CAP6610 Project Report 2

Justin Ho
justinho@ufl.edu

February 28, 2023

1 Introduction

As previously mentioned, after my initial foray into the starter DCGAN and DCVAE that every guide uses as an introduction to generative models, this week I decided to challenge myself further and dive into the realm of the conditional class of GANS. One weakness as mentioned in class with the original GAN and VAE formulation is that they do not provide you with the ability to select what you want to generate. Typically you pass in the latent vector for the GAN or VAE and whatever image you get from the generator/decoder is what you get. However, the conditional architecture for both models seeks to change this design flaw by introducing some clever techniques to make image generation controllable

2 Conditioning the training process for the CGAN and the CVAE

The Conditional GAN (CGAN) [1] is a GAN that allows the user to generate any image they want that the generator was trained to produce as long as a class label and a latent vector is passed in. This is possible due to a change made to the GAN's training loop. Instead of just passing in a set of images to the discriminator, auxiliary information typically in the form of a class label is passed in as well. The class label is also passed into the generator as well which allows both to be "conditioned" to the label and overall allows the generator to associate a particular type of image to be generated with a label allowing the user to be able to control the resultant image of the generator.

The Conditional VAE (CVAE) [2] like the CGAN is a VAE that also allows the user to control the output of the decoder model. The edits required to train a CGAN is almost identical to that of a CGAN. Auxiliary information is passed in the form of a class label to both the encoder and decoder which allows the CVAE to learn to generate a particular image when it receives a certain label.

3 Implementation of the CGAN and the CVAE

Many modern examples that I found of the CGAN are typically run on the MNIST dataset. This dataset is simple and allows for a pretty quick training process. However, because I worked with the MNIST dataset in the last report I wanted to challenge myself and try to implement both on the CIFAR-10 dataset which is a much more complex and noisy dataset. I started with CGAN code designed initially for the MNIST dataset [3], and augmented it to support the CIFAR-10 input images. Porting the MNIST CGAN code to support the CIFAR-10 dataset was a long process since I had to modify some logic in the train loop to work which took some tinkering. Also even when the train loop did work, finding the right model structure was time consuming since it involved a lot of time expensive experimentation to get the image to look good. For my implementation of the CIFAR-10 CGAN I decided to utilize the upscaling convolution block brought up in the last report as technique to improve image generation quality.

For the CVAE most of the examples are written in PyTorch for the MNIST dataset and the ones that are written in Keras are outdated and are written using old TensorFlow 1 conventions. I started the process of implementing a modern TensorFlow 2 Keras CVAE from a normal VAE [4] but this is a pretty time expensive process that I couldn't complete for this report.

4 Analysis

Overall, the images produced by the CGAN was of pretty high quality considering the complexity of the CIFAR-10 dataset (Figure 1). The only flaws that I noticed from the images was that sometimes the images had some visual artifacts which could be due to a bug or mistake I made in the network architecture. The ability to select class works well and you can generate multiple images from the same class to get an overall feel of what the generator has captured in regards to the class.

5 Conclusion and Next Steps

Overall this was a really informative week for me. As I've mentioned, before I've never really had to work with custom training loops before so this was a new experience. Tinkering with the training loop and the process of editing it to work was intimidating but also super enlightening. For the next report I am planning on hopefully finishing my implementation of the TensorFlow 2 CVAE on the CIFAR-10 dataset which I am excited about because there are not examples that I could find. If I have time I also am planning on experimenting with the other types of GANs and VAEs out there.

References

[1] https://arxiv.org/abs/1411.1784

[2] https://proceedings.neurips.cc/paper/2015/hash/8d55a249e6baa5c06772297520da2051-Abstract.html

[3] https://keras.io/examples/generative/conditional_gan/

[4] https://www.tensorflow.org/tutorials/generative/cvae

6 Appendix

Code: https://drive.google.com/file/d/1j6YphHMIW7wKmZA26hrunsTzIkBYz6Yc/view?usp=sharing

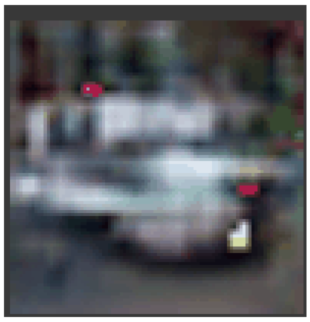

Figure 1: Results of the CGAN generator when passed in the class label for a car and some noise. Note the visual artifacts

Conditional GAN on CIFAR-10

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

```
      Preparing metadata (setup.py) ... d
```

```
from tensorflow import keras
from tensorflow.keras import layers

from tensorflow_docs.vis import embed
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import imageio
```

```
batch_size = 100
num_channels = 3
num_classes = 10
image_size = 32
latent_dim = 110
```

```
# We'll use all the available examples from both the training and test
# sets.
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
all_images = np.concatenate([x_train, x_test])
all_labels = np.concatenate([y_train, y_test])

# Scale the pixel values to [0, 1] range, add a channel dimension to
# the images, and one-hot encode the labels.
all_images = all_images.astype("float32") / 255.0
all_labels = keras.utils.to_categorical(all_labels, 10)

# Create tf.data.Dataset.
dataset = tf.data.Dataset.from_tensor_slices((all_images, all_labels))
dataset = dataset.shuffle(buffer_size=1024).batch(batch_size)

print(f"Shape of training images: {all_images.shape}")
print(f"Shape of training labels: {all_labels.shape}")
```

```
     Shape of training images: (60000, :   32, 3)
     Shape of training labels: (60000, 10)
```

```
generator_in_channels = latent_dim + num_classes
discriminator_in_channels = num_channels + num_classes
print(generator_in_channels, discriminator_in_channels)
```

```
     120
```

```
# Create the discriminator.
discriminator = keras.Sequential(
    [
        keras.layers.InputLayer((32, 32, discriminator_in_channels)),

        layers.Conv2D(16, (3, 3), strides=(2, 2), padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Dropout(0.5),

        layers.Conv2D(32, (3, 3), strides=(1, 1), padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Dropout(0.5),
        layers.BatchNormalization(),

        layers.Conv2D(64, (3, 3), strides=(2, 2), padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Dropout(0.5),
        layers.BatchNormalization(),

        layers.Conv2D(128, (3, 3), strides=(1, 1), padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Dropout(0.5),
        layers.BatchNormalization(),
        layers.BatchNormalization(),

        layers.Conv2D(256, (3, 3), strides=(2, 2), padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Dropout(0.5),
        layers.BatchNormalization(),

        layers.Conv2D(512, (3, 3), strides=(1, 1), padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Dropout(0.5),
        layers.BatchNormalization(),
        layers.Flatten(),

        layers.Dense(1),
    ],
    name="discriminator",
)

# Create the generator.
generator = keras.Sequential(
    [
        keras.layers.InputLayer((generator_in_channels,)),
        # We want to generate 128 + num_classes coefficients to reshape into a
        # 7x7x(128 + num_classes) map.
        layers.Dense(4 * 4 * generator_in_channels * 2, activation="relu", use_bias=False),


        layers.Reshape((4, 4, generator_in_channels * 2)),

        layers.UpSampling2D(size=(2, 2)),
        layers.Conv2D(192, (4, 4), strides=(1, 1), padding="same", activation="relu"),
        layers.BatchNormalization(),

        layers.UpSampling2D(size=(2, 2)),
        layers.Conv2D(96, (4, 4), strides=(1, 1), padding="same", activation="relu"),
        layers.BatchNormalization(),

        layers.UpSampling2D(size=(2, 2)),
        layers.Conv2D(3, (4, 4), strides=(1, 1), padding="same", activation="tanh"),
    ],
    name="generator",
)


class ConditionalGAN(keras.Model):
    def __init__(self, discriminator, generator, latent_dim):
        super().__init__()
        self.discriminator = discriminator
        self.generator = generator
        self.latent_dim = latent_dim
        self.gen_loss_tracker = keras.metrics.Mean(name="generator_loss")
        self.disc_loss_tracker = keras.metrics.Mean(name="discriminator_loss")

    @property
    def metrics(self):
        return [self.gen_loss_tracker, self.disc_loss_tracker]

    def compile(self, d_optimizer, g_optimizer, loss_fn):
        super().compile()
        self.d_optimizer = d_optimizer
        self.g_optimizer = g_optimizer
        self.loss_fn = loss_fn

    def train_step(self, data):
        # Unpack the data.
        real_images, one_hot_labels = data

        # Add dummy dimensions to the labels so that they can be concatenated with
        # the images. This is for the discriminator.
        image_one_hot_labels = one_hot_labels[:, :, None, None]
        image_one_hot_labels = tf.repeat(
            image_one_hot_labels, repeats=[image_size * image_size]
        )
        image_one_hot_labels = tf.reshape(
            image_one_hot_labels, (-1, image_size, image_size, num_classes)
        )

        # Sample random points in the latent space and concatenate the labels.
        # This is for the generator.
        batch_size = tf.shape(real_images)[0]
        random_latent_vectors = tf.random.normal(shape=(batch_size, self.latent_dim))
        random_vector_labels = tf.concat(
            [random_latent_vectors, one_hot_labels], axis=1
        )

        # Decode the noise (guided by labels) to fake images.
        generated_images = self.generator(random_vector_labels)

        # Combine them with real images. Note that we are concatenating the labels
        # with these images here.
        fake_image_and_labels = tf.concat([generated_images, image_one_hot_labels], -1)
        real_image_and_labels = tf.concat([real_images, image_one_hot_labels], -1)
        combined_images = tf.concat(
            [fake_image_and_labels, real_image_and_labels], axis=0
        )

        # Assemble labels discriminating real from fake images.
        labels = tf.concat(
            [tf.ones((batch_size, 1)), tf.zeros((batch_size, 1))], axis=0
        )

        # Train the discriminator.
        with tf.GradientTape() as tape:
            predictions = self.discriminator(combined_images)
            d_loss = self.loss_fn(labels, predictions)
        grads = tape.gradient(d_loss, self.discriminator.trainable_weights)
        self.d_optimizer.apply_gradients(
            zip(grads, self.discriminator.trainable_weights)
        )

        # Sample random points in the latent space.
        random_latent_vectors = tf.random.normal(shape=(batch_size, self.latent_dim))
        random_vector_labels = tf.concat(
            [random_latent_vectors, one_hot_labels], axis=1
        )

        # Assemble labels that say "all real images".
        misleading_labels = tf.zeros((batch_size, 1))

        # Train the generator (note that we should *not* update the weights
        # of the discriminator)!
        with tf.GradientTape() as tape:
            fake_images = self.generator(random_vector_labels)
            fake_image_and_labels = tf.concat([fake_images, image_one_hot_labels], -1)
            predictions = self.discriminator(fake_image_and_labels)
            g_loss = self.loss_fn(misleading_labels, predictions)
        grads = tape.gradient(g_loss, self.generator.trainable_weights)
        self.g_optimizer.apply_gradients(zip(grads, self.generator.trainable_weights))

        # Monitor loss.
        self.gen_loss_tracker.update_state(g_loss)
        self.disc_loss_tracker.update_state(d_loss)
        return {
            "g_loss": self.gen_loss_tracker.result(),
            "d_loss": self.disc_loss_tracker.result(),
        }

cond_gan = ConditionalGAN(
    discriminator=discriminator, generator=generator, latent_dim=latent_dim
)
cond_gan.compile(
    d_optimizer=keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5),
    g_optimizer=keras.optimizers.Adam(learning_rate=0.0002, beta_1=0.5),
    loss_fn=keras.losses.BinaryCrossentropy(from_logits=True),
)

cond_gan.fit(dataset, epochs=20)
```

```
Epoch 1/20
600/600 [==============================] - 59s 68ms/step - g_loss: 0.8155 - d_loss: 0.6832
Epoch 2/20
600/600 [==============================] - 41s 68ms/step - g_loss: 0.8353 - d_loss: 0.6700
Epoch 3/20
600/600 [==============================] - 40s 67ms/step - g_loss: 0.8291 - d_loss: 0.6678
Epoch 4/20
```

```
600/600 [==============================] - 40s 67ms/step - g_loss: 0.8222 - d_loss: 0.6715
Epoch 5/20
600/600 [==============================] - 40s 67ms/step - g_loss: 0.8525 - d_loss: 0.6649
Epoch 6/20
600/600 [==============================] - 40s 67ms/step - g_loss: 0.8792 - d_loss: 0.6563
Epoch 7/20
600/600 [==============================] - 40s 67ms/step - g_loss: 0.8642 - d_loss: 0.6515
Epoch 8/20
600/600 [==============================] - 40s 67ms/step - g_loss: 0.8649 - d_loss: 0.6548
Epoch 9/20
600/600 [==============================] - 40s 67ms/step - g_loss: 0.8969 - d_loss: 0.6451
Epoch 10/20
600/600 [==============================] - 40s 67ms/step - g_loss: 0.9167 - d_loss: 0.6376
Epoch 11/20
600/600 [==============================] - 40s 67ms/step - g_loss: 0.9407 - d_loss: 0.6278
Epoch 12/20
600/600 [==============================] - 40s 67ms/step - g_loss: 0.9978 - d_loss: 0.6099
Epoch 13/20
600/600 [==============================] - 40s 67ms/step - g_loss: 1.0364 - d_loss: 0.5979
Epoch 14/20
600/600 [==============================] - 40s 67ms/step - g_loss: 1.0728 - d_loss: 0.5889
Epoch 15/20
600/600 [==============================] - 40s 67ms/step - g_loss: 1.1001 - d_loss: 0.5730
Epoch 16/20
600/600 [==============================] - 40s 67ms/step - g_loss: 1.1661 - d_loss: 0.5654
Epoch 17/20
600/600 [==============================] - 40s 67ms/step - g_loss: 1.2103 - d_loss: 0.5418
Epoch 18/20
600/600 [==============================] - 40s 67ms/step - g_loss: 1.2645 - d_loss: 0.5332
Epoch 19/20
600/600 [==============================] - 40s 67ms/step - g_loss: 1.3215 - d_loss: 0.5156
Epoch 20/20
600/600 [==============================] - 40s 67ms/step - g_loss: 1.3688 - d_loss: 0.5008
<keras.callbacks.History at 0x7f33dc54a5
```

```
# We first extract the trained generator from our Conditiona GAN.
trained_gen = cond_gan.generator

# Choose the number of intermediate images that would be generated in
# between the interpolation + 2 (start and last images).
num_interpolation = 20  # @param {type:"integer"}

# Sample noise for the interpolation.
interpolation_noise = tf.random.normal(shape=(1, latent_dim))
interpolation_noise = tf.repeat(interpolation_noise, repeats=num_interpolation)
interpolation_noise = tf.reshape(interpolation_noise, (num_interpolation, latent_dim))


def interpolate_class(first_number, second_number):
    # Convert the start and end labels to one-hot encoded vectors.
    first_label = keras.utils.to_categorical([first_number], num_classes)
    second_label = keras.utils.to_categorical([second_number], num_classes)
    first_label = tf.cast(first_label, tf.float32)
    second_label = tf.cast(second_label, tf.float32)

    # Calculate the interpolation vector between the two labels.
    percent_second_label = tf.linspace(0, 1, num_interpolation)[:, None]
    percent_second_label = tf.cast(percent_second_label, tf.float32)
    interpolation_labels = (
        first_label * (1 - percent_second_label) + second_label * percent_second_label
    )

    # Combine the noise and the labels and run inference with the generator.
    noise_and_labels = tf.concat([interpolation_noise, interpolation_labels], 1)
    fake = trained_gen.predict(noise_and_labels)
    return fake


start_class = 0  # @param {type:"slider", min:0, max:9, step:1}
end_class = 0  # @param {type:"slider", min:0, max:9, step:1}

fake_images = interpolate_class(start_class, end_class)
```

num_interpolation: 20

start_class: 0

end_class: 0

```
1/1 [==============================] - 0s 35ms/s
```

```
fake_images *= 255.0
converted_images = fake_images.astype(np.uint8)
converted_images = tf.image.resize(converted_images, (96, 96)).numpy().astype(np.uint8)
imageio.mimsave("animation.gif", converted_images, fps=1)
```

```
embed.embed_file("animation.gif")
```

```
!pip install tensorflow.examp

    Looking in indexes: https://pypi.org/simple, https://us-pyt  n.pkg.dev/colab-wheels/public/simple/
    ERROR: Could not find a version that satisfies the requirement tensorflow.examples (from versions: none)
    ERROR: No matching distribution found for tensorflow.examples
```

Conditional Variational Auto-Encoder

```
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


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=((32, 32, discriminator_in_channels))),

            tf.keras.layers.Conv2D(
                filters=32, kernel_size=4, strides=(2, 2), activation='relu'),
            tf.keras.layers.BatchNormalization(),
            tf.keras.layers.ReLU(),

            tf.keras.layers.Conv2D(
                filters=32, kernel_size=4, strides=(2, 2), activation='relu'),
            tf.keras.layers.BatchNormalization(),
            tf.keras.layers.ReLU(),

            tf.keras.layers.Conv2D(
                filters=32, kernel_size=4, strides=(2, 2), activation='relu'),
            tf.keras.layers.BatchNormalization(),
            tf.keras.layers.ReLU(),

            # No activation
            tf.keras.layers.Dense(latent_dim + latent_dim),
        ]
    )
    self.decoder = tf.keras.Sequential(
        [
            tf.keras.layers.InputLayer(input_shape=(generator_in_channels,)),
            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(192, (4, 4), strides=(1, 1), padding='same', activation='relu'),
            layers.BatchNormalization(),

            tf.keras.layers.UpSampling2D(size=(2, 2)),
            tf.keras.layers.Conv2D(96, (4, 4), strides=(1, 1), padding='same', activation='relu'),
            layers.BatchNormalization(),

            # No activation
            tf.keras.layers.Conv2D(3, (4, 4), 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, y, 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 = 110

model = CVAE(latent_dim)


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

  print('Epoch: {}, time elapse for current epoch: {}'
        .format(epoch, end_time - start_time))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-17-975fe8b20e25> in <module>
      2   start_time = time.time()
      3   for train_x in all_images:
----> 4     train_step(model, train_x, optimizer)
      5   end_time = time.time()
      6

                              1 frames
/usr/local/lib/python3.8/dist-packages/tensorflow/python/autograph/impl/api.py in
converted_call(f, args, kwargs, caller_fn_scope, options)
    437     try:
    438       if kwargs is not None:
--> 439         result = converted_f(*effective_args, **kwargs)
    440       else:
    441         result = converted_f(*effective_args)

TypeError: in user code:


    TypeError: tf__train_step() missing 1 required positional argument:
'optimizer'
```