and compute the Euclidean distance between them.

All these 3 functions have their own pros and cons. I have tested them with real images to see their performances. Function 1 does well when 2 images are visually similar to human being for example, me. While function 2 and 3 are too pathetic while evaluating “good” situations. So I manually made some management with combining all 3 functions together with some weight:

def mtx\_similar(arr1:np.ndarray, arr2:np.ndarray) ->float:

similar1 = mtx\_similar1(arr1,arr2)

similar2 = mtx\_similar2(arr1,arr2)

similar3 = mtx\_similar3(arr1,arr2)

if similar1 > 0.9:

similar = 7/9\*similar1 + 2/9\*similar2

if similar1 < 0.8:

similar = 1/6\*similar1 + 2.5/6\*similar2 + 2.5/6\*similar3

if similar1 >=0.8 and similar1 <=0.9:

similar = 4/9\*similar1 + 1/3\*similar2 + 2/9\*similar3

return similar

One thing shall be explained is that this measurement above is never good enough to be the final similarity function since lots of work shall be done with it. It is somehow satisfying but it’s just a very rough manually made function.

**Part4. Step1.3.3,**

First of all, we calculate the similarity of 100 samples randomly picked from training data set and tracked them down as previously explained. Here is the table of the stats I have collected of the average similarity under certain dimensionality.

K = 1, average similarity = 0.564489821709614, loss: 0.2135 - val\_loss: 0.2141

K = 2, average similarity = 0.6870530963796431, loss: 0.1816 - val\_loss: 0.1825

K = 4, average similarity = 0.8445063544534027, loss: 0.1446 - val\_loss: 0.1474

K = 6, average similarity = 0.9172682344910005, loss: 0.1225 - val\_loss: 0.1234

K = 8, average similarity = 0.9401797929854635, loss: 0.1102 - val\_loss: 0.1106

-0.9487098134247839 loss: 0.1004 - val\_loss: 0.1025

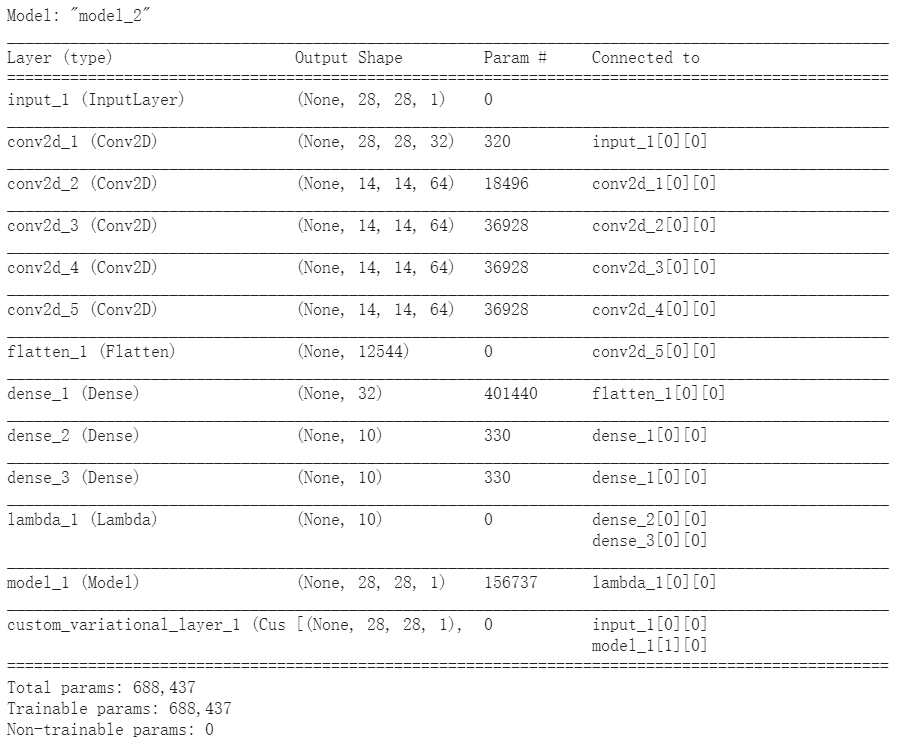
K = 10, average similarity = 0.9412496733839781, loss: 0.1101 - val\_loss: 0.1110

+0.9436439850288749 loss: 0.1035 - val\_loss: 0.1060

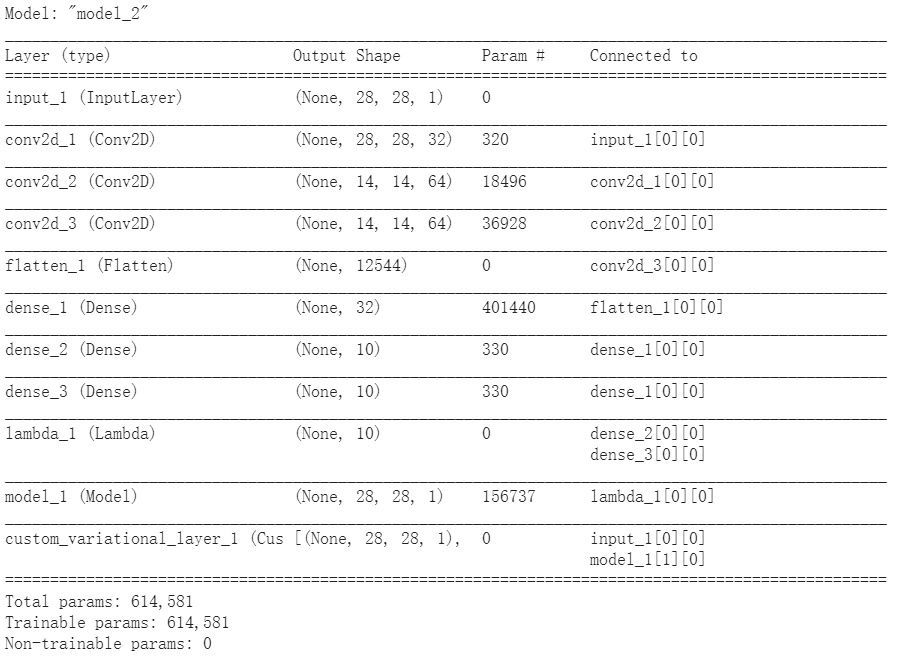
Here is the plot of similarity as k changes:

**Part5. Step1.4**

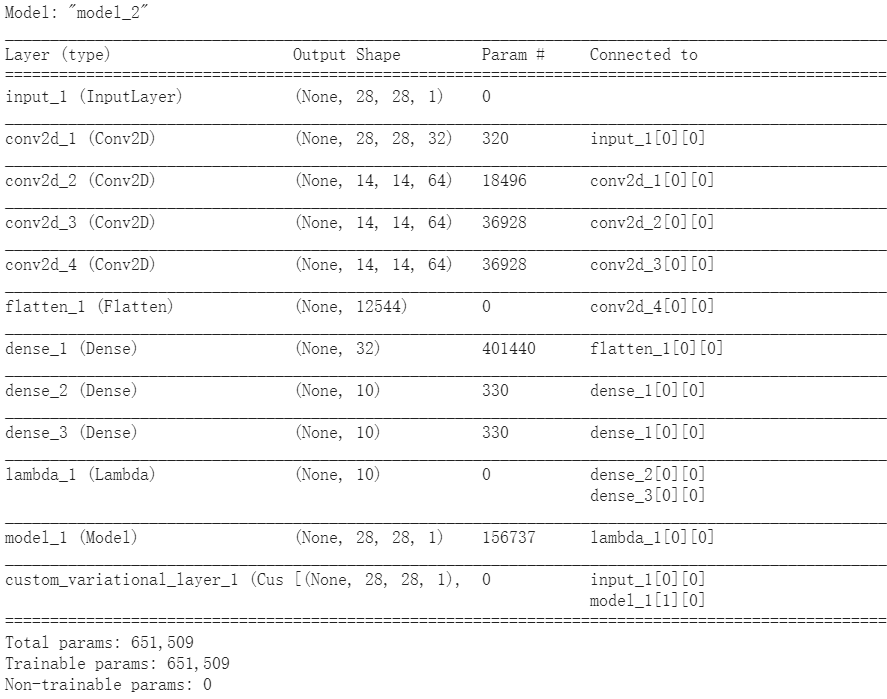
5 layer: 0.9200825508241895



3 layer: 0.9526497399725763



Normal: 0.9371218278168989



**Part6. What I have learnt**