

CAP6610 Project Report 1

Justin Ho
justinho@ufl.edu

February 14, 2023

1 Introduction

I have worked with some GANs before but never VAEs, so I decided to start simple and to implement both models on the MNIST number dataset. I downloaded TensorFlow on my computer and watched some videos and read documentation online on how to set up both models. One issue that I ran into was that even with such a simple dataset, my computer struggled to reasonably run this so I had to port my code over to Google Colab. After porting it over to TensorFlow, both models trained much quicker and once I got both running, I began to design some experiments to compare and contrast the two approaches two generation. My environment on Google Colab is Python 3.8 with TensorFlow version 2.11.0.

2 Experiment Design

Based on the research I have seen, generative models are tricky to evaluate [1]. There exist qualitative approaches that utilize human evaluators which are expensive and requires a great deal of man power. Typically in this paradigm, an evaluator is given a selection of different generated images and they must rate which one looks best out of the selection [2]. On the other hand you have quantitative techniques such as Frechet Inception Distance and Inception Score which are metrics that utilize a pre-trained InceptionV3 trained on generated images to provide a quantitative score on how good a generative models output images are [3]. A weakness of this technique is that the InceptionV3 backbone that powers both metrics was trained on color images and the MNIST number dataset only possesses a single color channel meaning results obtained from InceptionV3 will not be particularly accurate.

With these constraints in mind, I decided to utilize a more simplistic experiment design. For evaluation, I will be utilizing a combination of qualitative and applied quantitative approaches. In terms of qualitative analysis, the image generated by both models will be evaluated by me based on things such as noisiness and legibility. For applied quantitative approaches, I wanted to explore more practical quantitative metrics outside of accuracy, so for this experiment training time per epoch.

Now that evaluation techniques are defined, we can explore what variables we are going to be manipulating between GANs and VAEs. The variable that is going to be manipulated by my experimentation is the use of transposed convolution versus upsampling convolutions. An influential paper by Odena and et al noted how transposed convolutions are actually not particularly ideal in the generative portions of both models as they can result in excessive noise in certain situations [4]. The authors of this paper propose the use of an upscale convolution operator which gets around this problem altogether. Most guides online utilize the transposed convolution so I want to evaluate how the upsampling convolution compares to this common technique. All models within the same class of GAN or VAE will be trained on the same amount of epochs: 50 for GANs and 10 for VAEs. You can find the models and code attached in the appendix.

3 Analysis

The results of this experiment followed what previous literature has noted. Starting with the GANs, the images created by the GAN that utilized upsampling convolution (Fig 2) were significantly more legible and less noisy than the images produced by the traditional transposed convolution (Fig 1). This can be attributed to the fact that the filter size of 5 used is not divisible by the stride length 2. The paper by Odena and et al notes that one of the techniques that can be utilized to mitigate noisiness in GAN images is ensuring that these two values are divisible so this follows previous literature. In terms of quantitative metrics, the upsampling convolution GAN had a longer training time than the traditional transposed convolution GAN as the transposed convolution typically took 11 sec per epoch while the upsampling convolution GAN took 12-13 sec.

In regards to VAE, the images generated by the transposed convolution VAE (Fig 3) looked more legible to me than the images generated by the VAE with upsampling convolutions (Fig 4). It's hard to see but examining things such as the 0 in the third column first row shows additional noise in the upsampling VAE. This was surprising to me initially but then I realized that this made sense because the images generated by the VAE were less sharp than those of the GAN and because utilizing upsampling convolutions suppresses sharpness, the resulting image of the VAE with upsampling would be worse than with the GAN. In terms of training time, the VAE with upsampling convolutions was also slower than its peer with transposed convolutions (11 sec vs 10 sec).

4 Conclusion and Next Steps

Overall this was a fun first step into trying to understand the intricacies of GANs and VAEs. Experimenting with them by comparing two different upsampling strategies was tricky to get it to work but once I got it work it was rewarding seeing it generating images. In terms of next steps after my initial foray into GANs I am going to transition into exploring the conditional architectures of VAEs and GANs and the effect that upscaling and transposed convolutions have on these architectures.

References

- [1] A. Borji, Pros and Cons of GAN Evaluation Measures: New Developments, United States: Applied Soft Computing, 2017, CoRR, <https://arxiv.org/abs/2103.09396>
- [2] H. Alqahtani, et al., AN ANALYSIS OF EVALUATION METRICS OF GANS, 2019, International Conference on Information Technology and Applications (ICITA), <https://arxiv.org/abs/2103.09396>
- [3] M. Heusel, et al., GANs Trained by a Two Time-Scale Update Rule Converge to a Nash Equilibrium, CoRR, <http://arxiv.org/abs/1706.08500>
- [4] Odena, et al., "Deconvolution and Checkerboard Artifacts", Distill, 2016. <http://doi.org/10.23915/distill.00>

5 Appendix

Code: https://colab.research.google.com/drive/1sQISsb_m0PblZQ2zaQaWinpETmPiImBa?usp=sharing

Utilized the following starter code and modified for experimentation:

<https://www.tensorflow.org/tutorials/generative/cvae>

<https://www.tensorflow.org/tutorials/generative/dcgan>

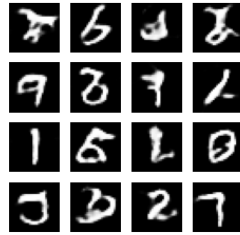


Figure 1: Results of transposed convolution GAN after 50 epochs



Figure 2: Results of upsampling convolution GAN after 50 epochs

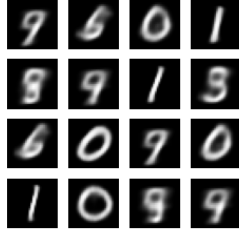


Figure 3: Results of transposed convolution VAE after 10 epochs

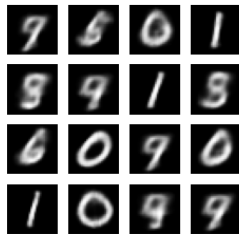


Figure 4: Results of upsampling convolution VAE after 10 epochs

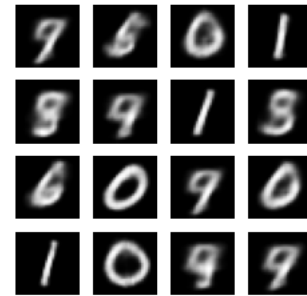
image_at_epoch_0050.png

image_at_epoch_C

...

GAN EXPERIMENTS

- Transposed Convolution vs Resize Convolution vs Different layer depth
 - Qualitative analysis
 - Appearance
 - Fidelity
 - weaknesses
 - Quantitative analysis
 - Model size
 - Training time
 - FID



TRANSPOSED CONVOLUTION GAN

```
import tensorflow as tf
```

```
!pip install imageio
!pip install git+https://github.com/tensorflow/docs
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheel
Requirement already satisfied: imageio in /usr/local/lib/python3.8/dist-packages (fr
Requirement already satisfied: numpy in /usr/local/lib/python3.8/dist-packages (fr
Requirement already satisfied: pillow in /usr/local/lib/python3.8/dist-packages (fr
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheel
Collecting git+https://github.com/tensorflow/docs
  Cloning https://github.com/tensorflow/docs to /tmp/pip-req-build-noffjh0y
  Running command git clone --filter=blob:none --quiet https://github.com/tensorf
  Resolved https://github.com/tensorflow/docs to commit 51cce0203d63843dc5c625928f
  Preparing metadata (setup.py) ... done
Requirement already satisfied: astor in /usr/local/lib/python3.8/dist-packages (fr
Requirement already satisfied: absl-py in /usr/local/lib/python3.8/dist-packages (
Requirement already satisfied: Jinja2 in /usr/local/lib/python3.8/dist-packages (f
Requirement already satisfied: nbformat in /usr/local/lib/python3.8/dist-packages
Requirement already satisfied: protobuf in /usr/local/lib/python3.8/dist-packages
Requirement already satisfied: pyyaml in /usr/local/lib/python3.8/dist-packages (f
Requirement already satisfied: MarkupSafe>=0.23 in /usr/local/lib/python3.8/dist-p
Requirement already satisfied: jsonschema>=2.6 in /usr/local/lib/python3.8/dist-p
Requirement already satisfied: jupyter-core in /usr/local/lib/python3.8/dist-pack
Requirement already satisfied: traitlets>=5.1 in /usr/local/lib/python3.8/dist-pa
Requirement already satisfied: fastjsonschema in /usr/local/lib/python3.8/dist-pa
Requirement already satisfied: importlib-resources>=1.4.0 in /usr/local/lib/pythor
Requirement already satisfied: attrs>=17.4.0 in /usr/local/lib/python3.8/dist-pa
Requirement already satisfied: pyrsistent!=0.17.0,!0.17.1,!0.17.2,>=0.14.0 in /
Requirement already satisfied: platformdirs>=2.5 in /usr/local/lib/python3.8/dist-
Requirement already satisfied: zipp>=3.1.0 in /usr/local/lib/python3.8/dist-packa
Building wheels for collected packages: tensorflow-docs
  Building wheel for tensorflow-docs (setup.py) ... done
  Created wheel for tensorflow-docs: filename=tensorflow_docs-0.0.0.dev0-py3-none-
  Stored in directory: /tmp/pip-ephem-wheel-cache-_6xpei7f/wheels/3b/ee/a2/ab4d36
Successfully built tensorflow-docs
Installing collected packages: tensorflow-docs
Successfully installed tensorflow-docs-0.0.0.dev0
```

```
import glob
import imageio
import matplotlib.pyplot as plt
import numpy as np
import os
import PIL
from tensorflow.keras import layers
import time

from IPython import display

(train_images, train_labels), (_, _) = tf.keras.datasets.mnist.load_data
```

Downloading data from <https://storage.googleapis.com/tensorflow/tf-keras-datasets/11490434/11490434> [=====] - 0s 0us/step

```

train_images = train_images.reshape(train_images.shape[0], 28, 28, 1).astype('float32')
train_images = (train_images - 127.5) / 127.5 # Normalize the images to [-1, 1]

BUFFER_SIZE = 60000
BATCH_SIZE = 256

# Batch and shuffle the data
train_dataset = tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)

def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((7, 7, 256)))
    assert model.output_shape == (None, 7, 7, 256) # Note: None is the batch size

    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False))
    assert model.output_shape == (None, 7, 7, 128)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    assert model.output_shape == (None, 14, 14, 64)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

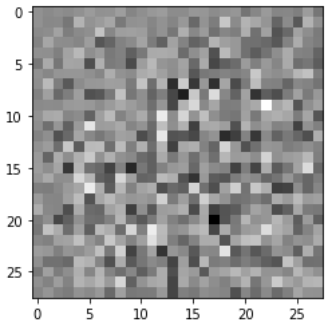
    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    assert model.output_shape == (None, 28, 28, 1)

    return model

generator = make_generator_model()

noise = tf.random.normal([1, 100])
generated_image = generator(noise, training=False)

plt.imshow(generated_image[0, :, :, 0], cmap='gray')

<matplotlib.image.AxesImage at 0x7f6449421c10>

def make_discriminator_model():
    model = tf.keras.Sequential()
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same', input_shape=[28, 28, 1]))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Flatten())
    model.add(layers.Dense(1))

```

```

return model

discriminator = make_discriminator_model
decision = discriminator(generated_image)
print (decision)

tf.Tensor([[0.00127015]], shape=(1, 1), dtype=float32)

# This method returns a helper function to compute cross entropy loss
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)

def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss

def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)

generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)

checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
                                  discriminator_optimizer=discriminator_optimizer,
                                  generator=generator,
                                  discriminator=discriminator)

EPOCHS = 50
noise_dim = 100
num_examples_to_generate = 16

# You will reuse this seed overtime (so it's easier)
# to visualize progress in the animated GIF
seed = tf.random.normal([num_examples_to_generate, noise_dim])

# Notice the use of `tf.function`
# This annotation causes the function to be "compiled".
@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)

        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)

    gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_variables)

    generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator.trainable_variables))

def generate_and_save_images(model, epoch, test_input):
    # Notice `training` is set to False.
    # This is so all layers run in inference mode (batchnorm).
    predictions = model(test_input, training=False)

    fig = plt.figure(figsize=(4, 4))

    for i in range(predictions.shape[0]):
        plt.subplot(4, 4, i+1)
        plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5, cmap='gray')
        plt.axis('off')

```



```

plt.savefig('image_at_epoch_{:04d}.png'.format(epoch))
plt.show()

def train(dataset, epochs):
    for epoch in range(epochs):
        start = time.time()

        for image_batch in dataset:
            train_step(image_batch)

        # Produce images for the GIF as you go
        display.clear_output(wait=True)
        generate_and_save_images(generator,
                                epoch + 1,
                                seed)

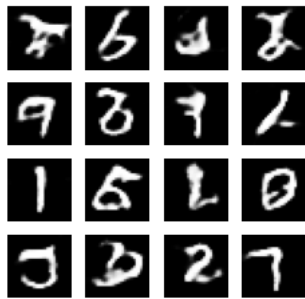
        # Save the model every 15 epochs
        if (epoch + 1) % 15 == 0:
            checkpoint.save(file_prefix = checkpoint_prefix)

        print ('Time for epoch {} is {} sec'.format(epoch + 1, time.time()-start))

    # Generate after the final epoch
    display.clear_output(wait=True)
    generate_and_save_images(generator,
                            epochs,
                            seed)

train(train_dataset, EPOCH

```



▼ UPSAMPLE CONVOLUTION GAN

```

def make_generator_upsize_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((7, 7, 256)))
    assert model.output_shape == (None, 7, 7, 256) # Note: None is the batch size

    model.add(layers.UpSampling2D(size=(2, 2)))
    model.add(layers.Conv2D(64, (3, 3), strides=(1, 1), padding='same', use_bias=False))
    assert model.output_shape == (None, 14, 14, 64)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.UpSampling2D(size=(2, 2)))
    model.add(layers.Conv2D(1, (3, 3), strides=(1, 1), padding='same', use_bias=False))
    assert model.output_shape == (None, 28, 28, 1)

    return model

def make_discriminator_model():
    model = tf.keras.Sequential()
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
                            input_shape=[28, 28, 1]))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

```

```

model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
model.add(layers.LeakyReLU())
model.add(layers.Dropout(0.3))

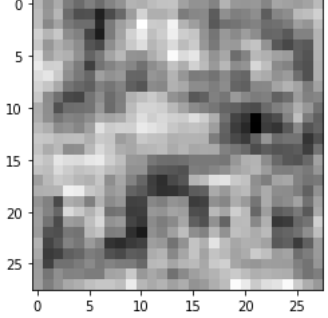
model.add(layers.Flatten())
model.add(layers.Dense(1))

return model

discriminator = make_discriminator_model()
generator = make_generator_upsize_model()

noise = tf.random.normal([1, 100])
generated_image = generator(noise, training=False)

plt.imshow(generated_image[0, :, :, 0], cmap='gray')

<matplotlib.image.AxesImage at 0x7f636c3d64f0>

# Notice the use of `tf.function`
# This annotation causes the function to be "compiled".
@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)

        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)

    gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_variables)

    generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator.trainable_variables))

def generate_and_save_images(model, epoch, test_input):
    # Notice `training` is set to False.
    # This is so all layers run in inference mode (batchnorm).
    predictions = model(test_input, training=False)

    fig = plt.figure(figsize=(4, 4))

    for i in range(predictions.shape[0]):
        plt.subplot(4, 4, i+1)
        plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5, cmap='gray')
        plt.axis('off')

    plt.savefig('image_at_epoch_{:04d}_UC.png'.format(epoch))
    plt.show()

generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)

def train(dataset, epochs):
    for epoch in range(epochs):

```

```

start = time.time()

for image_batch in dataset:
    train_step(image_batch)

# Produce images for the GIF as you go
display.clear_output(wait=True)
generate_and_save_images(generator,
                          epoch + 1,
                          seed)

# Save the model every 15 epochs
if (epoch + 1) % 15 == 0:
    checkpoint.save(file_prefix = checkpoint_prefix)

print ('Time for epoch {} is {} sec'.format(epoch + 1, time.time()-start))

# Generate after the final epoch
display.clear_output(wait=True)
generate_and_save_images(generator,
                          epochs,
                          seed)

train(train_dataset, EPOCHS)

```



▼ TRANPOSED CONVOLUTION VAE

```

from IPython import display

import glob
import imageio
import matplotlib.pyplot as plt
import numpy as np
import PIL
import tensorflow as tf
import tensorflow_probability as tfp
import time

(train_images, _), (test_images, _) = tf.keras.datasets.mnist.load_data()

def preprocess_images(images):
    images = images.reshape((images.shape[0], 28, 28, 1)) / 255.
    return np.where(images > .5, 1.0, 0.0).astype('float32')

train_images = preprocess_images(train_images)
test_images = preprocess_images(test_images)

train_size = 60000
batch_size = 32
test_size = 10000

train_dataset = (tf.data.Dataset.from_tensor_slices(train_images)
                  .shuffle(train_size).batch(batch_size))
test_dataset = (tf.data.Dataset.from_tensor_slices(test_images)
                 .shuffle(test_size).batch(batch_size))

```

```

class CVAE(tf.keras.Model):
    """Convolutional variational autoencoder."""

    def __init__(self, latent_dim):
        super(CVAE, self).__init__()
        self.latent_dim = latent_dim
        self.encoder = tf.keras.Sequential(
            [
                tf.keras.layers.InputLayer(input_shape=(28, 28, 1)),
                tf.keras.layers.Conv2D(
                    filters=32, kernel_size=3, strides=(2, 2), activation='relu'),
                tf.keras.layers.Conv2D(
                    filters=64, kernel_size=3, strides=(2, 2), activation='relu'),
                tf.keras.layers.Flatten(),
                # No activation
                tf.keras.layers.Dense(latent_dim + latent_dim),
            ]
        )

        self.decoder = tf.keras.Sequential(
            [
                tf.keras.layers.InputLayer(input_shape=(latent_dim,)),
                tf.keras.layers.Dense(units=7*7*32, activation=tf.nn.relu),
                tf.keras.layers.Reshape(target_shape=(7, 7, 32)),
                tf.keras.layers.Conv2DTranspose(
                    filters=64, kernel_size=3, strides=2, padding='same',
                    activation='relu'),
                tf.keras.layers.Conv2DTranspose(
                    filters=32, kernel_size=3, strides=2, padding='same',
                    activation='relu'),
                # No activation
                tf.keras.layers.Conv2DTranspose(
                    filters=1, kernel_size=3, strides=1, padding='same'),
            ]
        )

    @tf.function
    def sample(self, eps=None):
        if eps is None:
            eps = tf.random.normal(shape=(100, self.latent_dim))
        return self.decode(eps, apply_sigmoid=True)

    def encode(self, x):
        mean, logvar = tf.split(self.encoder(x), num_or_size_splits=2, axis=1)
        return mean, logvar

    def reparameterize(self, mean, logvar):
        eps = tf.random.normal(shape=mean.shape)
        return eps * tf.exp(logvar * .5) + mean

    def decode(self, z, apply_sigmoid=False):
        logits = self.decoder(z)
        if apply_sigmoid:
            probs = tf.sigmoid(logits)
            return probs
        return logits

    optimizer = tf.keras.optimizers.Adam(1e-4)

    def log_normal_pdf(sample, mean, logvar, raxis=1):
        log2pi = tf.math.log(2. * np.pi)
        return tf.reduce_sum(
            -.5 * ((sample - mean) ** 2. * tf.exp(-logvar) + logvar + log2pi),
            axis=raxis)

    def compute_loss(model, x):
        mean, logvar = model.encode(x)
        z = model.reparameterize(mean, logvar)
        x_logit = model.decoder(z)
        cross_ent = tf.nn.sigmoid_cross_entropy_with_logits(logits=x_logit, labels=x)
        logpx_z = -tf.reduce_sum(cross_ent, axis=[1, 2, 3])
        logpz = log_normal_pdf(z, 0., 0.)
        logqz_x = log_normal_pdf(z, mean, logvar)
        return -tf.reduce_mean(logpx_z + logpz - logqz_x)

```

```

@tf.function
def train_step(model, x, optimizer):
    """Executes one training step and returns the loss.

    This function computes the loss and gradients, and uses the latter to
    update the model's parameters.
    """
    with tf.GradientTape() as tape:
        loss = compute_loss(model, x)
        gradients = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients, model.trainable_variables))

epochs = 10
# set the dimensionality of the latent space to a plane for visualization later
latent_dim = 2
num_examples_to_generate = 16

# keeping the random vector constant for generation (prediction) so
# it will be easier to see the improvement.
random_vector_for_generation = tf.random.normal(
    shape=[num_examples_to_generate, latent_dim])
model = CVAE(latent_dim)

def generate_and_save_images(model, epoch, test_sample):
    mean, logvar = model.encode(test_sample)
    z = model.reparameterize(mean, logvar)
    predictions = model.sample(z)
    fig = plt.figure(figsize=(4, 4))

    for i in range(predictions.shape[0]):
        plt.subplot(4, 4, i + 1)
        plt.imshow(predictions[i, :, :, 0], cmap='gray')
        plt.axis('off')

    # tight_layout minimizes the overlap between 2 sub-plots
    plt.savefig('image_at_epoch_{:04d}_vae.png'.format(epoch))
    plt.show()

# Pick a sample of the test set for generating output images
assert batch_size >= num_examples_to_generate
for test_batch in test_dataset.take(1):
    test_sample = test_batch[0:num_examples_to_generate, :, :, :]

generate_and_save_images(model, 0, test_sample)

for epoch in range(1, epochs + 1):
    start_time = time.time()
    for train_x in train_dataset:
        train_step(model, train_x, optimizer)
    end_time = time.time()

    loss = tf.keras.metrics.Mean()
    for test_x in test_dataset:
        loss(compute_loss(model, test_x))
    elbo = -loss.result()
    display.clear_output(wait=False)
    print('Epoch: {}, Test set ELBO: {}, time elapse for current epoch: {}'.
          .format(epoch, elbo, end_time - start_time))
    generate_and_save_images(model, epoch, test_sample)

```

```
Epoch: 10 Test set FID: 155.78416447971304 time elapsed for current epoch: 6.41
def display_image(epoch_no):
    return PIL.Image.open('image_at_epoch_{:04d}.png'.format(epoch_no))
```



▼ UPSAMPLING CONVOLUTION VAE



```
def generate_and_save_images(model, epoch, test_sample):
    mean, logvar = model.encode(test_sample)
    z = model.reparameterize(mean, logvar)
    predictions = model.sample(z)
    fig = plt.figure(figsize=(4, 4))

    for i in range(predictions.shape[0]):
        plt.subplot(4, 4, i + 1)
        plt.imshow(predictions[i, :, :, 0], cmap='gray')
        plt.axis('off')

    # tight_layout minimizes the overlap between 2 sub-plots
    plt.savefig('image_at_epoch_{:04d}_vae_uc.png'.format(epoch))
    plt.show()

class CVAE(tf.keras.Model):
    """Convolutional variational autoencoder."""

    def __init__(self, latent_dim):
        super(CVAE, self).__init__()
        self.latent_dim = latent_dim
        self.encoder = tf.keras.Sequential(
            [
                tf.keras.layers.InputLayer(input_shape=(28, 28, 1)),
                tf.keras.layers.Conv2D(
                    filters=32, kernel_size=3, strides=(2, 2), activation='relu'),
                tf.keras.layers.Conv2D(
                    filters=64, kernel_size=3, strides=(2, 2), activation='relu'),
                tf.keras.layers.Flatten(),
                # No activation
                tf.keras.layers.Dense(latent_dim + latent_dim),
            ]
        )

        self.decoder = tf.keras.Sequential(
            [
                tf.keras.layers.InputLayer(input_shape=(latent_dim,)),
                tf.keras.layers.Dense(units=7*7*32, activation=tf.nn.relu),
                tf.keras.layers.Reshape(target_shape=(7, 7, 32)),
                tf.keras.layers.UpSampling2D(size=(2, 2)),
                tf.keras.layers.Conv2D(64, (3, 3), strides=(1, 1), padding='same', activation='relu'),
                tf.keras.layers.UpSampling2D(size=(2, 2)),
                tf.keras.layers.Conv2D(32, (3, 3), strides=(1, 1), padding='same', activation='relu'),
                # No activation
                tf.keras.layers.Conv2D(1, (3, 3), strides=(1, 1), padding='same', use_bias=False),
            ]
        )

    @tf.function
    def sample(self, eps=None):
        if eps is None:
            eps = tf.random.normal(shape=(100, self.latent_dim))
        return self.decode(eps, apply_sigmoid=True)

    def encode(self, x):
        mean, logvar = tf.split(self.encoder(x), num_or_size_splits=2, axis=1)
        return mean, logvar

    def reparameterize(self, mean, logvar):
        eps = tf.random.normal(shape=mean.shape)
        return eps * tf.exp(logvar * .5) + mean

    def decode(self, z, apply_sigmoid=False):
        logits = self.decoder(z)
        if apply_sigmoid:
            probs = tf.sigmoid(logits)
```

```

        return probs
    return logits

optimizer = tf.keras.optimizers.Adam(1e-4)

def log_normal_pdf(sample, mean, logvar, raxis=1):
    log2pi = tf.math.log(2. * np.pi)
    return tf.reduce_sum(
        -.5 * ((sample - mean) ** 2. * tf.exp(-logvar) + logvar + log2pi),
        axis=raxis)

def compute_loss(model, x):
    mean, logvar = model.encode(x)
    z = model.reparameterize(mean, logvar)
    x_logit = model.decode(z)
    cross_ent = tf.nn.sigmoid_cross_entropy_with_logits(logits=x_logit, labels=x)
    logpx_z = -tf.reduce_sum(cross_ent, axis=[1, 2, 3])
    logpz = log_normal_pdf(z, 0., 0.)
    logqz_x = log_normal_pdf(z, mean, logvar)
    return -tf.reduce_mean(logpx_z + logpz - logqz_x)

@tf.function
def train_step(model, x, optimizer):
    """Executes one training step and returns the loss.

    This function computes the loss and gradients, and uses the latter to
    update the model's parameters.
    """
    with tf.GradientTape() as tape:
        loss = compute_loss(model, x)
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))

epochs = 10
# set the dimensionality of the latent space to a plane for visualization later
latent_dim = 2
num_examples_to_generate = 16

# keeping the random vector constant for generation (prediction) so
# it will be easier to see the improvement.
random_vector_for_generation = tf.random.normal(
    shape=[num_examples_to_generate, latent_dim])
model = CVAE(latent_dim)

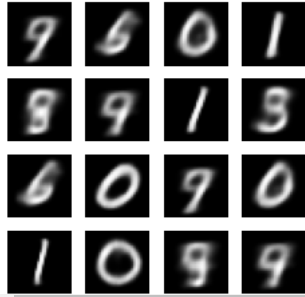
generate_and_save_images(model, 0, test_sample)

for epoch in range(1, epochs + 1):
    start_time = time.time()
    for train_x in train_dataset:
        train_step(model, train_x, optimizer)
    end_time = time.time()

    loss = tf.keras.metrics.Mean()
    for test_x in test_dataset:
        loss(compute_loss(model, test_x))
    elbo = -loss.result()
    display.clear_output(wait=False)
    print('Epoch: {}, Test set ELBO: {}, time elapse for current epoch: {}'.
          .format(epoch, elbo, end_time - start_time))
    generate_and_save_images(model, epoch, test_sample)

```

Epoch: 10, Test set ELBO: -155.81797790527344, time elapse for current epoch: 7.16



✓ 2m 2s completed at 10:59 PM

