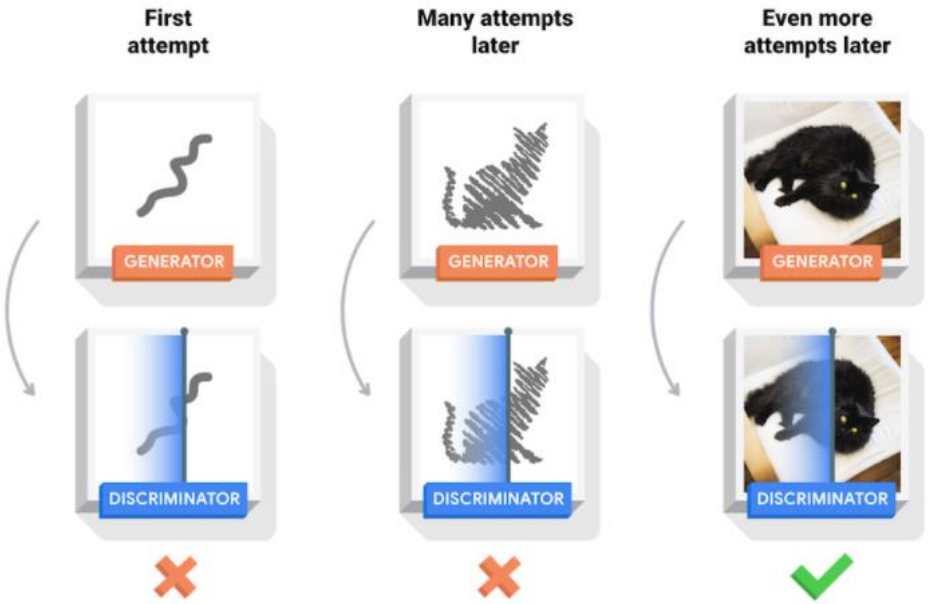
Chciałbym przedstawić ciekawy program, który generuje obraz odręcznie narysowanych cyfr za pomocą sieci Deep Convolutional Generative Adversarial Network (DCGAN).

Generative Adversarial Networks (GAN) polega na tym, że dwa modele są trenowane jednocześnie. Generator uczy się tworzyć obrazy, które wyglądają na prawdziwe, a dyskryminator uczy się odróżniać rzeczywiste obrazy od podróbek. Podczas treningu generator staje się coraz lepszy w tworzeniu obrazów, które wyglądają realistycznie, a dyskryminator coraz lepiej je rozróżnia. Proces osiąga równowagę, gdy dyskryminator nie może już odróżnić prawdziwych obrazów od podróbek.




Na początku należy stworzyć modele czyli generator i dyskryminator. Tworzone są za pomocą interfejsu Keras Sequential API.

Generator wykorzystuje tf.keras.layers.Conv2DTranspose do tworzenia obrazu.

```
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, activation='tanh'))
    assert model.output_shape == (None, 28, 28, 1)

    return model
```

Tutaj tworzymy obraz za pomocą nieprzeszkolonego generatora.

```
generator = make_generator_model()

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

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

Dyskryminator to klasyfikator obrazów

```
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
```

Następnie używamy dyskryminatora do klasyfikowania wygenerowanych obrazów jako prawdziwe lub fałszywe. Dyskryminator również jest jeszcze nieprzeszkolony.

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

Utrata dyskryminatora - metoda określa ilościowo, jak dobrze dyskryminator jest w stanie odróżnić prawdziwe obrazy od podróbek. Porównuje przewidywania dyskryminatora dotyczące rzeczywistych obrazów z tablicą jedynek, a przewidywania dyskryminatora dotyczące fałszywych (generowanych) obrazów z tablicą zer.

```
  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
```

Utrata generatora - określa ilościowo, jak dobrze był on w stanie oszukać dyskryminator. Jeśli generator działa dobrze, dyskryminator sklasyfikuje fałszywe obrazy jako rzeczywiste.

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

Pętla szkoleniowa rozpoczyna się, gdy generator otrzymuje losowe ziarno (seed) jako dane wejściowe. To ziarno jest używane do tworzenia obrazu. Dyskryminator jest dalej używany do klasyfikowania obrazów rzeczywistych (pochodzących ze zbioru uczącego) i obrazów fałszywych (wytwarzanych przez generator). Straty są obliczane dla każdego z modeli, a gradienty są używane do aktualizacji generatora i dyskryminatora.

.

```
@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))
```

Aby wytrenować model należy użyć fukcji train()

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

    for image_batch in dataset:
      train_step(image_batch)

    display.clear_output(wait=True)
    generate_and_save_images(generator,
                             epoch + 1,
                             seed)

    if (epoch + 1) % 15 == 0:
      checkpoint.save(file_prefix = checkpoint_prefix)

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

  display.clear_output(wait=True)
  generate_and_save_images(generator,
                           epochs,
                           seed)
```

Na początku treningu wygenerowane obrazy wyglądają jak losowy szum. W miarę postępu szkolenia generowane cyfry będą wyglądać coraz bardziej realistycznie.