Deep Learning Assignment 3 Report

Generative Models with the MNIST Dataset: Variatioonal Autoencoders and Generative Adversarial Network

Taylan Tatlı

Physics PhD student

2022804009

taylan.tatl@gmail.com

CMPE 597 Assignment 3

Department of Computer Engineering,
Boğaziçi University, 31/05/2023

Contents

1 Introduction

In this assignment, the subject is to implement generative models with the MNIST dataset. Two architecture is used, which are Variational AutoEncoder(VAE) and Generative Adversarial Network (GAN). Network architectures is given in two main sections. The MNIST dataset has 70,000 small square 28×28 pixel grayscale images, handwritten numbers, which will correspond to one of 10 classes, integer values from 0 to 9.

2 Variational Autoencoder

Autoencoders are a type of neural network that can be used to learn deep parameters of input data. The network first applies transformations that map the input data into a lower dimensional space which is the encoder part of the network. Then, the network uses the encoded data to recreate the inputs which is the decoder [1]. Using variational autoencoders, it's not only possible to compress data, also possible to generate new objects that the autoencoder has seen before, since it simply tells the distribution creating the new vectors.

2.1 Network, Model_VAE.py

The model network contain two main part :Encoder and Decoder.

2.1.1 Encoder Network

For this part, a single-layer LSTM encoder is implemented. The determined hidden dimension is 64. The LSTM functions are used by the TensorFlow Kerase deep learning library for implementation.

2.1.2 Decoder Network

For this part, a convolutional decoder is implemented with transposed convolutional layers. For the generation process, we need combine operations of an UpSampling2D layer followed by a normal Conv2D layer and Transposed Convolutional layers are doing that, combining these two. The decoder takes a random vector sampled from the distribution that was outputted by the encoder and decodes it to a 28×28 grayscale image. The number of layer is 6, kernel size 3 for each, fileter are 32, 64 and 64 for transposed convolutions. The summary of both encoder and decoder is given in Figure 1.

```
----- Encoder -----
input hidden_dim:  64
input x:  (None, 28, 28)
lstm:  (None, 28, 64)
flatten_lstm:  (None, 1792)
mu:  (None, 64)
sigma:  (None, 64)
epsilon:  (None, 64)
z:  (None, 64)
```

(a) Encoder

```
----- Decoder -----
input z:  (None, 64)
1. fc:  (None, 98)
2. fc:  (None, 196)
2. reshaped_fc:  (None, 14, 14, 1)
3. tconv:  (None, 14, 14, 32)
3. dropout:  (None, 14, 14, 32)
4. tconv:  (None, 14, 14, 64)
4. dropout:  (None, 14, 14, 64)
5. tconv:  (None, 28, 28, 64)
5. flatten_tconv:  (None, 50176)
6. fc:  (None, 784)
6. reshaped_output:  (None, 28, 28)
```

(b) Decoder

Figure 1: Both Encoder and Decoder Network summary is given.

2.2 Loss Functions and KL-Divergence Term

The implemented Loss Funciton is binary-cross entropy and also as a regularization term KL-Divergence is implemented, which makes sure our latent values are sampled from a normal distribution. Code is given in Figure 2, and the result will be given in Training part.

```
def loss(self, decoder, mu, sigma):
  flatten_y = tf.reshape(self.y, shape=[-1, 28 * 28])
  unreshaped = tf.reshape(decoder, [-1, 28 * 28])
  unreshaped = tf.clip_by_value(unreshaped, 1e-7, 1-1e-7)

  # binary cross entropy loss (reconstruction loss)
  binary_loss = tf.reduce_sum(-flatten_y*tf.math.log(unreshaped)-(1.0 - flatten_y)*tf.math.log(1.0 - unreshaped), axis=1)

  # kl divergence loss:
  kl_loss = -0.5 * tf.reduce_sum(1.0 + 2.0 * sigma - tf.square(mu) - tf.exp(2.0 * sigma), axis=1)

  # general loss
  general_loss = tf.reduce_mean(binary_loss + kl_loss, axis=0)

  return general_loss, binary_loss, kl_loss
```

Figure 2: Loss code implementation.

2.3 Training

Some chosen parameters for training are hidden space dimension is 64, epoch number is 100 and batch size is 256. The picture of training loop code is given in Figure 3.

2.3.1 Total VAE loss, KL divergence and Reconstruction loss

For each epoch loss values are calculated and after 100 epoch training, the Loss values are given in Figure 4.

```
model_fid = InceptionV3(include_top=False, pooling='avg', input_shape=(299,299,3))
# main training loop
plot_general_loss_, plot_rec_loss_, plot_kl_loss_, FID_epoch = [], [], [],[]

for epoch in range(epoch_number):

    number_of_batch = int(len(X_train) / batch_size)

    general_loss_, rec_loss_, kl_loss_,fid_batch = 0, 0, 0, 0

    for i in range(number_of_batch):

        batch = X_train[i*batch_size:(i+1)*batch_size]  # we take only data (not labels)
        batch = np.reshape(batch, [-1, 28, 28])  # reshape the data

        # train the network for one batch
        _, general_loss_, rec_loss_, kl_loss_, decoder_ = sess.run(
            [train_optimizer, general_loss, rec_loss, kl_loss, decoder_output],
            feed_dict={x: batch, y: batch, train_mode: True})

        if i == number_of_batch-1 and epoch%10==0: # For FID values
            randoms = [np.random.normal(0, 1, hidden_dim) for _ in range(100)]
            decoder_images = sess.run(decoder_output, feed_dict = {z: randoms, train_mode: False})
            #plt.imshow[decoder_images[0]]
            images1=scale_images(batch,new_shape=(299,299,3))
            images2=scale_images(decoder_images,new_shape=(299,299,3))
            fid_batch = calculate_fid(model_fid,images1,images2)
            print("Fid Value:",fid_batch)
            FID_epoch.append(fid_batch)
        if i == 0: # at the end of the each epoch, or at the beginning.
            print(
                "|--- Epoch: {} \t---> General loss: {:.2f} | Reconstruction loss: {:.2f} | KL-divergence loss: {:.2f}".format(
                    epoch, general_loss_, np.mean(rec_loss_), np.mean(kl_loss_)))

            plot_general_loss_.append(general_loss_)
            plot_rec_loss_.append(np.mean(rec_loss_))
            plot_kl_loss_.append(np.mean(kl_loss_))
            #FID_epoch.append(fid_batch)
```

Figure 3: Training Loop for VAE.

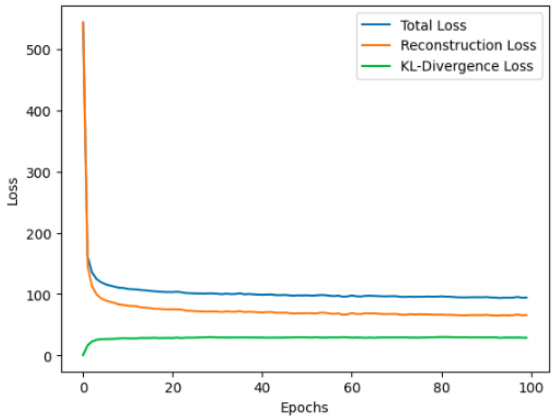
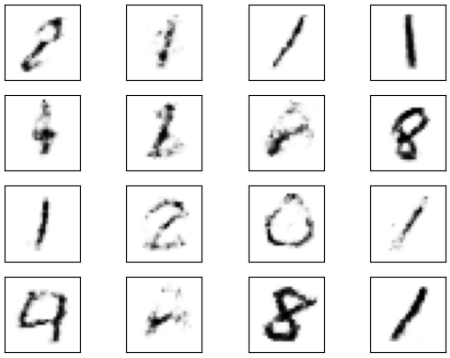


Figure 4: Loss Values for 100 Epoch.

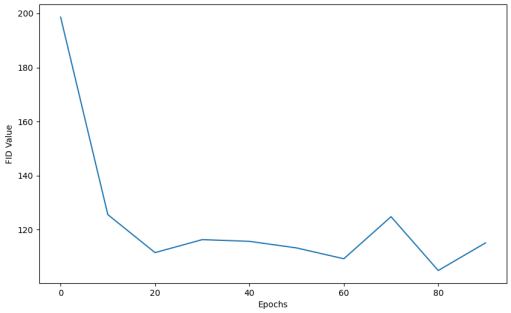
2.3.2 Generated Images, FID Values

After saved model, by loading the trained decoder, and 100 images from 100 random vectors are generated, a part of example is given in Figure 5. Also, quantitative visual perception metrics, Firchet Inception Distance (FID) values are given in Figure 5.

The inspired reference for FID implementation is [2]. If we investigate the FID values, we see it started with high value for beginning of training, which we have high total loss, but as the training goes, we can easily see that both FID values and total loss decreases and at the point that the are note decreasing, they shows parallel characteristics to each other. However, at the end, we don't have very good result. If we can see the generated images figure, it is not well drawn figure and so corresponding FID value is not very good,



(a) Generated Images



(b) FID Values

Figure 5: Generated images at the end and FID values for each 10 epochs.

but our FID values are parallel to our model performance,loss functions and generally decreasing so our model generates images better generally after every 10 epochs training.

Other references for VAE code implementation goes here: [3, 4]

3 Generative Adversarial Network

The images in the dataset feed the training loop of created Generative Adversarial Network (GAN). The generator model learns how to create new handwritten digits between 0 and 9, by using a discriminator which distinguishes real images from the dataset and outputs (new image from the generator model). Details are given in following sections

3.1 Network, Model_GAN.py

Model (network) of GAN has two main parts, one is Discrminator and the other is Generator. Two of them uses the convolutional layers and At the end, a defined GAN function (model) combine two to create whole model such that the generator receives as input random points in the latent space and generates samples too feed the discriminator network and the output of this can be used to update the model weights of the generator. The inspired reference for code implementation of GAN is [5].

3.1.1 Discrminator Network

Discrminator model is a binary classification problem. Image with one channel and with sized of 28x28 in pixels are inputs and whether the sample is real or fake is the output. The written model has two convolution layers with 64 filters, kernel size of 3, stride of 2. It has no pooling layers, and at the end, contains sigmoid activation function. The Loss function is binary cross entropy loss. The other activation function used between layer are LeakyRelu, optimizer is Adam with 0.0001 learning rate, and also dropout regularization teqnique is used between convolutional layers. The model architecture is given in below. The 2×2 stride provide down-sample the input image, first from 28×28 to 14×14, and finally 7×7, at that point the discrminator makes an prediction.

3.1.2 Generator Network

The generator model creates new and fake but real-like images of handwritten digits. It takes a point from the latent space (a vector space defined randomly and of values with

a Gaussian distribution) as input and gives image output(in our case, a square greyscale image.) The generator model will assign meaning to klatent space points and so the latent vector space represents a compressed representation of the output space which only generator knows in the sense of generating images. It involves two main elements. The first is a Dense layer having enough nodes to represent a low-resolution version of the output image. The next major architectural innovation involves upsampling the low-resolution image to a higher resolution version of the image. One way is to use Conv2DTranspose (a way that combine operations of an UpSampling2D layer followed by a normal Conv2D layer). The model summary outputs are given in Figure 6.

Layer (type)	Output Shape	Param #
conv2d_7 (Conv2D)	(None, 14, 14, 64)	640
leaky_re_lu_11 (LeakyReLU)	(None, 14, 14, 64)	0
dropout_6 (Dropout)	(None, 14, 14, 64)	0
conv2d_8 (Conv2D)	(None, 7, 7, 64)	36928
leaky_re_lu_12 (LeakyReLU)	(None, 7, 7, 64)	0
dropout_7 (Dropout)	(None, 7, 7, 64)	0
flatten_3 (Flatten)	(None, 3136)	0
dense_6 (Dense)	(None, 1)	3137

Total params: 40,705
Trainable params: 40,705
Non-trainable params: 0

(a) Discriminator

Layer (type)	Output Shape	Param #
dense_5 (Dense)	(None, 6272)	633472
leaky_re_lu_8 (LeakyReLU)	(None, 6272)	0
reshape_1 (Reshape)	(None, 7, 7, 128)	0
conv2d_transpose (Conv2DTranspose)	(None, 14, 14, 128)	262272
leaky_re_lu_9 (LeakyReLU)	(None, 14, 14, 128)	0
conv2d_transpose_1 (Conv2DTranspose)	(None, 28, 28, 128)	262272
leaky_re_lu_10 (LeakyReLU)	(None, 28, 28, 128)	0
conv2d_6 (Conv2D)	(None, 28, 28, 1)	6273

Total params: 1,164,289
Trainable params: 1,164,289
Non-trainable params: 0

(b) Generator.

Figure 6: Model Summary for both Discriminator and Generator Networks

3.2 Training

The weights in the generator model are updated based on the performance of the discriminator model. This defines adversarial relationship between these two models. Some chosen parameters for training are latent dimension is 100, epoch number is 100 and batch size is 256. The picture of training code is given in Figure 7.

3.2.1 Generator and Discriminator Loss

The generator is responsible for producing new images by studying original inputs, so we want to minimize this, on the other hand, the discriminator aims at classifying data of

```python
# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=100, n_batch=256):
    bat_per_epo = int(dataset.shape[0] / n_batch)
    half_batch = int(n_batch / 2)
    d_loss_epoch=[]
    g_loss_epoch=[]
    FID_epoch=[]
    # manually enumerate epochs
    for i in range(n_epochs):
        epoch_d_loss=0
        epoch_g_loss=0
        fid_epoch=0
        # enumerate batches over the training set
        for j in range(bat_per_epo):
            # get randomly selected 'real' samples
            X_real, y_real = generate_real_samples(dataset, half_batch)
            # generate 'fake' examples
            X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
            #Calculate FID
            if j==int(dataset.shape[0]/n_batch)-1 and i%10==0: # at the end of the batch and for each
              X_real_fid=scale_images(X_real,(299,299,3))
              X_fake_fid = scale_images(X_fake,(299,299,3))
              fid_batch = calculate_fid(model_fid,X_real_fid,X_fake_fid)
              print("FID VALUE:", fid_batch, "Epoch:",i)
            # create training set for the discriminator
            X, y = vstack((X_real, X_fake)), vstack((y_real, y_fake))
            # update discriminator model weights
            d_loss, _ = d_model.train_on_batch(X, y)

            epoch_d_loss += d_loss
            # prepare points in latent space as input for the generator
            X_gan = generate_latent_points(latent_dim, n_batch)
            # create inverted labels for the fake samples
            y_gan = ones((n_batch, 1))
            # update the generator via the discriminator's error
            g_loss = gan_model.train_on_batch(X_gan, y_gan)
            epoch_g_loss += g_loss
            # summarize loss on this batch
        epoch_d_loss /= bat_per_epo
        epoch_g_loss /= bat_per_epo
        #fid_epoch/=bat_per_epo
        #print("FID VALUE:", fid_epoch)
        print('>%d, %d/%d, d=%.3f, g=%.3f' % (i+1, j+1, bat_per_epo, epoch_d_loss, epoch_g_loss))
        d_loss_epoch.append(epoch_d_loss)
        g_loss_epoch.append(epoch_g_loss)
        FID_epoch.append(fid_batch)
    return d_loss_epoch, g_loss_epoch, FID_epoch
```

Figure 7: Training loop for GAN.

two classes, fake or real and we want to maximize because, generator will give the fake inputs to the discrminator so the more discriminator loss means better model we have. The generator and discrminator loss curves are given in Figure 8.

The plot is one of expected, oscilates little bit but generator loss decreases and discriminator loss increases.

3.3 Test, Image Generator for GAN

After model saved, model is loaded and tried to generate new images. The generation of each image requires a point in the latent space as input and after that our model will predict these points. The generated images are given in Figure.. and a quantitative visual perception metrics Frchet Inception d-Distance (FID) values for every 10 epoch is given in Figure 9. The inspired reference for FID implementation is [2] If we investigate the FID values, we see it started with high value for beginning of training, which we have

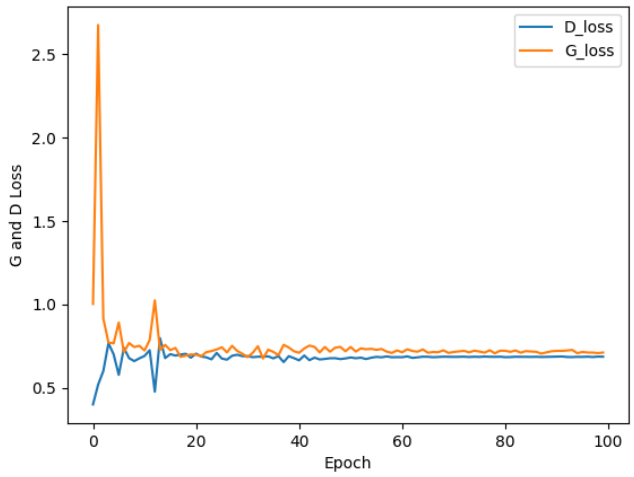
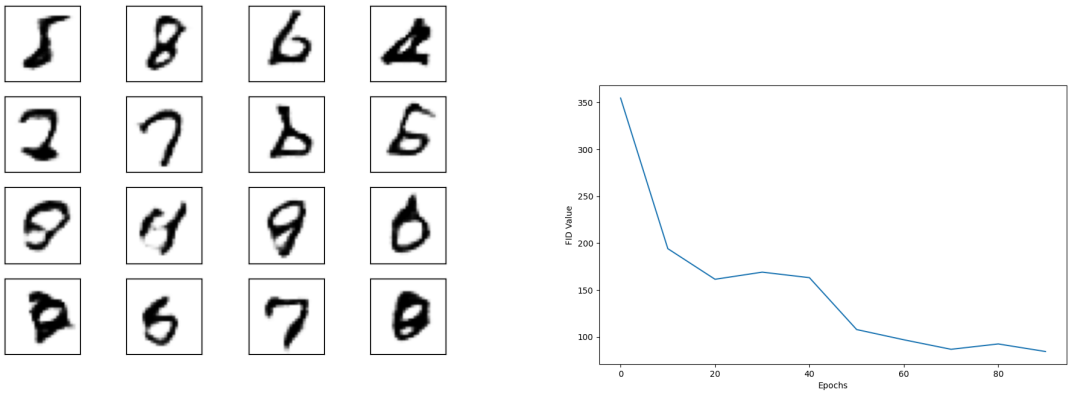


Figure 8: Loss result for GAN.

high generator loss, but as the training goes, we can easily see that both FID values and generator loss decreases and at the point that they are not decreasing, they shows parallel characteristics to each other. However, at the end, we don't have very good result. If we can see the generated images figure, it is not well drawn figure and so corresponding FID value is not very good, but our FID values are parallel to our model performance,loss functions and generally decreasing so our model generates images better generally after every 10 epochs training.



(a) Generated Images (b) FID Values

Figure 9: Generated images at the end and FID values for each 10 epochs.

3.4 WasserStein GAN Implementation

3.4.1 Code Implementation

To implement WasserStein Gan, first a WasserStein Loss is defined and it is given to Discrminator Network loss, also implementation is tried two different optmizers, which are Adam and RMSprop but it diddnt give sensible results (Rest of the code implementation is same with Standard Gan). There should be an mistake on all architecture. The code for model loss and compile is given in Figure..

```
def wasserstein_loss(y_true, y_pred):
 return backend.mean(y_true) * backend.mean(y_pred)
```

(a) WasserStein Loss

```
    # compile model
    if wasserstein==True:
      model.compile(loss=wasserstein_loss, optimizer=tf.keras.optimizers.RMSprop(lr=2e-5), metrics=['accuracy'])
    else:
      model.compile(loss='binary_crossentropy', optimizer=Adam(learning_rate=0.0001), metrics=['accuracy'])
    return model

# load and prepare mnist training images
```

(b) WasserStein Model Compile

Figure 10: Model Implementation for WasserStein GAN.

3.4.2 Generated Images and Loss Plot for WasserStein

Generated Images and Loss Plot for WasserStein GAN are given in Figure11 and Figure 12.

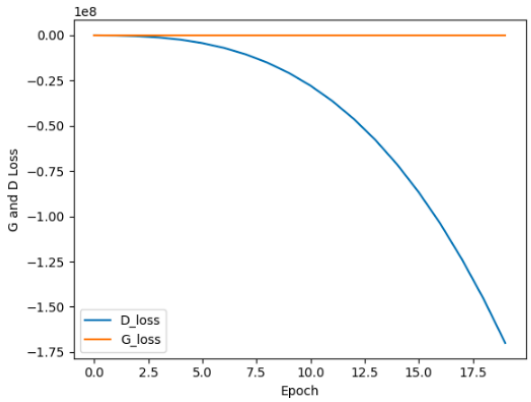


Figure 11: Loss result for WGAN.

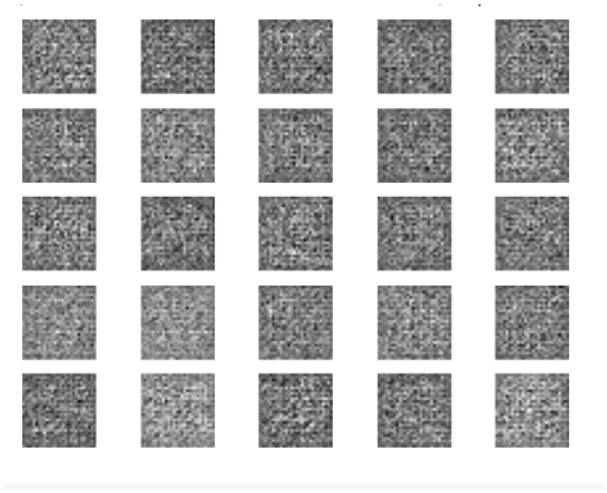


Figure 12: Generated Images result for WGAN.

References

[1] F. Mohr, "Teaching a variational autoencoder (vae) to draw mnist characters." https://towardsdatascience.com/teaching-a-variational-autoencoder-vae-to-draw-mnist-characters-978675c95776.

[2] J. Brownlee, "How to implement the frechet inception distance (fid) for evaluating gans." https://machinelearningmastery.com/how-to-implement-the-frechet-inception-distance-fid-from-scratch/.

[3] TensorFlow, "Convolutional variational autoencoder." https://www.tensorflow.org/tutorials/generative/cvae.

[4] D. Hafner, "Building variational auto-encoders in tensorflow." https://danijar.com/building-variational-auto-encoders-in-tensorflow/.

[5] J. Brownlee, "How to develop a gan for generating mnist handwritten digits." https://machinelearningmastery.com/how-to-develop-a-generative-adversarial-network-for-an-mnist-handwritten-digits-f