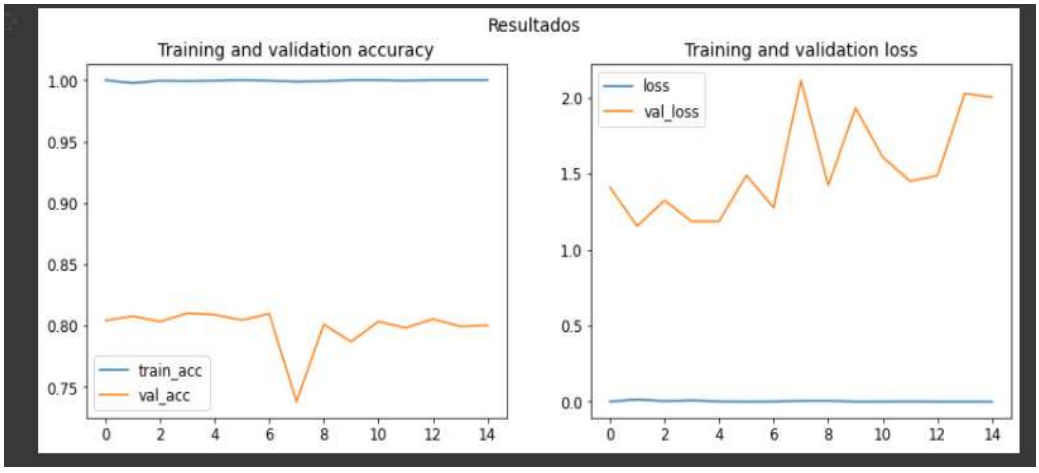
1. lasificación de secuencias: IMDB

1.1. *[1 pts] TODO 1. Añadir el código necesario para mostrar las curvas de aprendizaje y analizar las mismas.*

Respuesta: Las gráficas muestran un overfeating muy fuerte con la parte de train, ya que desde la primera época alcanza un accuracy de 100% y una función de perdida *loss* muy cercana al 0. Para la parte de validación el indicador de accuracy apenas llega al 80% y la función de coste va creciendo conforme avanza las épocas.

```
matplotlib.rcParams.update({'font.size': 12})

def print_result(my_history,my_dropout=None):

    acc = my_history.history['accuracy']
    val_acc = my_history.history['val_accuracy']

    loss = my_history.history['loss']
    val_loss = my_history.history['val_loss']

    # Extraemos el número de épocas
    epochs = range(len(acc))

    fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2,figsize=(14,5))
    # Representamos con plot nuestro accuracy por epoch
    ax1.plot(epochs, acc, label = 'train_acc')
    ax1.plot(epochs, val_acc, label = 'val_acc')
    if (my_dropout):
      title= 'Training and validation accuracy with dropout rate = '+str(my_dropout)
    else:
      title= 'Training and validation accuracy'

    ax1.set_title(title)
    ax1.legend()


    # Representamos con plot nuestro loss por epoch
    ax2.plot(epochs, loss, label = 'loss')
    ax2.plot(epochs, val_loss, label = 'val_loss')
    if (my_dropout):
      title= 'Training and validation loss with dropout rate = '+str(my_dropout)
    else:
      title= 'Training and validation loss'

    ax2.set_title(title)
    ax2.legend()
    fig.suptitle('Resultados')

    plt.show()


    return None
```

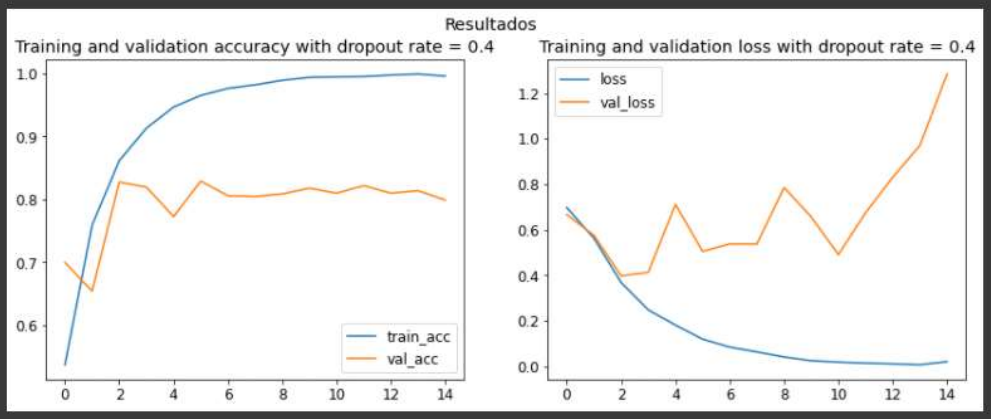


[1 pts]TODO 2. Añadir Dropout a nuestra red de la forma indicada y analizar el resultado.

Respuesta: En las gráficas se ve una mejora sustancial con respecto a la primera ejecución del primer apartado. Aunque apunta a overfeating/sobrajuste, no es tan fuerte como el del apartado 1, y de producirse produce en la épocas finales. Los marcadores para la parte de validación mejoran un poco, ya superan el 80% de accuracy y la función de perdida se mantiene más o menos estable entorno al 0,8 , aunque en las útimas épocas se dispara por encima del 1,2.

```
[ ] # Construimos el nuevo modelo con las indicaciones de Dropout al 0,4
    print('Build model...')
    model_2=None
    model_2 = Sequential(name='Model_with_Droput_Layers')
    model_2.add(Embedding(max_features, 32))
    model_2.add(Dropout(0.4))
    model_2.add(LSTM(32))
    model_2.add(Dropout(0.4))
    model_2.add(Dense(1, activation='sigmoid'))

    #Compilamos el modelo
    model_2.compile(loss='binary_crossentropy',
                    optimizer='rmsprop',
                    metrics=['accuracy'])

    # Visualizamos la arquitectura y los parámetros
    model_2.summary()
    #Entrenamos el modelo
    history_2=model_2.fit(x_train, y_train,
              batch_size=batch_size,
              epochs=15,
              validation_data=(x_test, y_test))

    #Evaluamos con la parte de test.
    score, acc = model_2.evaluate(x_test, y_test,
                                  batch_size=batch_size)
    print('Test score:', score)
    print('Test accuracy:', acc)
```

```
[ ] print_result(history_2,my_dropout=0.4)
```



1.2. ***[1 pts]TODO 3. Modificar la red para utilizar una combinación de CNN, LSTM y DNN que resuelva el problema de clasificación de secuencias. ¿Por qué combinamos estas redes?***

Respuesta: Utilizando estas tres tipologías de Red, lo que hacemos es aprovechar los puntos fuertes tanto de la CNN como de la LSTM, la DNN queda para la parte final de clasificación. Las CNN se usan y funcionan muy bien para la detección de características y patrones. Detectaría en el embedding de entrada una serie de patrones o características que se dieran de forma constante para los textos con sentimiento positivo como para los de sentimiento negativo. Una vez obtenido este vector de características y generadas a modo de secuencias, se le pasaría a la LSTM, que funciona muy bien tratando secuencias y guardando relaciones entre los distintos elementos tanto de corto como de largo plazo.

1.3. ***[1 pts]. TODO 4. Hora de la creatividad, modifica la red que hemos desarrollado para obtener los mejores resultados posibles, cuéntame qué has hecho y qué resultados te ha dado (también me puedes contar lo que no haya funcionado), asegúrate de darme tus hipótesis acerca de por qué vemos tales resultados.***

Respuesta: He trabajado en 4 pasos. Cada paso agrega las soluciones de mejora del anterior paso. Y aunque los he ido combinando en cada uno de los 4 pasos, las actuaciones las dividiría en:

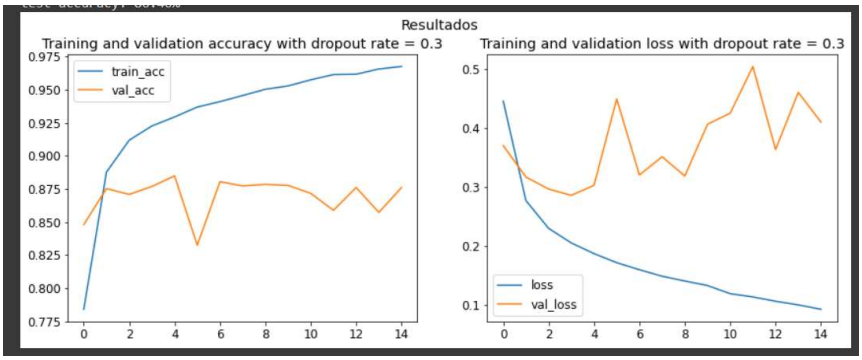
- ***Datos:*** Esta iniciativa tiene el objetivo de que cuanto mayor sea la cantidad de datos con las que entrene el modelo mejor será su entrenamiento. Así también importante es sacar de la fase de entreno los datos de test, para que luego la fase de evaluación con los datos de test, sea más fiable. En concreto lo que se ha realizado es:
    - Utilización de la totalidad del conjunto de los 25000 datos de entrenamiento como de test.
    - Creación de un dataset de validación con un 10% de los datos de entrenamiento (2500)
    - Aumento a unl Batch Size=512

- ***Arquitectura y Callbacks:*** Esta iniciativa tiene el objetivo de la mejora del aprendizaje, optando por reforzar la arquitectura incluyendo 2 capas de Dropout con un dropout_rate de 0.3 y la inclusión de dos tipos de callbacks que en principio nos permitirán optimizar la convergencia. Los callback utilizados son:
    - ***ReduceLROnPlateau.*** Nos permitirá ir reduciendo el learning rate, con un factor de reducción de 0.5, si se observa que en 2 épocas no mejora.
    - ***EarlyStopping.*** *Este callback corta el entrenamiento si ve que no hay mejora en 3 epocas consecutivas y se queda con el resultado.*

- ***Optimizadores y parámetros:*** Esta iniciativa va encaminada a la mejora de los scores, desde la optimización de parámetros del modelo, incluyendo los optimizadores. Las iniciativas en este ámbito han sido subir el número de epochs de 15 a 30 y 45. Utilización del optimizador Adam.

Una vez descritas las 3 inicativas en las que se agrupan los cambios, estos se han distribuidos en 4 fases, en las que se entremezclan. Las fases han sido:

- ***Fase 1: Ampliamos el número de datos, generamos un conjunto de validación , aumentamos batch_size=512, insertamos capa Dropout de 0.3***

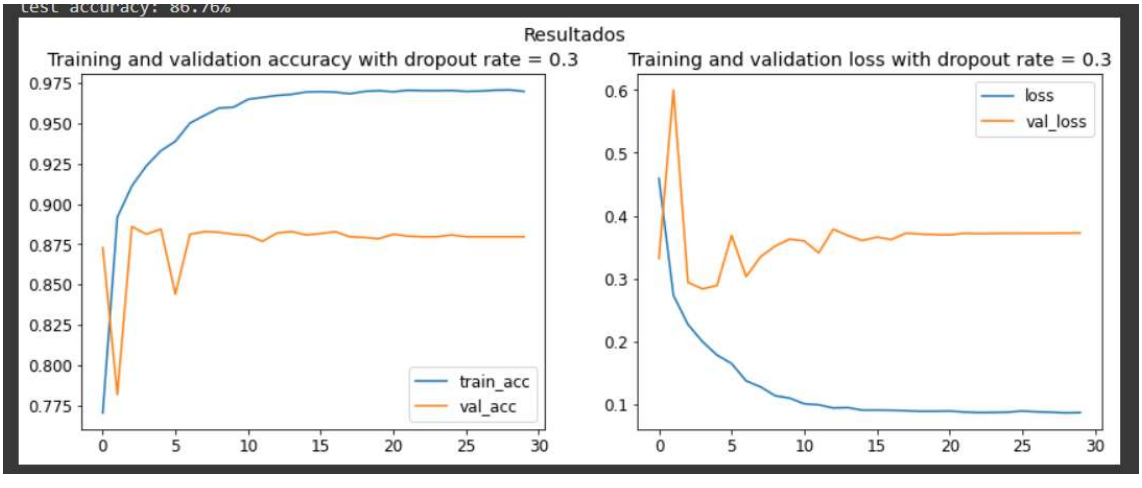


Resultados evaluación contra test: **loss: 0.43, accuracy: 86.40%**

En esta fase1 vemos que los cambios han tenido un gran impacto, ya que se ha incrementado en más de un 6% la mejora en el accuracy y la función de perdida se ha reducido en más de un punto llegando al 0,43. En las gráficas de entreno se ve una mejoría notable en cuanto al efecto de overfeating.

Estimo que el punto clave han sido la utilización de todos los datos y la inserción de capas de Dropout.

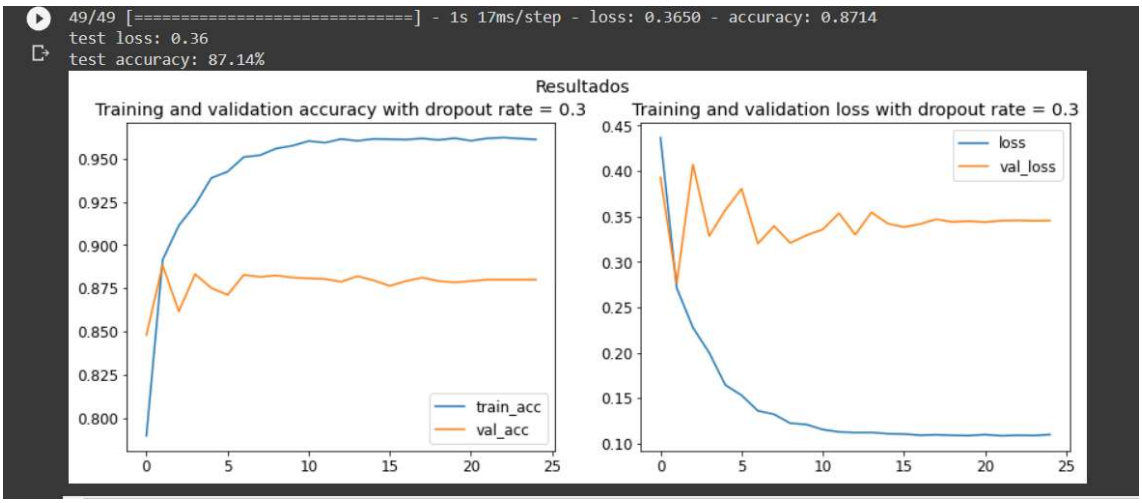
- ***Fase 2: Mejoras Fase 1 + añadimos un callback ReduceLROnPlateau y ampliamos epoch a 30***



Resultados evaluación contra test: **loss: 0.40, accuracy: 86.76%**

La adopción de estas medidas ha tenido un leve impacto de mejora, con respecto a las anteriores. Además, se observa que el incremente de épocas no aporta mejora desde la 15 en adelante ya que los valores se estabilizan.

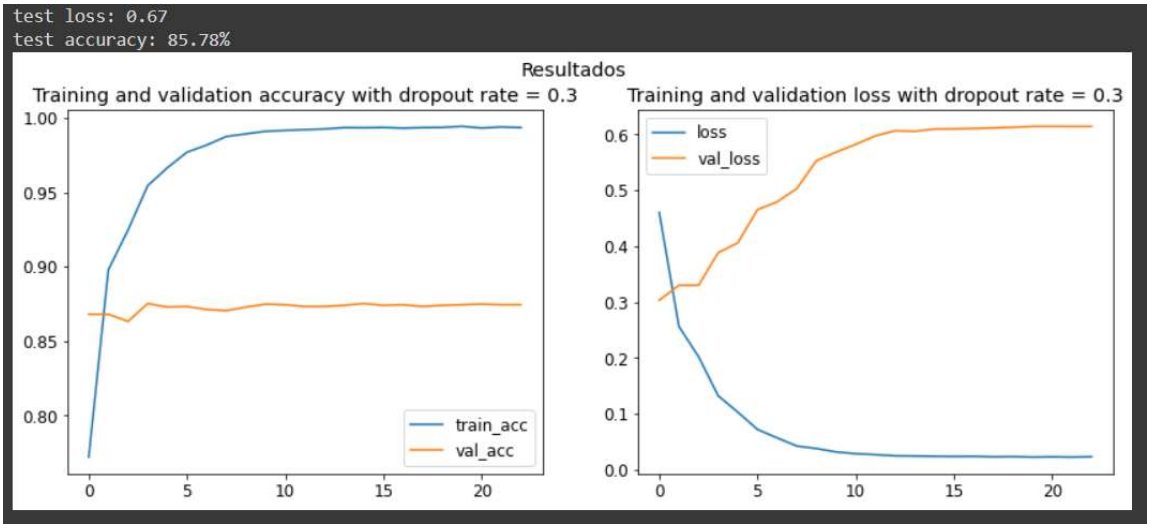
- ***Fase 3: Mejoras Fase 2 + Callback EarlyStopping***



Resultados evaluación contra test: **loss: 0.36, accuracy: 87.14%**

La adopción de estas medidas vuelve a mejorar un poco más los marcadores. Se supera el 87% de accuracy y se baja del 0,4 de la función de perdida.

- ***Fase 4: Mejoras Fase 3 + Nuevo Optimizador Adams***



Resultados evaluación contra test: **loss: 0.67, accuracy: 85.78%**

La acción de cambiar el optimizador por un Adam en este caso no ha tenido efecto de mejora, sino que ha bajado los scores.

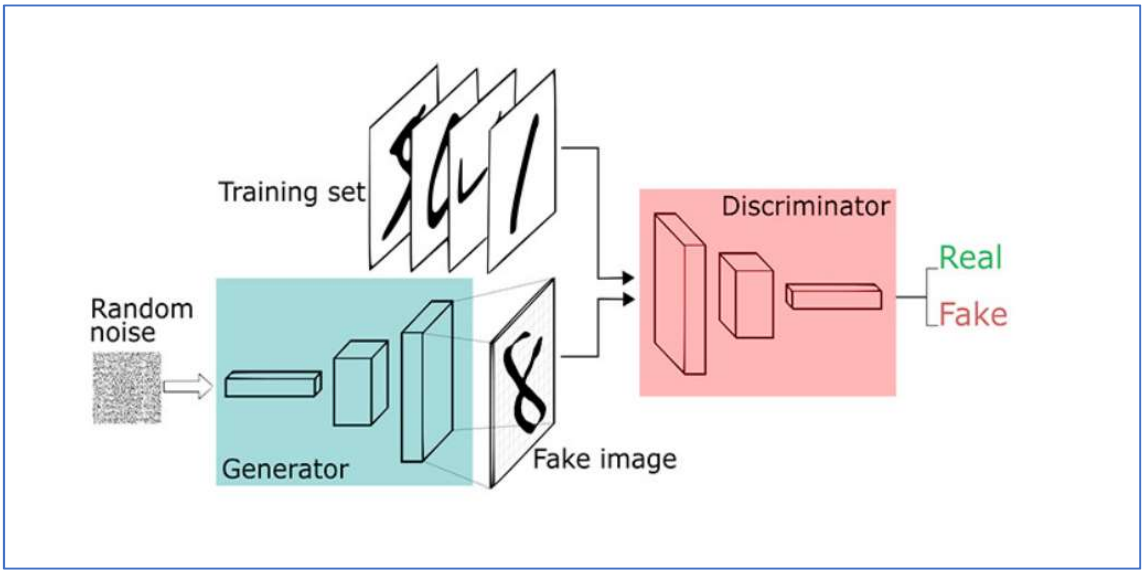
2. Generando datos de MNIST.

2.1. [1 pts] TODO 1. Explica a alto nivel el funcionamiento de las redes GAN

Respuesta:

Las redes GAN (Generative Adversarial Network) son un tipo de redes formadas a su vez por dos redes neuronales profundas enfrentadas, una llamada generador y otro discriminador, que compitiendo entre ellas en un juego de suma cero (las pérdidas de una se compensan con las ganancias de otras), llegan a generar algo completamente nuevo. Los casos de uso para lo que se están utilizando es la generación sintética de datos, data augmentation, generación de nuevos diseños en el ámbito **industrial** y de moda, generación de imágenes a partir de texto, *Deepfakes* …etc. Actualmente el caso de uso más conocido y más reconocible es la generación de imágenes reales.

Con más detalle, la red GAN se conforma con dos redes neuronales profundas, confrontadas o antagónicas, una de ellas es la ya mencionada con el nombre de **Generador** que genera muestras (texto, imágenes, sonidos …), a partir de un vector de ruido, y estas muestras son evaluadas por la segunda red el **Discriminador**, una red entrenada con muestras reales y cuya misión será discriminar si las muestras realizadas por el generador son reales (1) o son falsas (0).

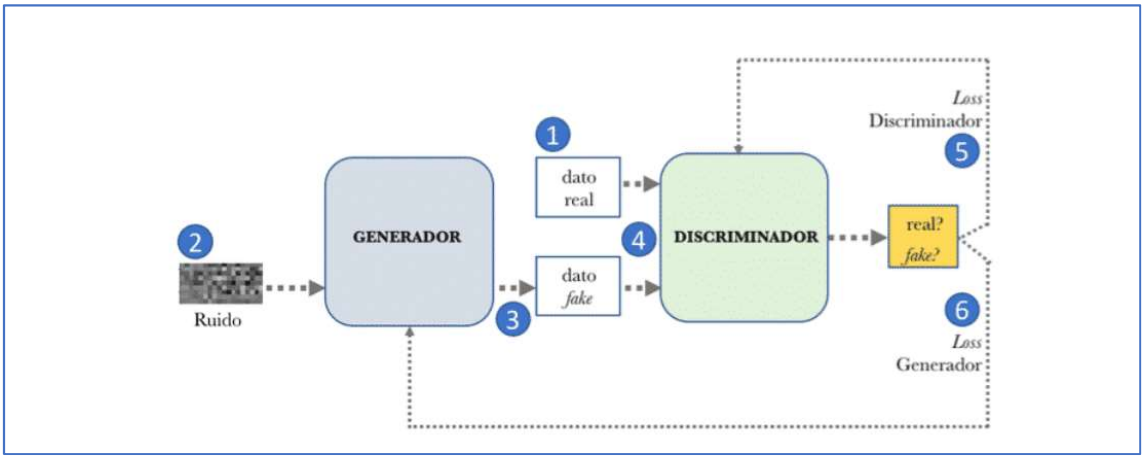


El funcionamiento se podría resumir en que al principio al discriminador le será muy sencillo clasificar como falsas las muestras que le envíe el generador, pero de esos resultados y por medio de los ajustes de parámetros y pesos que nos permite el backpropagation, el generador irá "aprendiendo" y creando cada vez muestras mejores, más cercanas a la realidad.

Al generador le costará cada vez más discriminar si son verdaderas o falsa, por lo que la probabilidad de ser determinada como falsa irá disminuyendo mientras la probabilidad de ser discriminada como verdadera irá creciendo.

En el momento que el entrenamiento converja y la probabilidad tanto de ser verdadera (1) o falsa (0) sea del 50% el entrenamiento finalizará, ya que estaríamos

en un escenario totalmente aleatorio en el que el discriminador no sabrá decantarse por si una muestra es falsa o verdadera.



2.2. [1 pts] TODO 2. Tanto el generador como el discriminador son redes convolucionales. ¿Cuáles son las Inputs y las Outputs esperadas tanto antes como después del entrenamiento?

Respuesta:

Generador:

- Input: al inicio del entrenamiento es un vector de ruido, en este caso como se entrenan por lotes, están siendo lotes de 256 vectores con una longitud de 100 cada uno.
- Output: Salidas de imágenes generadas con unas dimensiones de 28x28x1 a semejanza de las imágenes reales.

Discriminador:

- Input: Tanto imágenes originales del dataset de MNIST de 28x28x1, como imágenes creadas por el Generador de 28x28x1.
- Output: Salida con la clasificación de las imágenes 1 reales 0 falsas, tanto para el lote de imágenes reales como las creadas por el generador.

2.3. *TODO 3. Las funciones clave de este notebook son train y su subfunción train_step, explica paso a paso qué hacen*

Respuesta:

- **Train**: Es la función principal, es la que orquesta el entrenamiento de la red GAN. Contiene dos bucles anidados el principal sirve para recorrer las epocas y el secundario para recorrer los pasos o step de cada época. Por cada época muestra un ejemplo de imágenes creadas por el generador y cada 15 épocas hace un salvado de los modelos y pesos.

```
def train(dataset, epochs):
# Recibe como parámetros de entrada el dataset de imágenes reales de mnist, y el número de épocas que queremos que entrene la red GAN.
 for epoch in range(epochs):
# Bucle principal, que itera tantas veces como número de épocas prefijadas.
   start = time.time() # Inicializamos tiempo para luego saber cuanto tiempo dura cada época.

# Bucle secundario que itera tantas veces como batch de imgenes contenga el dataset. Sería equivalente al step per epoch que parametrizamos en los modelos.
   for image_batch in dataset:
# Función muy importante que es donde se produce el entrenamiento de cada paso, donde el generador crea imágenes sintéticas, el discriminador evalúa si son fake y se hace el ajuste de pesos y parámetros.
    train_step(image_batch)

# Visualizamos y guardamos en un fichero, un ejemplo de las imágenes que es capaz de generar el Generador en la época que esta en curso del entrenamiento.
   display.clear_output(wait=True)
   generate_and_save_images(generator,   epoch + 1, seed)

  # Se guarda el modelo y los pesos generados durante el entreno cada 15 épocas. Esto es útil por si tenemos que para el entreno sin finalizar y así poder retomarlo donde lo dejamos.
   if (epoch + 1) % 15 == 0:
    checkpoint.save(file_prefix = checkpoint_prefix)

#Mostramos tiempo de entrenamiento de la última época
   print ('Time for epoch {} is {} sec'.format(epoch + 1, time.time()-start))

# Visualizamos y guardamos en un fichero, un ejemplo de las imágenes que es capaz de generar el Generador al final del entrenamiento.
 display.clear_output(wait=True)
 generate_and_save_images(generator, epochs, seed)
```

- **Train_step**: Es la función que orquesta las llamadas al **generador**, para que genere imágenes y posteriormente realiza 2 llamadas al **discriminador** una con las imágenes fake recién creadas y otra con las reales provenientes del dataset. Posteriormente con los outputs del **discriminador** calcula la función de perdida, loss, con esas loss realiza el cálculo de los nuevos gradientes y actualiza pesos con los optimizadores asociados al generador y discriminador.

```
# Recibe como parámetros de entrada el batch de imágenes reales
def train_step(images):
#Se genera tantos vectores de ruido como tamaño del batch
   noise = tf.random.normal([BATCH_SIZE, noise_dim])
#Uilizamos la clase GradientTape, para podercalcular los gradientes nuevos en cada paso
   with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:

# Llamamos al generador para que genere images apartir del ruido generado
     generated_images = generator(noise, training=True)
# Con las imágenes generadas y con las reales se realiza una doble llamada al discriminador para que discrimine las imágenes.
     real_output = discriminator(images, training=True)
     fake_output = discriminator(generated_images, training=True)

# Con los valores obtenidos calculamos el valor de loss, tanto del discriminador como del generador. Utiizando la función de perdida que definimos en las funciones generator_loss y discriminator_loss
     gen_loss = generator_loss(fake_output)
     disc_loss = discriminator_loss(real_output, fake_output)

# Con los valores obtenidos de perdida para el generador, gen_loss y el discriminador disc_loss, calculamos los valores del gradiente para cada uno de ellos, con las instancias creadas de la clase GradientTape. Utilizamos el método gradient para cálcular los nuevos gradientes de cada red.

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

# Con los nuevos gradientes pasamos a actualizar los parámetros del discriminador y del generador.
generator_optimizer.apply_gradients(zip(gradients_of_generator,
generator.trainable_variables))

discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
discriminator.trainable_variables))
```

3. Generando imágenes de CIFAR-10

3.1. [1.5 pts] TODO 1. Completa el código necesario del Generador

Respuesta: El código es muy similar al del notebook de MNIST, en la que también se utiliza notación secuencial para ir montando la red. En lo que hay que ir prestando atención con los strides, en las distintas capas Conv2DTranspose que hacen el upsamplig, para calcular bien el tamaño que requerimos de imagen, en este caso 32x32x3.

```
def make_generator_model():
    # TODO1: Pon aquí tu código

    model = tf.keras.Sequential()
    model.add(layers.Dense(8*8*256, use_bias=False, input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((8, 8, 256)))
    #Tamaño resultante: 8x8x256
    assert model.output_shape == (None, 8, 8, 256) # Note: None is the batch size

    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False))
    #Tamaño resultante: 8x8x128
    assert model.output_shape == (None, 8,8, 128)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    #Tamaño resultante: 16x16x64
    assert model.output_shape == (None, 16, 16, 64)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(3, (5, 5), strides=(2, 2), padding='same', use_bias=False, activation='tanh'))
    #Tamaño resultante final: 32x32x3
    assert model.output_shape == (None, 32, 32, 3)

    return model
```

[1.5 pts] TODO 2. Completa el código necesario del Discriminador

Respuesta: También hemos utilizado notación secuencial de Keras para ir construyendo la red. Tenemos en cuenta el input_shape que sea de 32x32x3.
En la última capa Dense de 1 neurona que determina si es fake o real, no indicamos la función de activación, que debería ser la sigmoid, debido a que por cómo hemos configurado, la función de perdida, BinaryCrossentropy, con el parámetro, with logits=true, ya no haría falta.

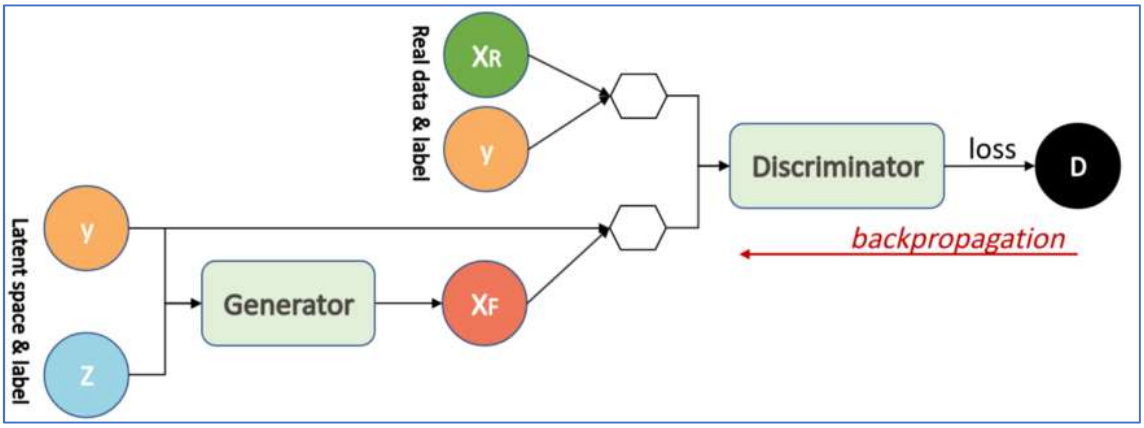
```
17] def make_discriminator_model():
        # TODO2: Pon aquí tu código
        model = tf.keras.Sequential()
        model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
                                input_shape=[32, 32, 3]))
        #Tamaño resultante: 16x16x64
        model.add(layers.LeakyReLU())
        model.add(layers.Dropout(0.3))

        model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
        #Tamaño resultante: 8x8x128
        model.add(layers.LeakyReLU())
        model.add(layers.Dropout(0.3))
        #Aplanamos antes de pasarlo a la capa Dense.
        model.add(layers.Flatten())
        model.add(layers.Dense(1))

        return model
```

4. [OPCIONAL] Conditional GANs

4.1. [0.5 pts] Explica lo que es una red GAN Condicional.

Respuesta:

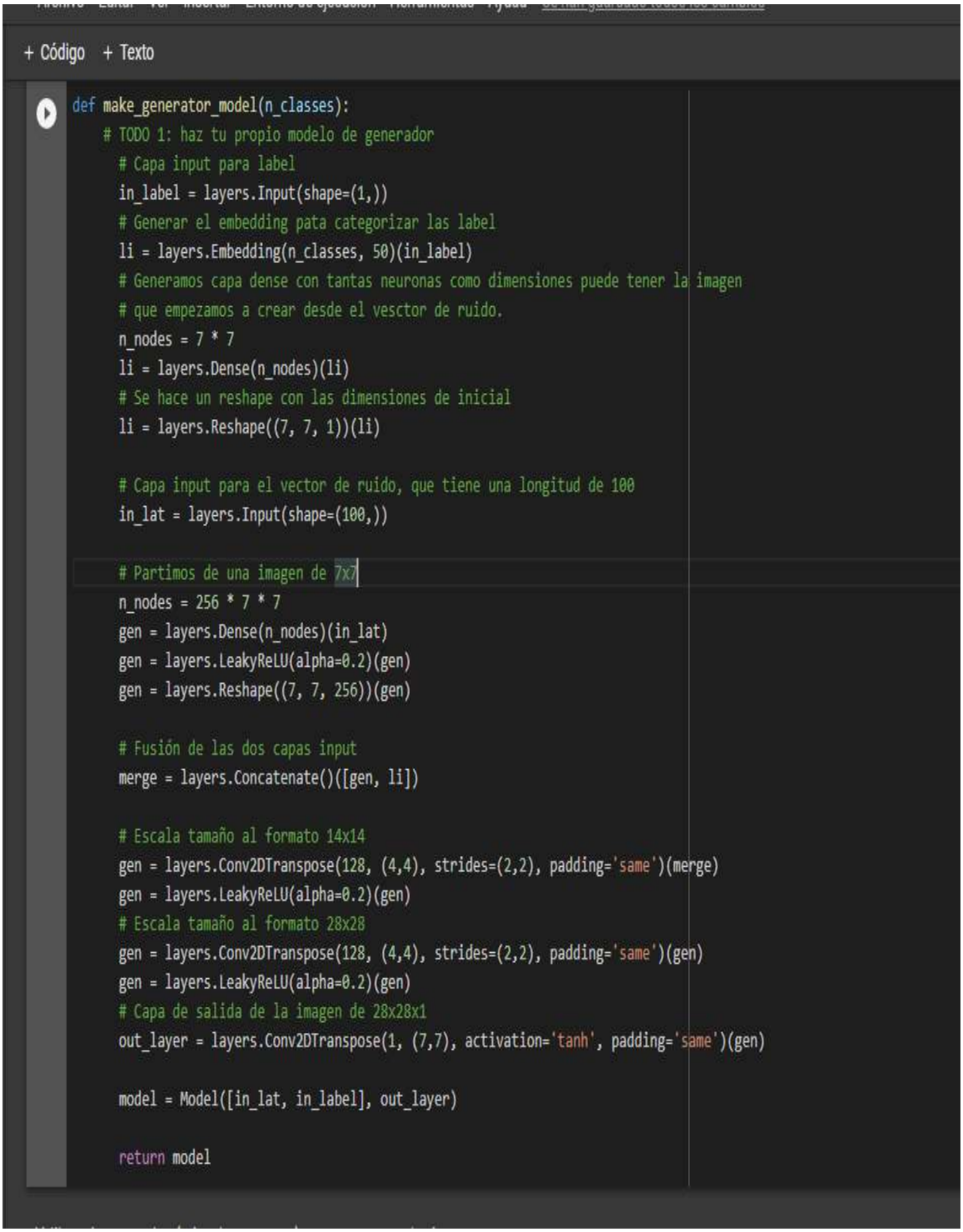
Una red GAN condicionada o CGAN es una red GAN al que se le condiciona la generación de imágenes con la utilización de etiquetas de clases. De esta forma se puede acotar la generación de las imágenes por parte del generador y centrar se en una tipología de imágenes.



4.2. [0.5 pts] Crea tu propia red condicional GAN capaz de generar muestras de la base de datos Fashion MNIST

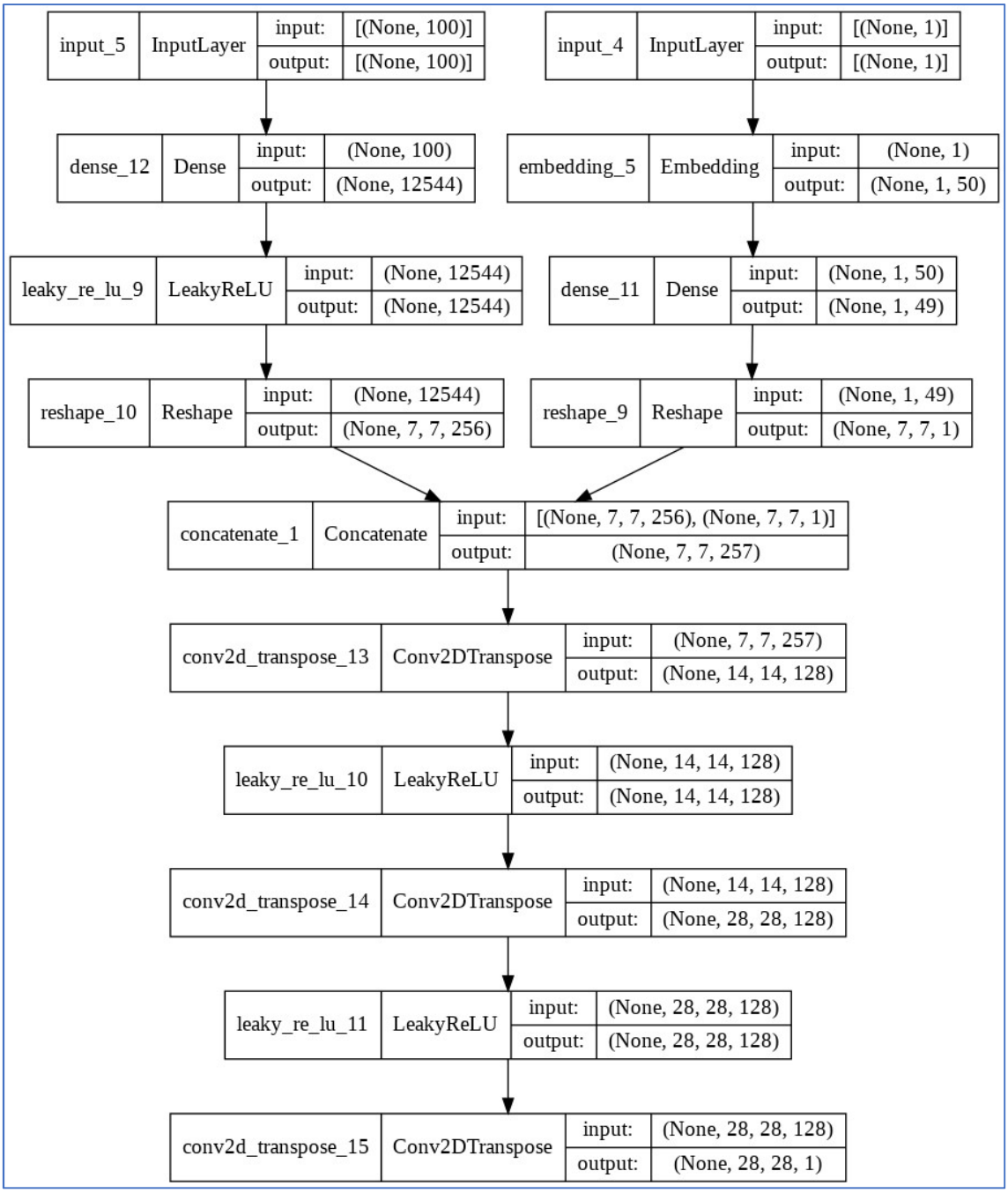
Respuesta: El planteamiento general es tratar las label como un input más del generador y discriminador, con lo que el planteamiento es de redes multinput en el que en un momento se fusionan en una entrada común. Seguidamente se muestra el código y la arquitectura generada.

Generador:



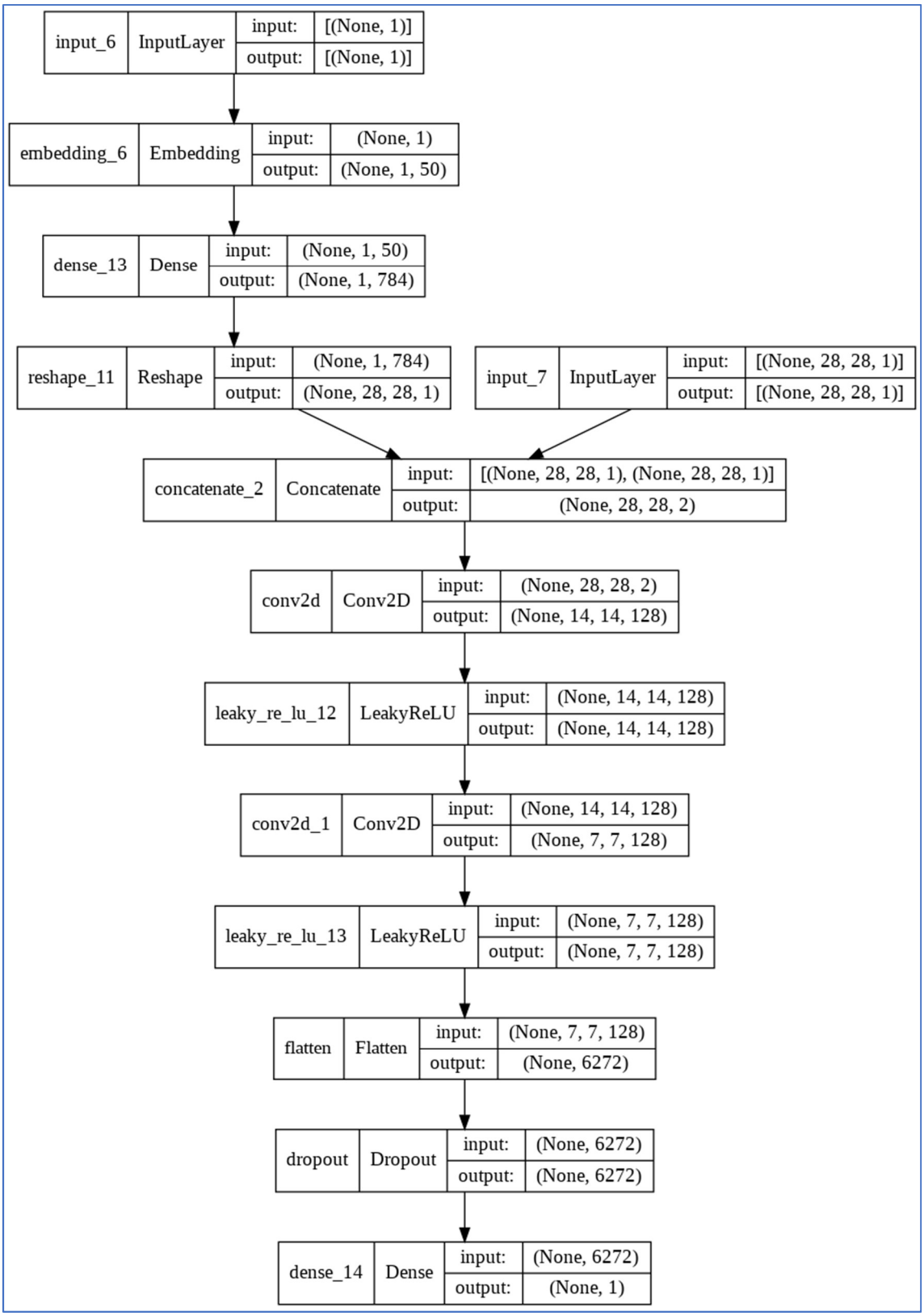
+ Código + Texto

```
def make_generator_model(n_classes):
    # TODO 1: haz tu propio modelo de generador
      # Capa input para label
      in_label = layers.Input(shape=(1,))
      # Generar el embedding pata categorizar las label
      li = layers.Embedding(n_classes, 50)(in_label)
      # Generamos capa dense con tantas neuronas como dimensiones puede tener la imagen
      # que empezamos a crear desde el vesctor de ruido.
      n_nodes = 7 * 7
      li = layers.Dense(n_nodes)(li)
      # Se hace un reshape con las dimensiones de inicial
      li = layers.Reshape((7, 7, 1))(li)

      # Capa input para el vector de ruido, que tiene una longitud de 100
      in_lat = layers.Input(shape=(100,))

      # Partimos de una imagen de 7x7
      n_nodes = 256 * 7 * 7
      gen = layers.Dense(n_nodes)(in_lat)
      gen = layers.LeakyReLU(alpha=0.2)(gen)
      gen = layers.Reshape((7, 7, 256))(gen)

      # Fusión de las dos capas input
      merge = layers.Concatenate()([gen, li])

      # Escala tamaño al formato 14x14
      gen = layers.Conv2DTranspose(128, (4,4), strides=(2,2), padding='same')(merge)
      gen = layers.LeakyReLU(alpha=0.2)(gen)
      # Escala tamaño al formato 28x28
      gen = layers.Conv2DTranspose(128, (4,4), strides=(2,2), padding='same')(gen)
      gen = layers.LeakyReLU(alpha=0.2)(gen)
      # Capa de salida de la imagen de 28x28x1
      out_layer = layers.Conv2DTranspose(1, (7,7), activation='tanh', padding='same')(gen)

      model = Model([in_lat, in_label], out_layer)

      return model
```


input_5 InputLayer input: [(None, 100)] output: [(None, 100)]
input_4 InputLayer input: [(None, 1)] output: [(None, 1)]
dense_12 Dense input: (None, 100) output: (None, 12544)
embedding_5 Embedding input: (None, 1) output: (None, 1, 50)
leaky_re_lu_9 LeakyReLU input: (None, 12544) output: (None, 12544)
dense_11 Dense input: (None, 1, 50) output: (None, 1, 49)
reshape_10 Reshape input: (None, 12544) output: (None, 7, 7, 256)
reshape_9 Reshape input: (None, 1, 49) output: (None, 7, 7, 1)
concatenate_1 Concatenate input: [(None, 7, 7, 256), (None, 7, 7, 1)] output: (None, 7, 7, 257)
conv2d_transpose_13 Conv2DTranspose input: (None, 7, 7, 257) output: (None, 14, 14, 128)
leaky_re_lu_10 LeakyReLU input: (None, 14, 14, 128) output: (None, 14, 14, 128)
conv2d_transpose_14 Conv2DTranspose input: (None, 14, 14, 128) output: (None, 28, 28, 128)
leaky_re_lu_11 LeakyReLU input: (None, 28, 28, 128) output: (None, 28, 28, 128)
conv2d_transpose_15 Conv2DTranspose input: (None, 28, 28, 128) output: (None, 28, 28, 1)

Discriminador:

```python
def make_discriminator_model(n_classes):

    #TODO: Haz tu propio discriminador
    # Capa input para label
    in_label = layers.Input(shape=(1,))
    # Generar el embedding pata categorizar las label
    li = layers.Embedding(n_classes, 50)(in_label)
    # Generamos capa dense con tasta neuroans como dimensiones puede tener la imagen
    n_nodes = 28*28*1
    li = layers.Dense(n_nodes)(li)
    # Se hace un reshape con las dimensiones de la imagen
    li = layers.Reshape((28,28, 1))(li)

    # Capa input para las imagenes
    in_image = layers.Input(shape=(28,28,1))

    # Capa que concatena los dos inputs
    merge = layers.Concatenate()([in_image, li])

    fe = layers.Conv2D(128, (3,3), strides=(2,2), padding='same')(merge)
    fe = layers.LeakyReLU(alpha=0.2)(fe)

    fe = layers.Conv2D(128, (3,3), strides=(2,2), padding='same')(fe)
    fe = layers.LeakyReLU(alpha=0.2)(fe)

    # Aplana el mapa de características
    fe = layers.Flatten()(fe)

    fe = layers.Dropout(0.4)(fe)

    # Capa Dense de 1 neurona para hacer la discriminación Real o Fake
    out_layer = layers.Dense(1)(fe)

    model = Model([in_image, in_label], out_layer)

    return model
```


input_6 InputLayer input: [(None, 1)] output: [(None, 1)]
embedding_6 Embedding input: (None, 1) output: (None, 1, 50)
dense_13 Dense input: (None, 1, 50) output: (None, 1, 784)
reshape_11 Reshape input: (None, 1, 784) output: (None, 28, 28, 1)
input_7 InputLayer input: [(None, 28, 28, 1)] output: [(None, 28, 28, 1)]
concatenate_2 Concatenate input: [(None, 28, 28, 1), (None, 28, 28, 1)] output: (None, 28, 28, 2)
conv2d Conv2D input: (None, 28, 28, 2) output: (None, 14, 14, 128)
leaky_re_lu_12 LeakyReLU input: (None, 14, 14, 128) output: (None, 14, 14, 128)
conv2d_1 Conv2D input: (None, 14, 14, 128) output: (None, 7, 7, 128)
leaky_re_lu_13 LeakyReLU input: (None, 7, 7, 128) output: (None, 7, 7, 128)
flatten Flatten input: (None, 7, 7, 128) output: (None, 6272)
dropout Dropout input: (None, 6272) output: (None, 6272)
dense_14 Dense input: (None, 6272) output: (None, 1)