Minor Project "Variational Auto-Encoder for Data Privacy"

Ritesh Meena
Ritik Bansal
Rahul Singh
Charan Singh Fagna

July 17, 2020

1) INTRODUCTION

1.1 MOTIVATION

Increasingly more data is collected about almost every aspect of human life ranging from health care delivery to social media. As the amount of collected data increases, more opportunities have emerged for leveraging this collected data for important Societal purposes. Since the collected data can be used to for important services and facilitate much needed research, many organizations are striving to share the data that they collect. At the same time, sharing micro data carries inherent risks to individual privacy.

To address this privacy challenge, solutions have been proposed in two broad categories. In the First category, the data anonymization based approaches which try to use various definitions to sanitize data so that it cannot be easily residentified. Although these approaches have some important use cases, they are not usually based on rigorous privacy definitions that can withstand various types of re-identification attacks.

In the second category, synthetic data generation approaches have been proposed to generate realistic synthetic data using rigorous differential privacy definition that we tried to implement and improve in our project..

1.2 AIM

We propose an auto-encoder Model, a generative deep learning technique that generates privacy preserving synthetic data. We test our approach on MNIST dataset and compare the results for data without Noise and data after applying DP Noise in the weights.

2) LITERATURE REVIEW

2.1 Differential Privacy

Differential privacy is the formal mathematical model that ensures privacy protection, and it is primarily used to analyze and release sensitive data statistics. Differential privacy utilizes randomized algorithms to sanitize sensitive information while bounding the privacy risk of revealing sensitive information.

Definition 1 (Sensitivity [11]). *For a given function f, the sensitivity of f is defined as a maximum absolute distance between two neighboring pairs (d,d')*

$$s_f = \max_{(d,d')} \|f(d) - f(d')\|, \tag{2}$$

where $\|.\|$ is L_1 norm.

The (ε, δ)-*differential privacy* of function f over data d is guaranteed by $\mathcal{F}(d)$ with the Gaussian mechanism:

$$\mathcal{F}(d) = f(d) + z, \tag{3}$$

where z is a random variable from distribution $\mathcal{N}(0, \sigma^2 {s_f}^2)$. Here, when $\varepsilon \in [0, 1]$, the relation among the parameters of Gaussian mechanism [12] is such that

$$\sigma^2 \varepsilon^2 \geqslant 2 \ln \left(1.25/\delta\right) s_f^2.$$

2.2 Deep Learning

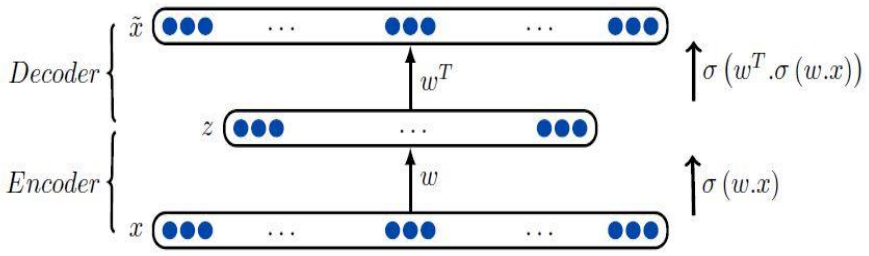
Deep learning is a subfield of machine learning that can be either supervised or unsupervised. The power of deep learning comes from discovering essential concepts of data as nested hierarchy concepts where simpler concepts are refined to Obtain complex concepts. Deep learning has been applied to many different research areas including computer vision, speech recognition and bio-informatics. We focus on the auto-encoder, an unsupervised deep learning technique that outputs a reconstruction of its input.

Similar to an artificial neural network, an auto-encoder is trained by optimizing an objective function. Stochastic gradient descent (SGD) is used as a scalable technique to solve this optimization problem. Rather than iterating over every Training instance, SGD iterates over a mini-batch of the instances

At each step t, model gradient is computed for a given batch B_t and learning parameter η. Then, the model parameter is updated for the next step as follows:

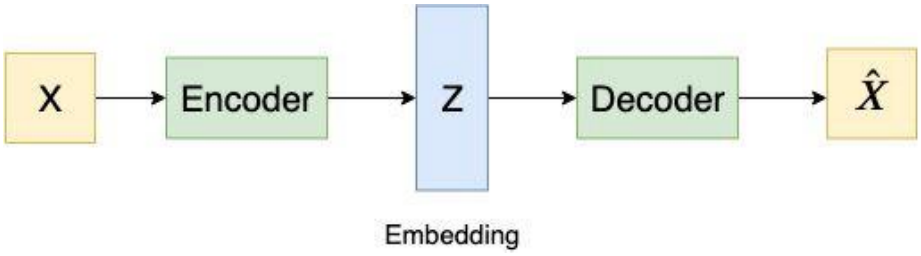
$$w_{t+1} = w_t - \eta \left(\frac{1}{|B_t|} \sum_{x_i \in B_t} \nabla_w \, \ell(w; x_i) \right). \tag{5}$$

Figure 1 presents two main phases of an auto-encoder: the **encoder** and the **decoder**.



.

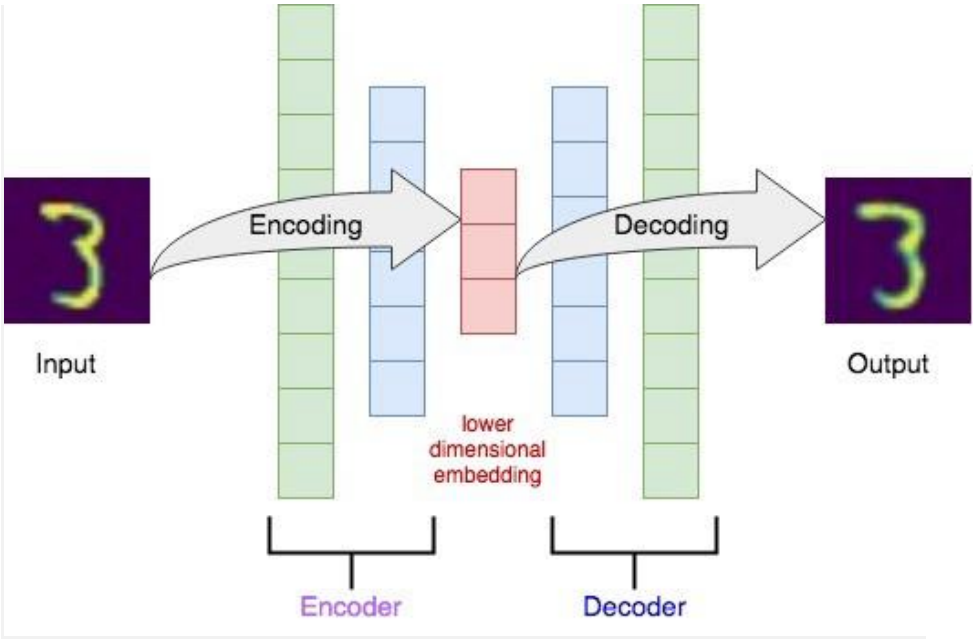
2.3 Traditional Auto-Encoders



The architecture of the traditional autoencoder

The traditional autoencoder is a neural network that contains an encoder and a decoder. The encoder takes a data point *X* as input and converts it to a lower-dimensional representation (embedding) *Z*. The decoder takes the lower-dimensional representation *Z* and returns a reconstruction of the original input *X-hat* that looks like the input *X*. The quality of the embedding determines the quality of the output *X-hat*. However, it might not be possible for the encoder to encode all information because the embedding has a lower dimensionality than the input. Therefore, if the embedding captures more information from the input, the output will have a better performance.

The architecture of Encoder and Decoder



Traditional autoencoder for MNIST dataset

The MNIST dataset contains grayscale images and each image has a shape of 28 by 28 pixels. The encoder takes this image and converts it to a lower-dimensional embedding *Z*. The decoder takes the embedding *Z* and returns a reconstructed input image, which is the output of the autoencoder.

Loss Function

The loss function of an autoencoder measures the information lost during the reconstruction. We want to

minimize the reconstruction loss to make *X-hat* closer to *X*. We often use mean square error as our loss function, which measures how close *X-hat* is to *X*.

$$L(X, \hat{X}) = ||X - \hat{X}||^2$$

2.4 Variational Autoencoders

2.4.1Why do we need the variational autoencoders?

One of the biggest advantages of the variational autoencoder is that VAE could generate new data from the original source dataset. In contrast, traditional autoencoder could only generate images that are similar to the original inputs. Let's say you want to build a garden that is filled with bushes. Every single bush needs to be different so your garden would look real. You definitely can't draw every single bush by yourself, a smarter way is using variational autoencoder to automatically generate new bushes for your garden.

Main idea

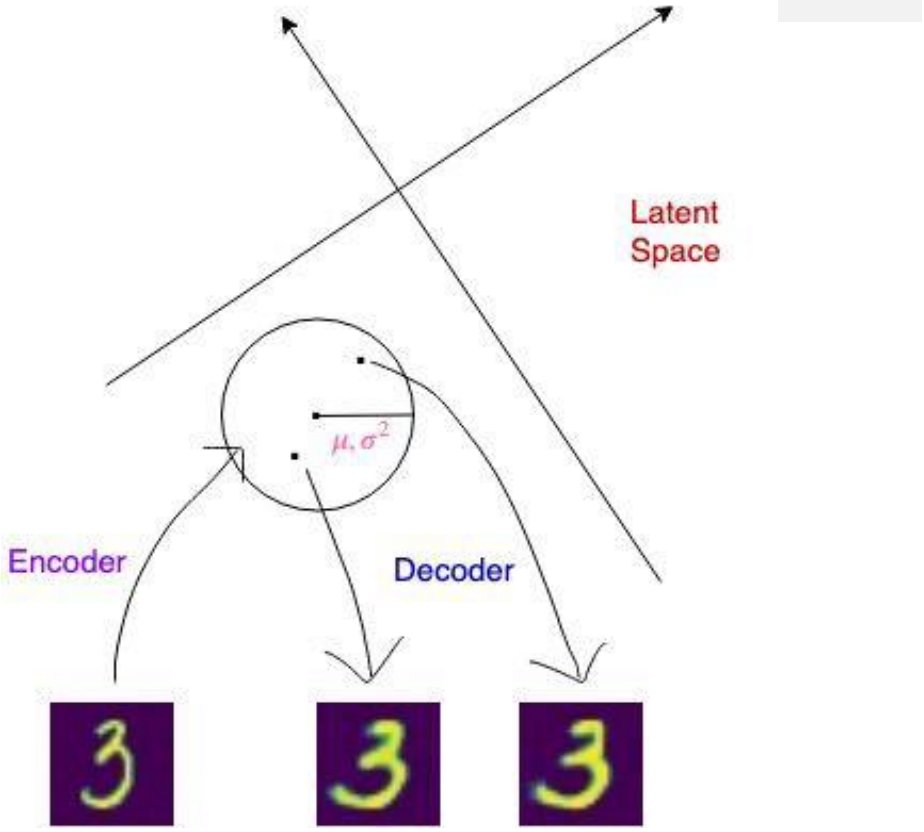
The main idea of a variational autoencoder is that it embeds the input *X* to a distribution rather than a point.

And then a random sample *Z* is taken from the distribution rather than generated from encoder directly.

1.3 The architecture of the Encoder and Decoder

The encoder for a VAE is often written as $q\varphi(z|x)$, which takes a data point *X* and produces a distribution. The distribution is usually parameterized as a multivariate Gaussian. Therefore, the encoder predicts the means and standard deviation of the Gaussian distribution. The lower-dimensional embedding *Z* is sampled from this distribution. The decoder is a variational approximation, $p\theta(x|z)$, which takes an embedding *Z* and produces the output *X-hat*.

An example of a variational autoencoder



1.4 Loss Function VAE

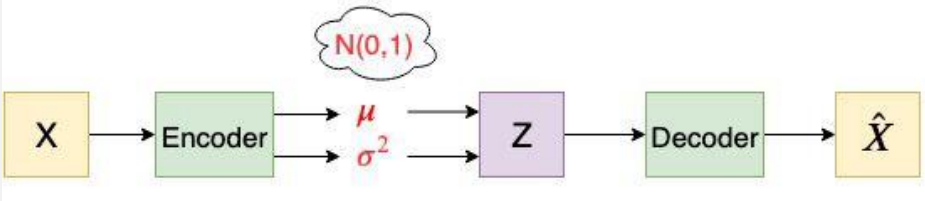
The loss function for VAE has two parts. The first part of the loss function is called the variational lower bound, which measures how well the network reconstructs the data. If the reconstructed data X is very different than the original data, then the reconstruction loss will be high. The second part of the loss function works as a regularizer. It is

the KL-divergence of the approximate from the true posterior (*p(z)*), which measures how closely the output distribution (*qφ(z|x)*) match to *p(z)*.

$$l_i(\theta, \phi) = -E_{z \sim q_\phi(z|x_i)}[log_{p_\theta}(x_i|z)] + KL(q_\phi(z|x_i)||p(z))$$

1.5 Summary

The idea of VAE can be generalized by the image below:

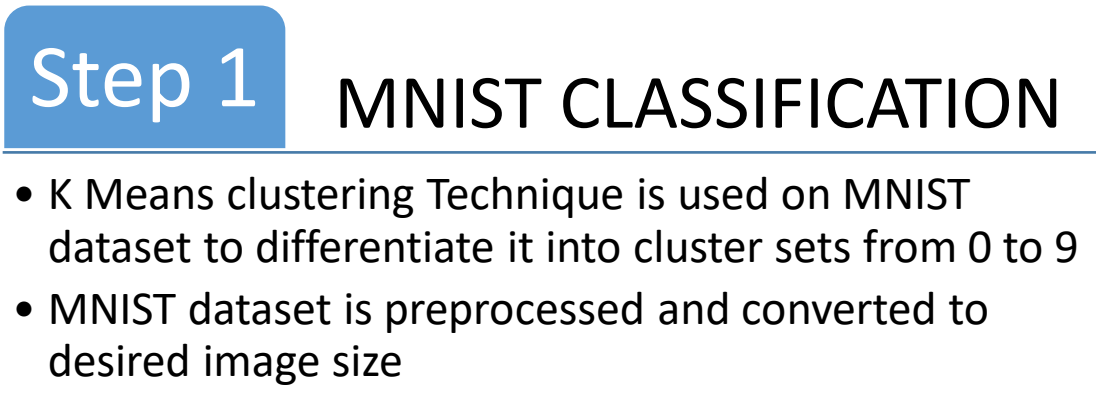


The architecture of the variational autoencoder

The encoder takes a datapoint *X* as input and generates μ and logσ² as outputs. The reason that we use logσ² instead of σ² is that σ² is non-negative, so we will need an extra activation function. But logσ² could be positive or negative. After we get μ and logσ², we try to make both μ and logσ² close to 0, which means μ is close to 0 and σ is close to 1. Therefore the final distribution will be close to *N(0,1)*. Finally, we want to generate the embedding *Z* from μ and σ

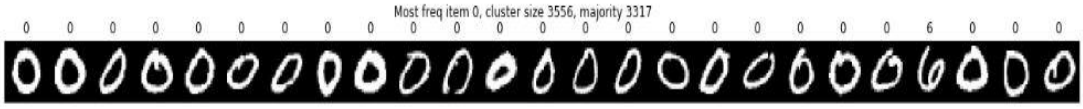
by $z = \mu + \sigma * \varepsilon$, where $\varepsilon \sim N(0,1)$. This is called the *reparameterization trick*. Now with the latent variable *Z*, we can generate our output *X-hat* through the decoder.

3) METHODOLOGY

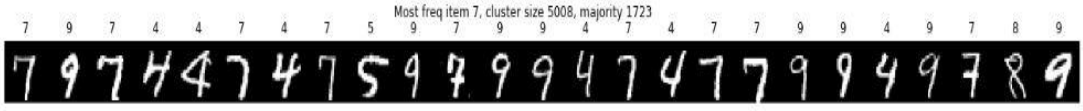


Plotting an extract of the 10 clusters, overall purity: 0.580214

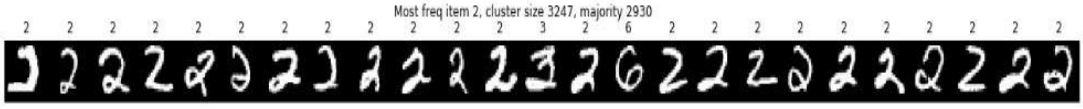
Most freq item 0, cluster size 3556, majority 3317



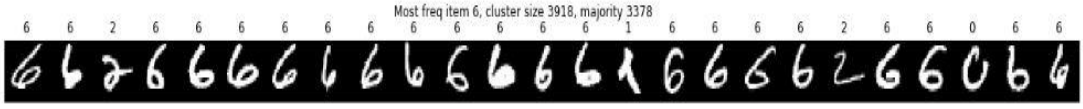
Most freq item 7, cluster size 5008, majority 1723



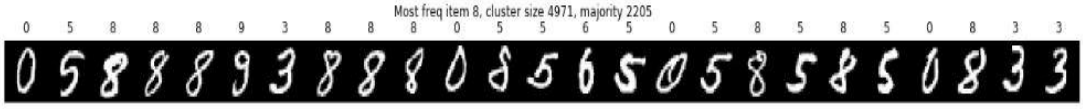
Most freq item 2, cluster size 3247, majority 2930



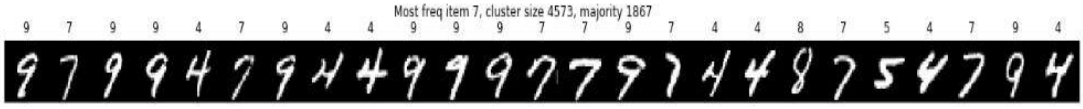
Most freq item 6, cluster size 3918, majority 3378



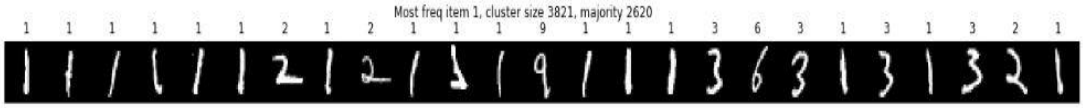
Most freq item 8, cluster size 4971, majority 2205



Most freq item 7, cluster size 4573, majority 1867



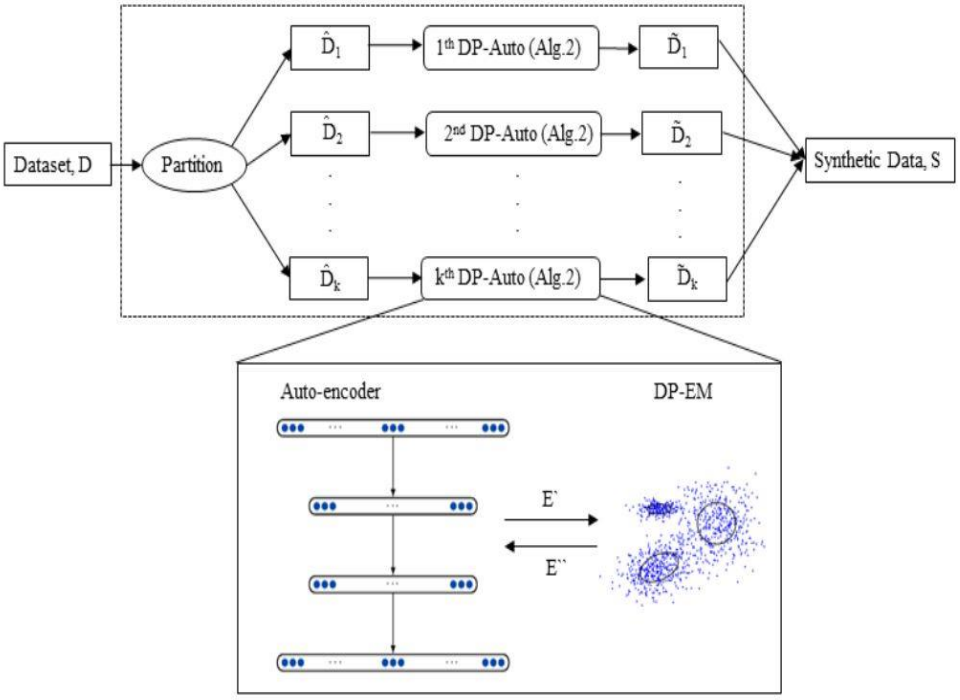
Most freq item 1, cluster size 3821, majority 2620



Step 2

VARIATIONAL AUTOENCODER

- A Artificial Neural Network is created with two Dense layer , one internal layer with relu activation function and reduced size and another output layer with sigmoid function.
- Autoencoder Model is applied on it & compilation of model is done
- The Autoencoder if fitted over the dataset
- Finally the weights obtained from training the model are saved for future processing



Our private auto-encoder employs steps to improve the optimization process with gradient computation and clipping. While a gradient is computed for a batch in the standard stochastic training techniques, we compute the gradient for each

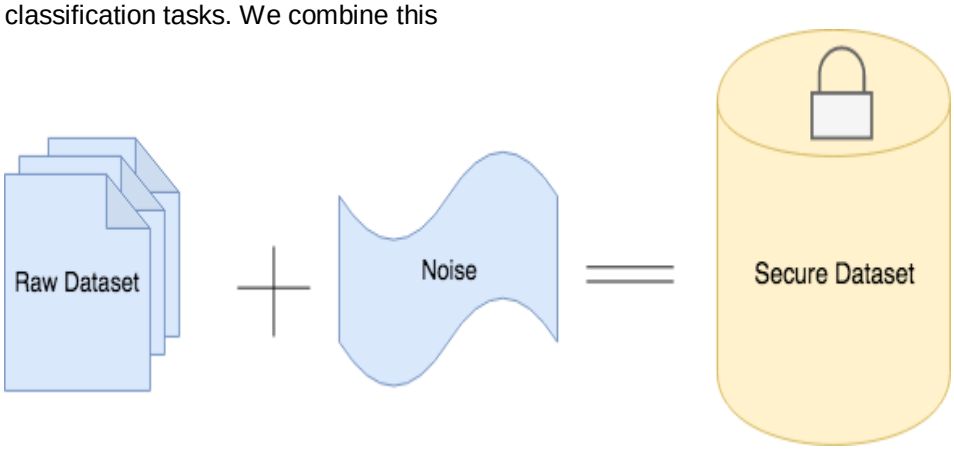
training instance instead. This approach improves the optimization process since it reduces the sensitivity of the gradient present at each instance. Norms of the gradients define the direction that optimizes the network parameters. However, in some deep networks, the gradients can be unstable and actuate in a large range. Such actuations can inhibit the learning process due to the increased vulnerability of the networks. To avoid this undesired situation, we bound norms of the previously

computed gradients by a clipping constant C. After clipping the gradients, noise is sampled from the Gaussian distribution with zero mean and standard deviation of C and added to the previously clipped gradients.

Step 3 Guassian Noise

- We infuse guassian noise with different epsilon value in the saved weights.
- Then these updated weights are loaded into the autoencoder model and the results are compared with the original output.

Our main framework aims to generate synthetic data without sacrificing the utility. A similar approach is proposed which designs a private convolutional neural network on supervised learning. However, this method can only be used in classification tasks. We combine this



method with DP-EM and to create a generative deep learning model.

Assume that we have the sensitive dataset D = { (x_1, y_1) , ... , (x_m, y_m) }, where every instance x has a label y. We partition the sensitive dataset D into k groups such that every instance x in a group has the same label y. The value of k is limited by the number of unique labels in dataset D. ((For this process we used K-Means Clustering technique on MNIST data set))

4) SOURCE CODE

****
****
START
****
****

```
#!/usr/bin/env python
# coding: utf-8
```

#Importing libraries

```
import tensorflow as tf
import keras
from keras.models import Model
from keras.layers import Dense,Input
from keras.datasets import mnist
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from keras.models import load_model
from sklearn.metrics import confusion_matrix
from keras.layers import GaussianNoise
```

#Input Data

(X_train,Y_train),(X_test,Y_test) = mnist.load_data()

#Data Preprocessing

X_train = X_train.astype('float32')

X_test = X_test.astype('float32')

X_train /= 255

X_test /= 255

#Reshaping model size (n,784)

X_train = X_train.reshape(len(X_train) ,np.prod(X_train.shape[1:]))

X_test = X_test.reshape(len(X_test),np.prod(X_test.shape[1:]))

Autoencoder Model

#Creating Autoencoder model

val = 64

img = Input(shape=(784,))

encoded = Dense(val, activation = 'relu')(img)

decoded = Dense(784, activation = 'sigmoid')(encoded)

autoencoder = Model(img,decoded)

encoder = Model(img,encoded)

```
encoded_val= Input(shape= (val,))


decoder_layer = autoencoder.layers[-1]


decoder = Model(encoded_val, decoder_layer(encoded_val))
```

#compiling the model

```
autoencoder.compile(optimizer= 'adam', loss = 'mse', )


checkpoint_filepath = '/tmp/checkpoint'
model_checkpoint_callback = keras.callbacks.ModelCheckpoint(
  filepath=checkpoint_filepath,
  save_weights_only=True,
  monitor='val_loss',
  save_best_only=True)
```

#fitting data into model

#1) Original data

```
autoencoder.fit(X_train, X_train, batch_size=32, epochs = 100, validation_data=
(X_test,X_test),callbacks=[model_checkpoint_callback] )
autoencoder.summary()


encoded_images= encoder.predict(X_test)
decoded_images = decoder.predict(encoded_images)
```

Addition of Gaussian Noise under DP std=0.1 and results

```
import matplotlib
index=np.random.randint(0,10000,10)
autoencoder.load_weights(checkpoint_filepath)
figure, ax = plt.subplots(1,10)
for ind,title in enumerate(index):
 ax.ravel()[ind].imshow(X_test[title].reshape(28,28))
 plt.title('original')
 plt.rcParams["figure.figsize"] = (50,50)
plt.show()

figure, ax = plt.subplots(1,10)
for ind,title in enumerate(index):
 decode_image=autoencoder.predict(X_test[title].reshape(1,784))
 decode_image=decode_image.reshape(784,)
 ax.ravel()[ind].imshow(decode_image.reshape(28,28))
 plt.title('decoded')
 plt.rcParams["figure.figsize"] = (50,50)
 plt.show()
```

noise to be added

#std=senstivity/epsilon

```
std=0.1
old_weights = autoencoder.layers[1].get_weights()
# adding gaussian noise in bias of first layer
old_weights[1]=old_weights[1]+np.random.normal(0,std,64)
# adding gaussiann noise in weights of first layer
for i in range(784):
 old_weights[0][i]=old_weights[0][i]+np.random.normal(0,std,64)

old_weights_2=autoencoder.layers[2].get_weights()
# adding gaussian noise in bias of second layer
old_weights_2[1]=old_weights_2[1]+np.random.normal(0,std,784)
# adding gaussiann noise in weights of second layer
for i in range(64):
 old_weights_2[0][i]=old_weights_2[0][i]+np.random.normal(0,std,784)

model1=autoencoder
model1.layers[1].set_weights(old_weights)
model1.layers[2].set_weights(old_weights_2)



figure, ax = plt.subplots(1,10)
for ind,title in enumerate(index):
 decode_image=model1.predict(X_test[title].reshape(1,784))
 decode_image=decode_image.reshape(784,)
 ax.ravel()[ind].imshow(decode_image.reshape(28,28))
 plt.title('after noise addition')
```

```
 plt.rcParams["figure.figsize"] = (50,50)
plt.show()
```

A DNN Model

```
truemodel=tf.keras.Sequential([
tf.keras.layers.Dense(512),
tf.keras.layers.Dense(256),
tf.keras.layers.Dense(128),
tf.keras.layers.Dense(64),
tf.keras.layers.Dense(10)
])

checkpoint_filepath = '/tmp/checkpoint1'
model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
   filepath=checkpoint_filepath,
   save_weights_only=True,
   monitor='val_loss',
   save_best_only=True)

truemodel.compile(optimizer=('adam'),loss=tf.keras.losses.SparseCategoricalCr
ossentropy(from_logits=True),metrics=['accuracy'])
truemodel.fit(X_train,Y_train,epochs=50,validation_data=(X_test,Y_test),callback
s=[model_checkpoint_callback])
```

```
truemodel.load_weights(checkpoint_filepath)
results = truemodel.evaluate(X_test, Y_test, batch_size=128)
print('truemodel')
print("test loss, test acc:", results)
```

#A DNN Model with Differential Private Dataset**

```
xtrain_noisy=model1.predict(X_train)
xtest_noisy=model1.predict(X_test)



testmodel=tf.keras.Sequential([
tf.keras.layers.Dense(512),
tf.keras.layers.Dense(256),
tf.keras.layers.Dense(128),
tf.keras.layers.Dense(64),
tf.keras.layers.Dense(10)
])

checkpoint_filepath = '/tmp/checkpoint2'
model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
  filepath=checkpoint_filepath,
  save_weights_only=True,
  monitor='val_loss',
  save_best_only=True)
```

```
testmodel.compile(optimizer=('adam'),loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),metrics=['accuracy'])

testmodel.fit(xtrain_noisy,Y_train,epochs=50,validation_data=(xtest_noisy,Y_test),callbacks=[model_checkpoint_callback])



testmodel.load_weights(checkpoint_filepath)

results = testmodel.evaluate(xtest_noisy, Y_test, batch_size=128)

print('testmodel')

print("test loss, test acc:", results)
```

Adding Gaussian Noise to Model weights under DP constraints std=0.05

```
truemodel.load_weights(checkpoint_filepath)

results = truemodel.evaluate(X_test, Y_test, batch_size=128)

print('truemodel')

print("test loss, test acc:", results)


testmodel.load_weights(checkpoint_filepath)

results = testmodel.evaluate(xtest_noisy, Y_test, batch_size=128)

print('testmodel')

print("test loss, test acc:", results)
```

 # noise to be added

#std=senstivity/epsilon

std=0.05

old_weights = truemodel.layers[0].get_weights()

 # adding gaussian noise in bias of first layer

old_weights[1]=old_weights[1]+np.random.normal(0,std,512)

 # adding gaussiann noise in weights of first layer

for i in range(784):

 old_weights[0][i]=old_weights[0][i]+np.random.normal(0,std,512)

##
#####################################

old_weights_2=truemodel.layers[1].get_weights()

 # adding gaussian noise in bias of second layer

old_weights_2[1]=old_weights_2[1]+np.random.normal(0,std,256)

 # adding gaussiann noise in weights of second layer

for i in range(512):

 old_weights_2[0][i]=old_weights_2[0][i]+np.random.normal(0,std,256)

##
#####################################

old_weights_3=truemodel.layers[2].get_weights()

 # adding gaussian noise in bias of third layer

old_weights_3[1]=old_weights_3[1]+np.random.normal(0,std,128)

 # adding gaussiann noise in weights of third layer

```
for i in range(256):
 old_weights_3[0][i]=old_weights_3[0][i]+np.random.normal(0,std,128)



##########################################################
#######################################
old_weights_4=truemodel.layers[3].get_weights()
# adding gaussian noise in bias of fourth layer
old_weights_4[1]=old_weights_4[1]+np.random.normal(0,std,64)
# adding gaussiann noise in weights of fourth layer
for i in range(128):
 old_weights_4[0][i]=old_weights_4[0][i]+np.random.normal(0,std,64)



##########################################################
#######################################
old_weights_5=truemodel.layers[4].get_weights()
# adding gaussian noise in bias of fifth layer
old_weights_5[1]=old_weights_5[1]+np.random.normal(0,std,10)
# adding gaussiann noise in weights of fifth layer
for i in range(64):
 old_weights_5[0][i]=old_weights_5[0][i]+np.random.normal(0,std,10)


model1=truemodel
model1.layers[0].set_weights(old_weights)
model1.layers[1].set_weights(old_weights_2)
model1.layers[2].set_weights(old_weights_3)
```

```
model1.layers[3].set_weights(old_weights_4)

model1.layers[4].set_weights(old_weights_5)

results = model1.evaluate(X_test, Y_test, batch_size=128)

print('model with gaussian noise')

print("test loss, test acc:", results)
```

****
**********END OF CODE**********
****

5) TOOLS TO BE USED

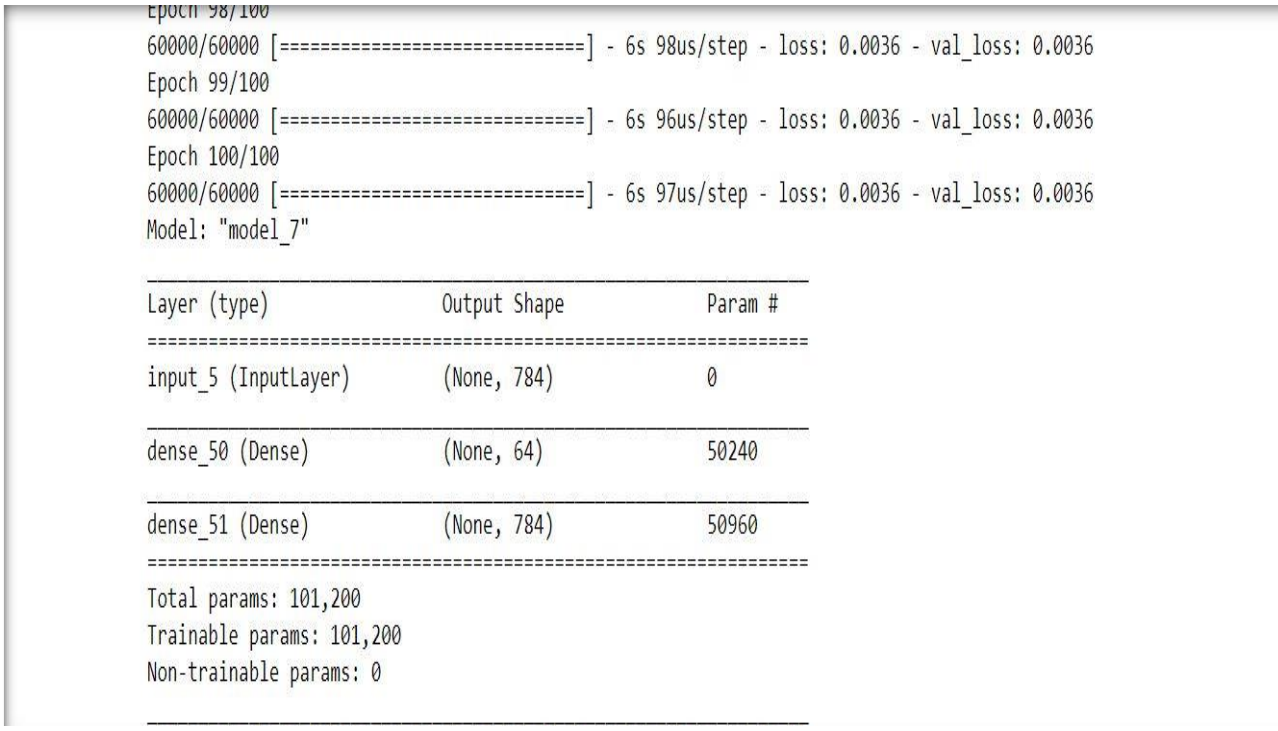
- Programming language used : Python.
- Libraries Used : Python(pandas, numpy, scikit-learn, Tensorflow, matplotlib, seaborn, Keras)
- Framework : Anaconda
- Platform Used : Jupyter Notebook, Spyder.

6) CONCLUSION

We propose a new generative deep learning method that produces synthetic data from a dataset while preserving the utility of the original dataset. Our generative auto-encoder method partitions the original data into groups, and then employs the private auto-encoder for each group. Auto-encoder learns the latent structure of each group, and uses expectation maximization algorithm to

simulate them. This approach eliminates impurity of groups and results in more accurate.
Also we save the excessively large computation time required by modifying the weights, which eliminates the trouble of fitting the data into the model for every twerk made.
The results are shown below for the accuracy of our model :

1) Auto Encoder Model :



```
Epoch 98/100
60000/60000 [==============================] - 6s 98us/step - loss: 0.0036 - val_loss: 0.0036
Epoch 99/100
60000/60000 [==============================] - 6s 96us/step - loss: 0.0036 - val_loss: 0.0036
Epoch 100/100
60000/60000 [==============================] - 6s 97us/step - loss: 0.0036 - val_loss: 0.0036
Model: "model_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_5 (InputLayer)         (None, 784)               0
_________________________________________________________________
dense_50 (Dense)             (None, 64)                50240
_________________________________________________________________
dense_51 (Dense)             (None, 784)               50960
=================================================================
Total params: 101,200
Trainable params: 101,200
Non-trainable params: 0
_________________________________________________________________
```

2) Accuracy of model before noise addition: **(Accuracy => 93.6%)**

```
In [116]: testmodel.load_weights(checkpoint_filepath)
          results = testmodel.evaluate(xtest_noisy, Y_test, batch_size=128)
          print('testmodel')
          print("test loss, test acc:", results)
```

```
79/79 [==============================] - 0s 2ms/step - loss: 0.2138 - accuracy: 0.9364
testmodel
test loss, test acc: [0.2137981653213501, 0.9363999962806702]
```

3) Accuracy after adding the Gaussian Noise: **(Accuracy => 76.27%)**

```
79/79 [==============================] - 0s 3ms/step - loss: 0.2880 - ac
truemodel
test loss, test acc: [0.288004070520401, 0.9218000173568726]
79/79 [==============================] - 0s 3ms/step - loss: 3.6266 - ac
testmodel
test loss, test acc: [3.6265506744384766, 0.4871000051498413]
79/79 [==============================] - 0s 2ms/step - loss: 0.8968 - ac
model with gaussian noise
test loss, test acc: [0.8967832326889038, 0.7627000212669373]
```

7) REFERENCES

1. Abadi, M., Chu, A., Goodfellow, I., McMahan, H.B., Mironov, I., Talwar, K., Zhang, L.: Deep learning with di_erential privacy. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 308{318. ACM (2016)Steeve Huang. Introduction to recommender system. part 1 (collaborative filtering, singular value decomposition).

2. Machine Learning and Differential Privacy Maria-Florina Balcan

3. Privacy Preserving Synthetic Data Release Using Deep Learning (Nazmiye Ceren Abay, Yan Zhou, Murat Kantarcioglu, Bhavani Thuraisingham, and Latanya Sweeney)

4. **Deep Learning with Differential Privacy (** Martín AbadI, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Kunal Talwar)

5. https://towardsdatascience.com/tutorial-on-variational-graph-auto-encoders-da9333281129