Conformalized Generative Adversarial Network

May 15, 2023

Contents

1 Class Diagram

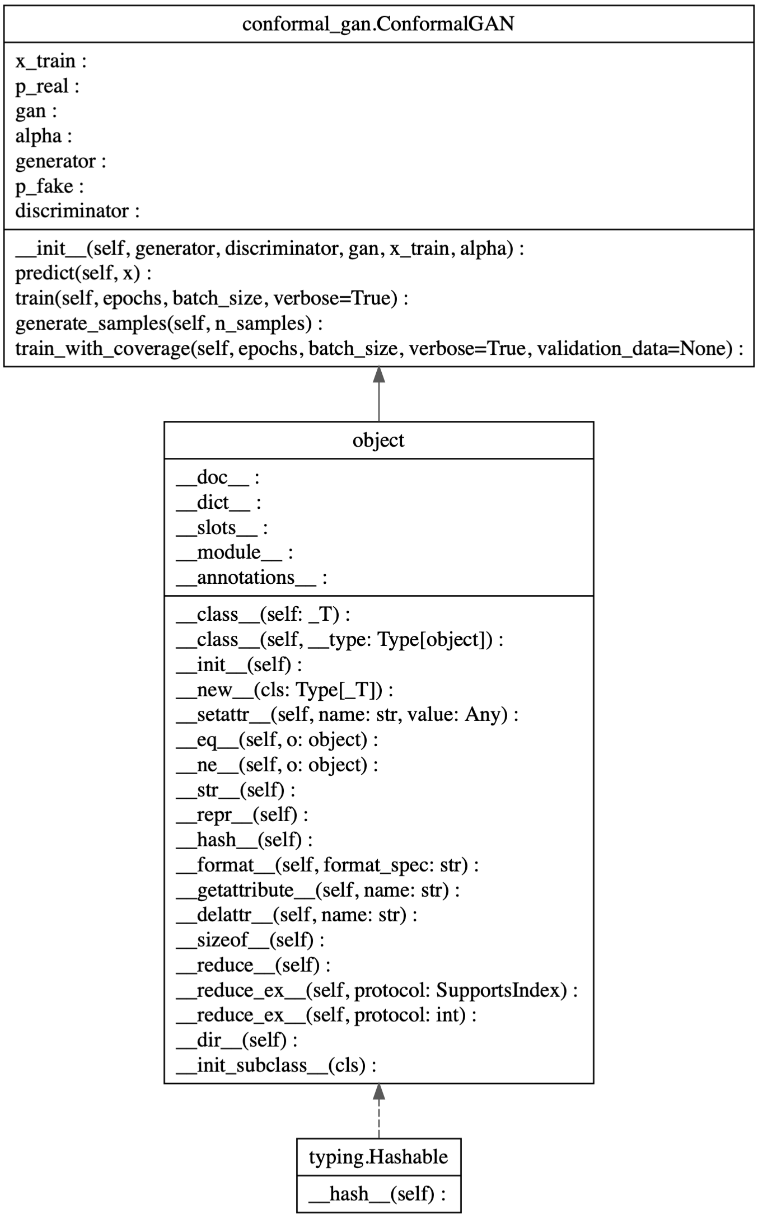

Figure 1: Class diagram for ConformalizedGAN

2 Sequence diagram

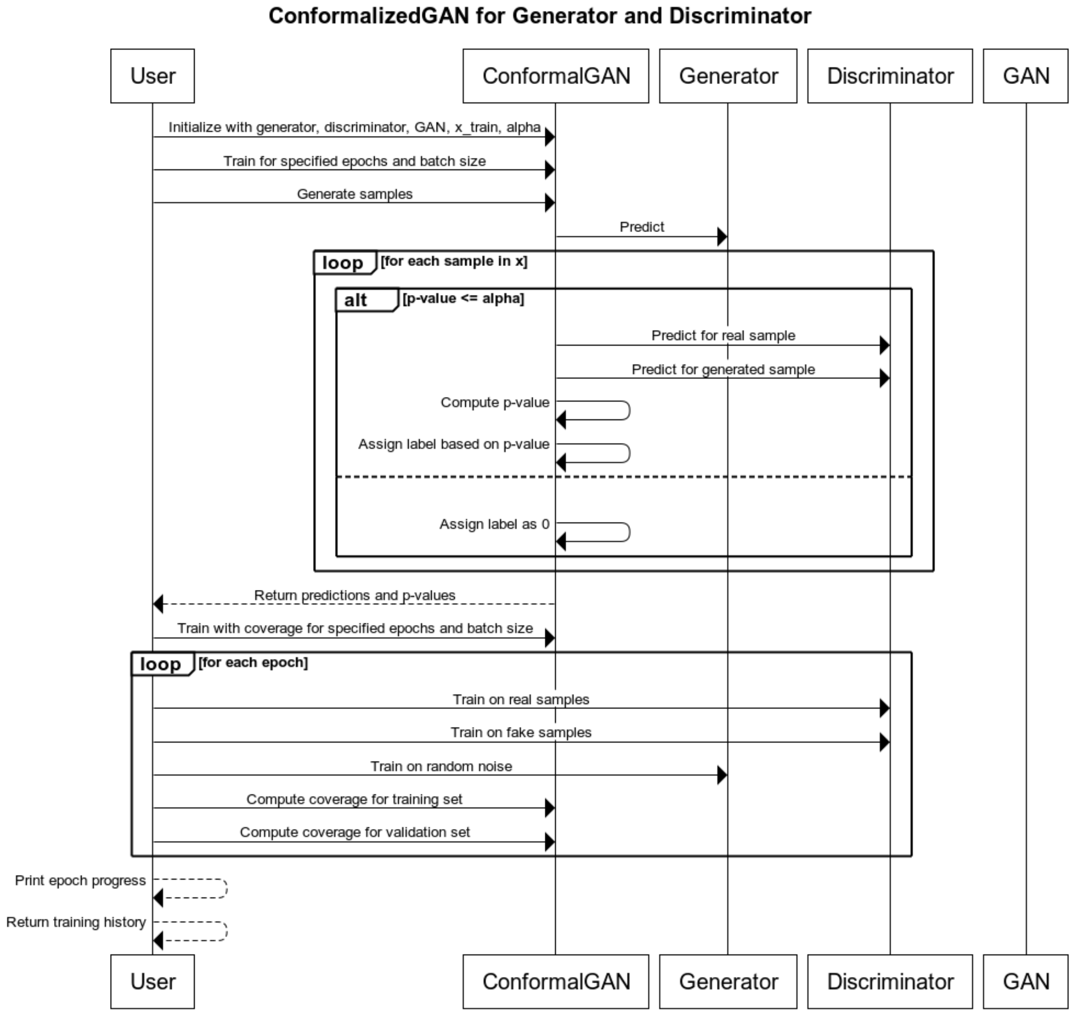

Figure 2: Sequence diagram for ConformalizedGAN

3 Source code

3.1 Conformal GAN

```python

import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense, Reshape, Flatten, Dropout, BatchNormalization, LSTM, LeakyReLU
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.optimizers import Adam

class ConformalGAN:
     def __init__(self, generator, discriminator, gan, x_train, alpha):
         self.generator = generator
         self.discriminator = discriminator
         self.gan = gan
         self.x_train = x_train
         self.alpha = alpha

     def predict(self, x):
         p_values = np.zeros((len(x), 2))
         y_pred = np.zeros((len(x),))
         for i, x_i in enumerate(x):
             # Generate a sample from the generator
             z = np.random.normal(size=(1, self.generator.input_shape[1]))
             x_gen = self.generator.predict(z)
             # Compute the p-value for the sample
             d_real = self.discriminator.predict(x_i.reshape(1, -1))[0]
             d_gen = self.discriminator.predict(x_gen.reshape(1, -1))[0]
             p_value = (np.sum(d_gen >= d_real) + 1) / (self.x_train.shape[0] + 1)
             p_values[i] = [1 - p_value, p_value]
             # Predict the label for the sample
             if p_value <= self.alpha:
                 y_pred[i] = 1
         return y_pred, p_values
     def train(self, epochs, batch_size, verbose=True):
         # Compute p-values for real and fake samples
         D_real = self.discriminator.predict(self.X_train)
         D_fake = self.discriminator.predict(self.generator.predict(np.random.normal(0, 1, (len(self.X_train), 100)))).ravel()

         # Compute conformal prediction intervals
         self.p_real = np.zeros_like(D_real)
         self.p_fake = np.zeros_like(D_fake)

         for i in range(len(D_real)):
             self.p_real[i] = np.sum(D_real >= D_real[i]) / (len(
```

```
D_real) + 1)
            self.p_fake[i] = np.sum(D_fake >= D_fake[i]) / (len(D_fake) + 1)

        # Train the GAN
        for epoch in range(epochs):
            # Train discriminator on real samples
            idx = np.random.randint(0, len(self.X_train), batch_size)
            X_real = self.X_train[idx]
            y_real = np.ones((batch_size, 1))
            d_loss_real = self.discriminator.train_on_batch(X_real, y_real)

            # Train discriminator on fake samples
            z = np.random.normal(0, 1, (batch_size, 100))
            X_fake = self.generator.predict(z)
            y_fake = np.zeros((batch_size, 1))
            d_loss_fake = self.discriminator.train_on_batch(X_fake, y_fake)

            # Train generator
            z = np.random.normal(0, 1, (batch_size, 100))
            y = np.ones((batch_size, 1))
            g_loss = self.gan.train_on_batch(z, y)

            # Print progress
            if verbose and epoch % 100 == 0:
                print(f"Epoch {epoch}: D_loss_real={d_loss_real[0]}, D_loss_fake={d_loss_fake[0]}, G_loss={g_loss}")

    def generate_samples(self, n_samples):
        z = np.random.normal(0, 1, (n_samples, 100))
        samples = self.generator.predict(z)

        # Compute prediction intervals for samples
        p_values = np.zeros((n_samples,))
        for i in range(n_samples):
            D = self.discriminator.predict(np.expand_dims(samples[i], axis=0)).ravel()
            p_values[i] = np.sum(D >= D.real[i]) / (len(D) + 1)

        lower_bound = np.percentile(samples, (self.alpha / 2) * 100, axis=0)
        upper_bound = np.percentile(samples, (1 - (self.alpha / 2)) * 100, axis=0)
        in_prediction_interval = (p_values > self.alpha / 2) & (p_values < 1 - self.alpha / 2)

        return samples, lower_bound, upper_bound, in_prediction_interval
```

```python
    def train_with_coverage(self, epochs, batch_size, verbose=True, validation_data=None):
        train_loss = []
        train_coverage = []
        if validation_data is not None:
            test_loss = []
            test_coverage = []
        for epoch in range(epochs):
            # Train the discriminator
            idx = np.random.randint(0, self.x_train.shape[0], batch_size)
            x_real = self.x_train[idx]
            y_real = np.ones((batch_size,))
            x_gen = self.generator.predict(np.random.normal(size=(batch_size, self.generator.input_shape[1])))
            y_gen = np.zeros((batch_size,))
            x = np.concatenate([x_real, x_gen], axis=0)
            y = np.concatenate([y_real, y_gen], axis=0)
            d_loss, d_acc = self.discriminator.train_on_batch(x, y)
            # Train the generator
            z = np.random.normal(size=(batch_size, self.generator.input_shape[1]))
            y = np.ones((batch_size,))
            g_loss = self.gan.train_on_batch(z, y)
            # Compute the coverage for the training set
            y_pred, p_values = self.predict(self.x_train)
            train_loss.append([d_loss, g_loss])
            train_coverage.append(np.mean(y_pred == y_real))
            # Compute the coverage for the validation set
            if validation_data is not None:
                x_val, y_val = validation_data
                y_pred_val, p_values_val = self.predict(x_val)
                d_loss_val, d_acc_val = self.discriminator.evaluate(x_val, y_val, verbose=0)
                g_loss_val = self.gan.evaluate(z, y, verbose=0)
                test_loss.append([d_loss_val, g_loss_val])
                test_coverage.append(np.mean(y_pred_val == y_val))

            if verbose:
                print("Epoch {}/{} - Discriminator Loss: {:.4f} - Discriminator Accuracy: {:.4f} - Generator Loss: {:.4f} - Train Coverage: {:.4f}".format(epoch+1, epochs, d_loss, d_acc, g_loss , train_coverage[-1]))
        # Return the training history
        history = {"train_loss": train_loss, "train_coverage": train_coverage}
        if validation_data is not None:
            history["test_loss"] = test_loss
            history["test_coverage"] = test_coverage
            return history
```

Listing 1: Python implementation for conformalized GAN.

3.2 Synthesize data using conformal GAN

```python

import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.layers import Input, Dense, Reshape, Flatten
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.optimizers import Adam
from conformal_gan import ConformalGAN

# Generate the training data
n_samples = 10000
seq_length = 50
data = np.zeros((n_samples, seq_length))
for i in range(n_samples):
    phase = np.random.uniform(low=0.0, high=2*np.pi)
    freq = np.random.uniform(low=0.01, high=0.1)
    amp = np.random.uniform(low=0.1, high=1.0)
    x = np.linspace(0, seq_length/freq, seq_length, endpoint=False)
    data[i] = amp * np.sin(2*np.pi*freq*x + phase)

# Normalize the data
data = (data - np.mean(data)) / np.std(data)

# Define the generator model
latent_dim = 10
generator = Sequential()
generator.add(Dense(256, input_dim=latent_dim, activation='relu'))
generator.add(Dense(512, activation='relu'))
generator.add(Dense(1024, activation='relu'))
generator.add(Dense(seq_length, activation='tanh'))

# Define the discriminator model
discriminator = Sequential()
discriminator.add(Flatten(input_shape=(seq_length,)))
discriminator.add(Dense(512, activation='relu'))
discriminator.add(Dense(256, activation='relu'))
discriminator.add(Dense(1, activation='sigmoid'))

# Compile the discriminator
discriminator.compile(loss='binary_crossentropy', optimizer=Adam(0.0002, 0.5), metrics=['accuracy'])

# Define the GAN
z = Input(shape=(latent_dim,))
time_series = generator(z)
validity = discriminator(time_series)
gan = Model(z, validity)

# Compile the GAN
gan.compile(loss='binary_crossentropy', optimizer=Adam(0.0002,
```

```
    0.5))

# Train the GAN with conformal prediction
alpha = 0.05
n_train = 8000
x_train_subset = data[:n_train]
conformal_gan = ConformalGAN(generator, discriminator, gan,
    x_train_subset, alpha=alpha)
train_loss, test_loss, train_coverage, test_coverage =
    conformal_gan.train_with_coverage(epochs=5000, batch_size=32,
     verbose=True)

# Generate synthetic time-series data
n_samples = 10
z = np.random.normal(size=(n_samples, latent_dim))
synthetic_data = generator.predict(z)

# Plot the training and validation loss
plt.plot(train_loss)
plt.plot(test_loss)
plt.title('Conformal GAN Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()

# Plot the training and validation coverage
plt.plot(train_coverage)
plt.plot(test_coverage)
plt.title('Conformal GAN Coverage')
plt.ylabel('Coverage')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()

# Plot the synthetic time-series data
for i in range(n_samples):
    plt.plot(synthetic_data[i])
plt.title('Synthetic Time-Series Data')
plt.xlabel('Time')
plt.ylabel('Value')
plt.show()
```

Listing 2: Python implementation for generating data using conformal GAN.