[Advanced-Deep-Learning-with-Keras/chapter4-gan/cgan-mnist-4.3.1.py at master · PacktPublishing/Advanced-Deep-Learning-with-Keras (github.com)](https://github.com/PacktPublishing/Advanced-Deep-Learning-with-Keras/blob/master/chapter4-gan/cgan-mnist-4.3.1.py)

Trains CGAN on MNIST using Keras

CGAN is Conditional Generative Adversarial Network. This version of CGAN is similar to DCGAN. The difference mainly is that the z-vector of geneerator is conditioned by a one-hot label to produce specific fake images. The discriminator is trained to discriminate real from fake images that are conditioned on specific one-hot labels.

[GAN-in-keras-on-mnist/GAN-keras-mnist-MLP.ipynb at master · kroosen/GAN-in-keras-on-mnist (github.com)](https://github.com/kroosen/GAN-in-keras-on-mnist/blob/master/GAN-keras-mnist-MLP.ipynb)

*# Optimizer*

adam **=** Adam(lr**=**0.0002, beta\_1**=**0.5)

g **=** Sequential()

g**.**add(Dense(256, input\_dim**=**z\_dim, activation**=**LeakyReLU(alpha**=**0.2)))

g**.**add(Dense(512, activation**=**LeakyReLU(alpha**=**0.2)))

g**.**add(Dense(1024, activation**=**LeakyReLU(alpha**=**0.2)))

g**.**add(Dense(784, activation**=**'sigmoid')) *# Values between 0 and 1*

g**.**compile(loss**=**'binary\_crossentropy', optimizer**=**adam, metrics**=**['accuracy'])

d **=** Sequential()

d**.**add(Dense(1024, input\_dim**=**784, activation**=**LeakyReLU(alpha**=**0.2)))

d**.**add(Dropout(0.3))

d**.**add(Dense(512, activation**=**LeakyReLU(alpha**=**0.2)))

d**.**add(Dropout(0.3))

d**.**add(Dense(256, activation**=**LeakyReLU(alpha**=**0.2)))

d**.**add(Dropout(0.3))

d**.**add(Dense(1, activation**=**'sigmoid')) *# Values between 0 and 1*

d**.**compile(loss**=**'binary\_crossentropy', optimizer**=**adam, metrics**=**['accuracy'])

d**.**trainable **=** **False**

inputs **=** Input(shape**=**(z\_dim, ))

hidden **=** g(inputs)

output **=** d(hidden)

gan **=** Model(inputs, output)

gan**.**compile(loss**=**'binary\_crossentropy', optimizer**=**adam, metrics**=**['accuracy'])

Functional API vs sequential API

[Keras Model Sequential API VS Functional API | by Rahmat Faisal | Analytics Vidhya | Medium](https://medium.com/analytics-vidhya/keras-model-sequential-api-vs-functional-api-fc1439a6fb10)

### ¿Qué es una Conditional GAN?

Una Conditional GAN (CGAN) es una variación de una Generative Adversarial Network (GAN) donde tanto el generador como el discriminador reciben información adicional sobre las condiciones bajo las cuales se deben generar y evaluar los datos. En este caso, las condiciones son las etiquetas de las clases de las imágenes (por ejemplo, dígitos del 0 al 9 en el dataset MNIST).

### Componentes de una CGAN

Una CGAN consta de dos redes principales:

1. \*\*Generador (Generator)\*\*: Este modelo toma un vector de ruido y una etiqueta de clase como entrada y genera una imagen sintética que intenta parecerse a una imagen real de la clase especificada.

2. \*\*Discriminador (Discriminator)\*\*: Este modelo toma una imagen (real o generada) y una etiqueta de clase como entrada y predice si la imagen es real o generada por el generador.

### Estructura del Generador

El generador toma como entrada dos vectores:

- \*\*Vector de ruido (noise)\*\*: Un vector aleatorio de dimensiones `noise\_dim`.

- \*\*Vector de etiquetas (label)\*\*: Un vector de una-hot encoding representando la clase de la imagen.

#### Funcionamiento del Generador:

1. \*\*Concatenación\*\*: El vector de ruido y el vector de etiquetas se concatenan para formar una entrada combinada.

2. \*\*Capas Densas y Activaciones\*\*: Esta entrada pasa por varias capas densas, cada una seguida por una activación LeakyReLU y normalización por lotes.

3. \*\*Generación de Imagen\*\*: La salida final pasa por una capa densa con activación 'tanh' y se remodela para obtener la forma deseada de la imagen (por ejemplo, 28x28x1 para MNIST).

#### Código del Generador:

```python

def build\_generator(noise\_dim, num\_classes, img\_shape):

noise = Input(shape=(noise\_dim,))

label = Input(shape=(num\_classes,))

model\_input = concatenate([noise, label])

x = Dense(256)(model\_input)

x = LeakyReLU(alpha=0.2)(x)

x = BatchNormalization(momentum=0.8)(x)

x = Dense(512)(x)

x = LeakyReLU(alpha=0.2)(x)

x = BatchNormalization(momentum=0.8)(x)

x = Dense(1024)(x)

x = LeakyReLU(alpha=0.2)(x)

x = BatchNormalization(momentum=0.8)(x)

x = Dense(np.prod(img\_shape), activation='tanh')(x)

img = Reshape(img\_shape)(x)

return Model([noise, label], img)

```

### Estructura del Discriminador

El discriminador toma como entrada:

- \*\*Imagen (img)\*\*: Una imagen que puede ser real (del conjunto de datos) o generada por el generador.

- \*\*Vector de etiquetas (label)\*\*: Un vector de una-hot encoding representando la clase de la imagen.

#### Funcionamiento del Discriminador:

1. \*\*Concatenación\*\*: La imagen es aplanada y concatenada con el vector de etiquetas para formar una entrada combinada.

2. \*\*Capas Densas y Activaciones\*\*: Esta entrada pasa por varias capas densas, cada una seguida por una activación LeakyReLU.

3. \*\*Clasificación\*\*: La salida final pasa por una capa densa con activación 'sigmoid' que predice si la imagen es real o falsa.

#### Código del Discriminador:

```python

def build\_discriminator(num\_classes, img\_shape):

img = Input(shape=img\_shape)

label = Input(shape=(num\_classes,))

model\_input = concatenate([Flatten()(img), label])

x = Dense(512)(model\_input)

x = LeakyReLU(alpha=0.2)(x)

x = Dense(512)(x)

x = LeakyReLU(alpha=0.2)(x)

x = Dense(512)(x)

x = LeakyReLU(alpha=0.2)(x)

validity = Dense(1, activation='sigmoid')(x)

return Model([img, label], validity)

```

### Entrenamiento de la CGAN

El entrenamiento de una CGAN sigue un ciclo en el que se actualizan alternadamente los pesos del generador y del discriminador:

1. \*\*Entrenamiento del Discriminador\*\*:

- Se generan imágenes falsas usando el generador.

- Se toman imágenes reales del conjunto de datos.

- Se entrena el discriminador para distinguir entre imágenes reales y falsas, usando las imágenes reales y las generadas, junto con sus etiquetas correspondientes.

2. \*\*Entrenamiento del Generador\*\*:

- Se congela el discriminador para que no se actualicen sus pesos.

- Se entrena el generador con la esperanza de que el discriminador clasifique las imágenes generadas como reales.

- Se proporciona retroalimentación al generador basada en la salida del discriminador.

#### Código de Entrenamiento:

```python

def train(generator, discriminator, combined, epochs, batch\_size, noise\_dim, num\_classes, img\_shape):

(X\_train, y\_train), (\_, \_) = mnist.load\_data()

X\_train = (X\_train.astype(np.float32) - 127.5) / 127.5

X\_train = np.expand\_dims(X\_train, axis=3)

y\_train = to\_categorical(y\_train, num\_classes)

valid = np.ones((batch\_size, 1))

fake = np.zeros((batch\_size, 1))

for epoch in range(epochs):

idx = np.random.randint(0, X\_train.shape[0], batch\_size)

imgs, labels = X\_train[idx], y\_train[idx]

noise = np.random.normal(0, 1, (batch\_size, noise\_dim))

gen\_labels = to\_categorical(np.random.randint(0, num\_classes, batch\_size), num\_classes)

gen\_imgs = generator.predict([noise, gen\_labels])

d\_loss\_real = discriminator.train\_on\_batch([imgs, labels], valid)

d\_loss\_fake = discriminator.train\_on\_batch([gen\_imgs, gen\_labels], fake)

d\_loss = 0.5 \* np.add(d\_loss\_real, d\_loss\_fake)

g\_loss = combined.train\_on\_batch([noise, gen\_labels], valid)

if epoch % 1000 == 0:

print(f"{epoch} [D loss: {d\_loss[0]}] [G loss: {g\_loss}]")

return generator

```

### Resumen del Proceso

1. \*\*Generador\*\*: Toma ruido y etiquetas, produce imágenes condicionadas.

2. \*\*Discriminador\*\*: Toma imágenes y etiquetas, predice si son reales o falsas.

3. \*\*Entrenamiento Alternado\*\*:

- Discriminador aprende a distinguir imágenes reales de falsas.

- Generador aprende a engañar al discriminador.

4. \*\*Entrenamiento Conjunto\*\*: Se combinan ambas redes para entrenar el generador a través de la retroalimentación del discriminador.

Esta estructura permite que la CGAN genere imágenes que no solo se ven reales, sino que también se alinean con las etiquetas de clase proporcionadas, lo cual es útil para tareas donde es importante condicionar la generación de datos en alguna clase específica.

Imagen que contiene Tabla

Descripción generada automáticamente

[Redes Neuronales Convolucionales en Profundidad - DataSmarts Español](https://datasmarts.net/es/redes-neuronales-convolucionales-en-profundidad/#:~:text=N%C3%BAmero%20de%20filtros%3A%20Es%20el,regi%C3%B3n%20de%20la%20capa%20anterior.)

# Calculo de las dimensiones entre capas Conv2D y TransConv2D

## Padding = valid

Output\_size = (Input\_size – 1) \* stride + kernel\_size – 2\*padding + output\_padding

### Parámetros

* **Input\_size:** Tamaño de la entrada (8 en este caso, porque estamos comenzando con 8x8).
* **stride:** Paso de la convolución (2 en este caso).
* **kernel\_size:** Tamaño del filtro (3 en este caso).
* **padding:** Tipo de padding. Para padding='valid', el valor es 0.
* **output\_padding:** Este valor generalmente es 0 a menos que lo especifiquemos explícitamente, lo cual no estamos haciendo aquí.

## Padding = same

Output\_size = (Input\_size) \* stride

Dudas

Como se comporta la binary cross entropy

    with tf.GradientTape() as tape:

ver bien como funca eso