UE21CS343BB2
Topics in Deep Learning
MNSIT Dataset Application

Dr. Shylaja S S
Director of Cloud Computing & Big Data (CCBD), Centre for Data Sciences & Applied Machine Learning (CDSAML)
Department of Computer Science and Engineering
shylaja.sharath@pes.edu

Ack: Divya K,
Teaching Assistant

Introduction

- Let us now look at an MNIST dataset application with DCGANs that involves training MNIST dataset with DCGANs to generate realistic-looking images of handwritten digits.

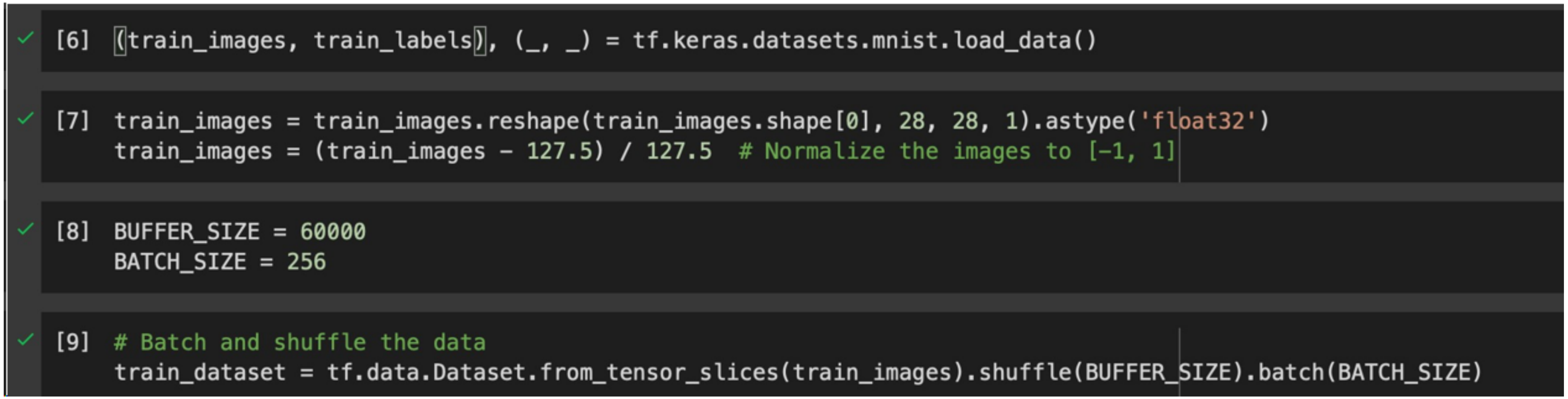
- MNIST is a popular dataset in machine learning and contains a large number of grayscale images of handwritten digits (0-9) from 0 to 9.

- During training, the generator tries to produce realistic images to fool the discriminator, while the discriminator tries to distinguish between real and fake images. Over time, through adversarial training, both networks improve their performance. Eventually, the generator learns to generate images that are indistinguishable from real handwritten digits in the MNIST dataset.

Topics in Deep Learning

DCGAN - Load and prepare the dataset

- We load the MNIST dataset, normalize the images and create the training dataset.



```
✓ [6] (train_images, train_labels), (_, _) = tf.keras.datasets.mnist.load_data()

✓ [7] train_images = train_images.reshape(train_images.shape[0], 28, 28, 1).astype('float32')
      train_images = (train_images - 127.5) / 127.5  # Normalize the images to [-1, 1]

✓ [8] BUFFER_SIZE = 60000
      BATCH_SIZE = 256

✓ [9] # Batch and shuffle the data
      train_dataset = tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
```

DCGAN - Creation of generator

- We will pass some random noise to the Generator. The random noise will be upscaled using Conv2DTranspose. Conv2DTranspose is used to transform a vector going in the opposite direction of a normal convolution.

- Use LeakyReLU for the layers except for the output layer. For the output layer, we use “tanh” activation function. LeakyReLU allows the gradients to flow better through the model architecture.

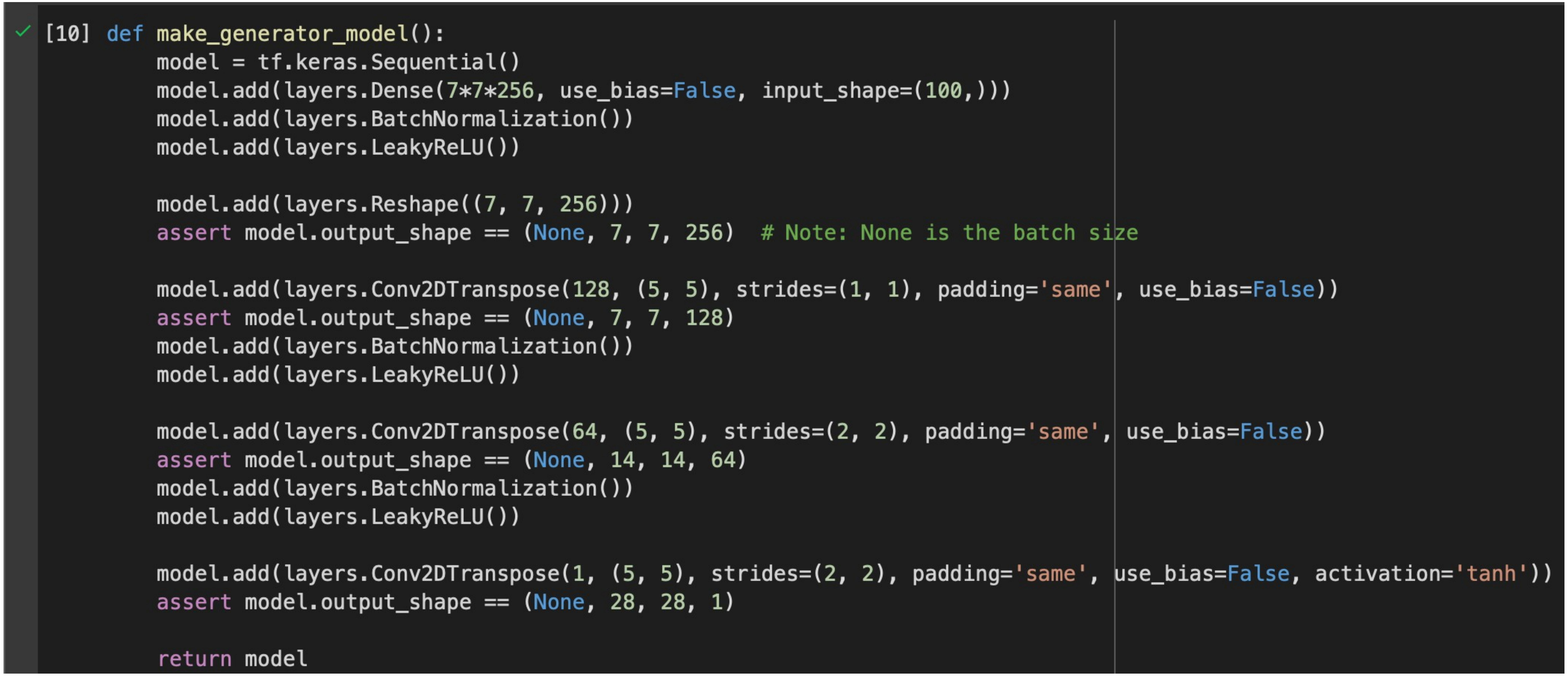
- Why Leaky ReLU? The ReLU activation function has a value between 0 and 1. When a positive input is passed to the ReLU or LeakyReLU the output will be a positive value however, when a negative input is passed to ReLU it will output 0, but Leaky ReLU will output a controlled negative value.

- tanh is used in the final output layer instead of sigmoid to ensure that gradients are maintained between -1 and 1 and are not set to zero as that stops the learning for the model.

- Batch Normalization is used for faster convergence as it standardizes the data to have a stabilizing impact on the training process.

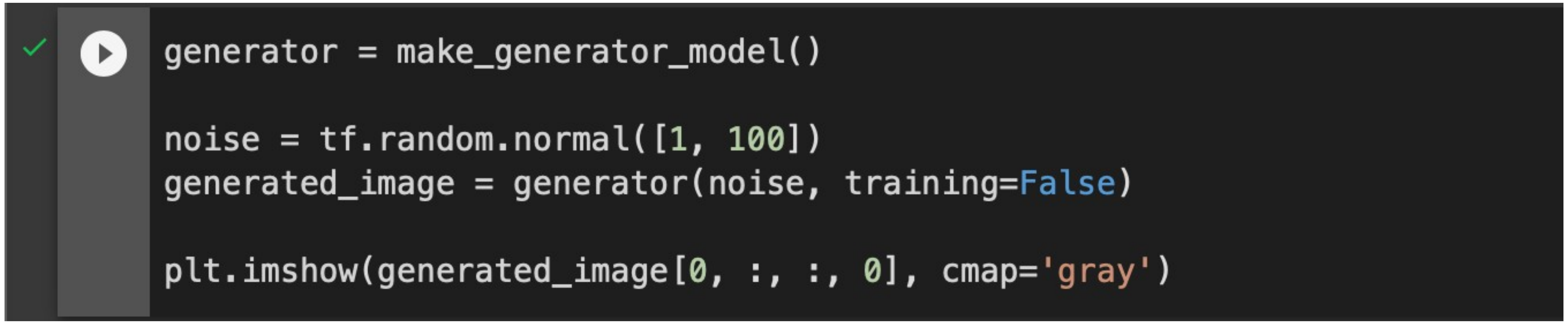
Topics in Deep Learning

DCGAN - Generator code



```
[10] 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
```

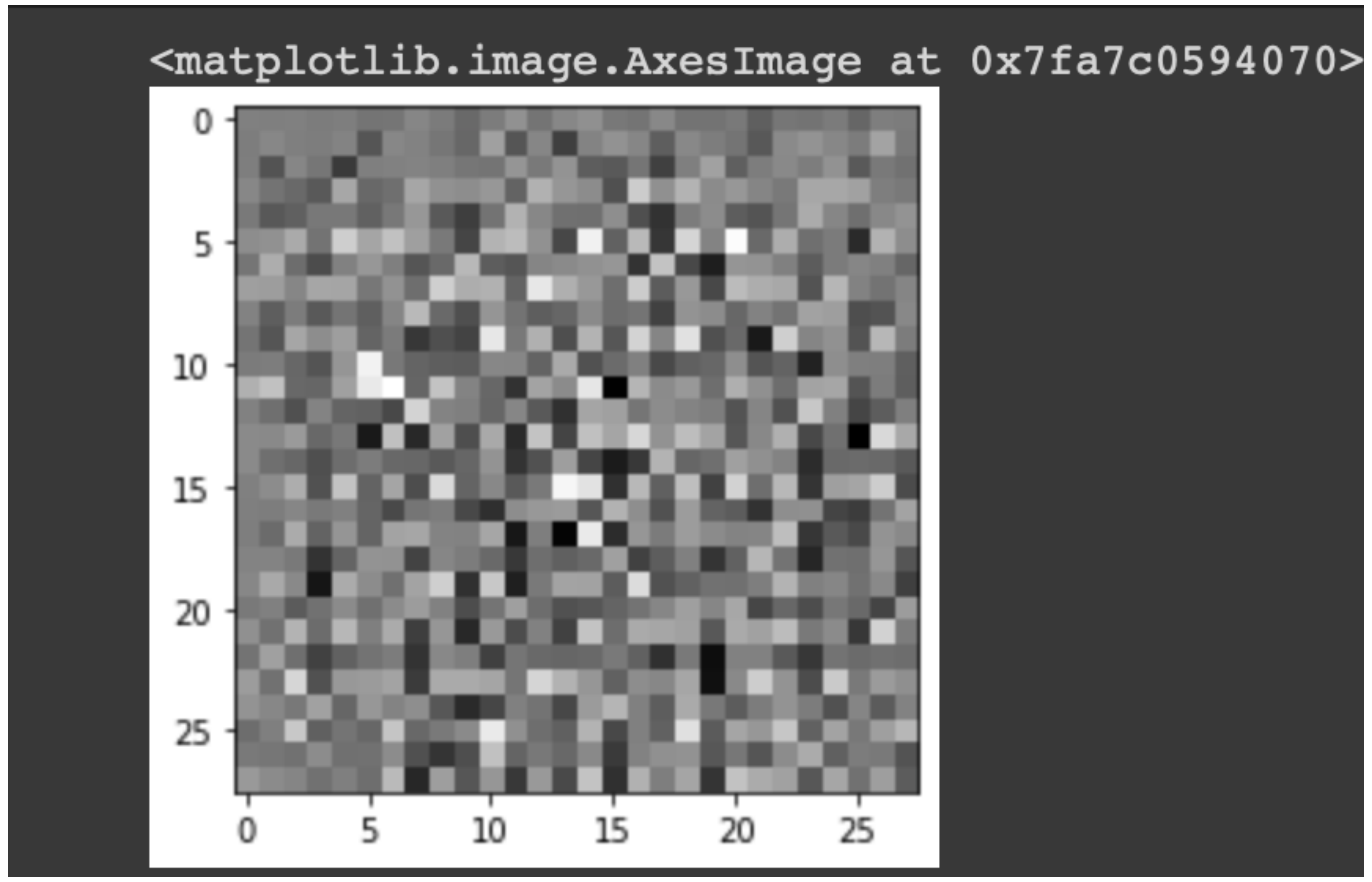
Topics in Deep Learning
DCGAN - Viewing an image by an untrained generator



```
generator = make_generator_model()

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

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

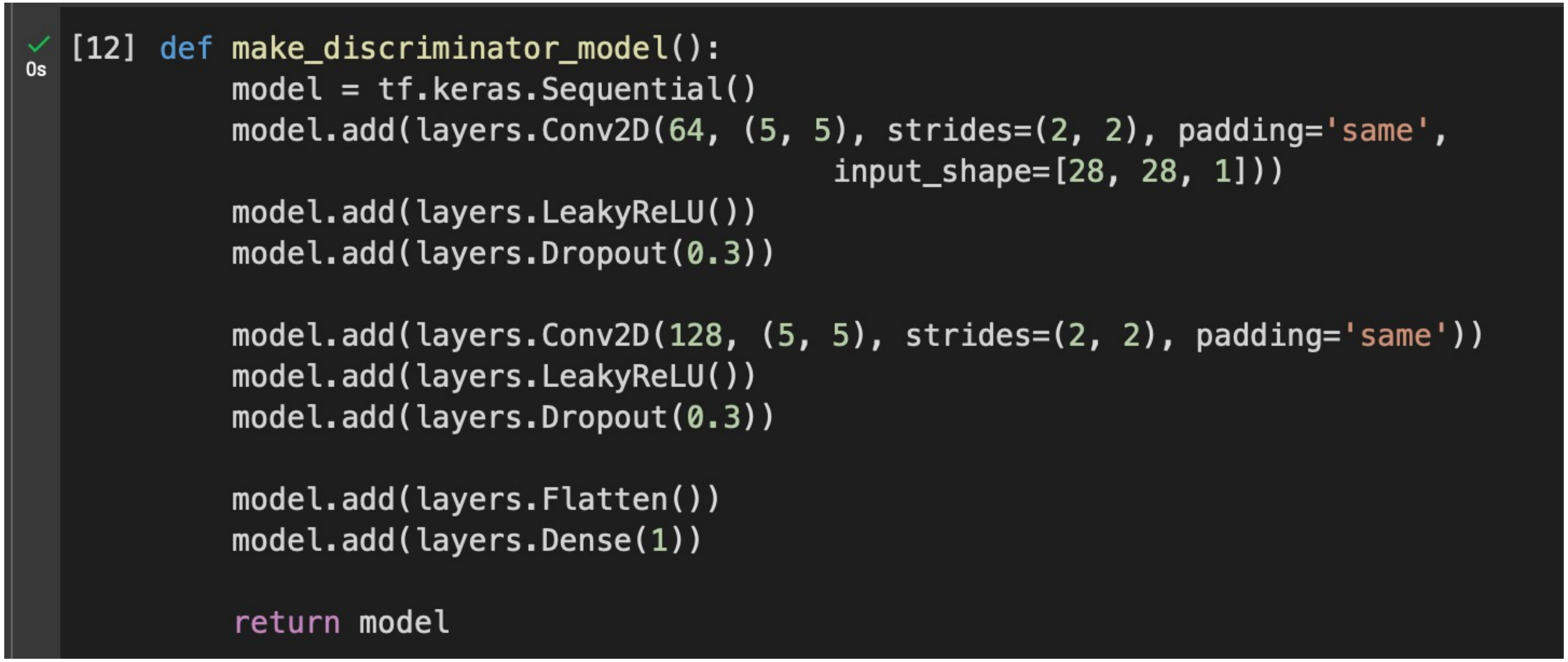
- If we use the (as yet untrained) generator to create an image.

```
<matplotlib.image.AxesImage at 0x7fa7c0594070>
```



DCGAN - Creation of the Discriminator

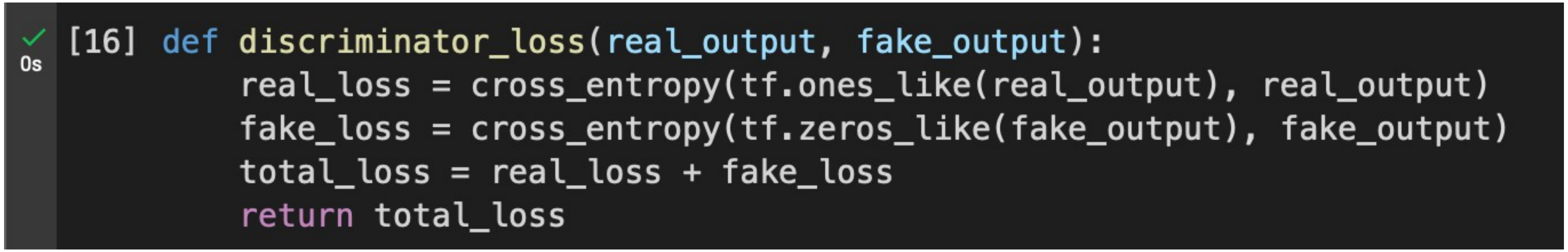
- The Discriminator is a CNN-based image classifier that classifies between the real and the fake image. It takes fake images generated from the Generator using random noise and the real images from the training dataset as input to classify.



```
✓  [12] def make_discriminator_model():
0s          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
```

DCGAN - Loss and optimisers - Discriminator loss

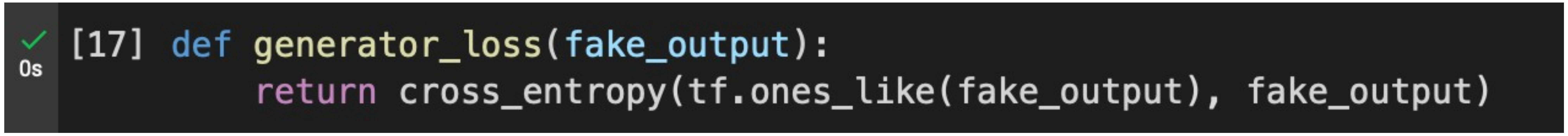
- This method quantifies how well the discriminator is able to distinguish real images from fakes. It compares the discriminator's predictions on real images to an array of 1s, and the discriminator's predictions on fake (generated) images to an array of 0s. Thus the Discriminator loss is a sum of loss functions for the distribution of the data for both fake as well as real data.



```
0s [16] 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
```

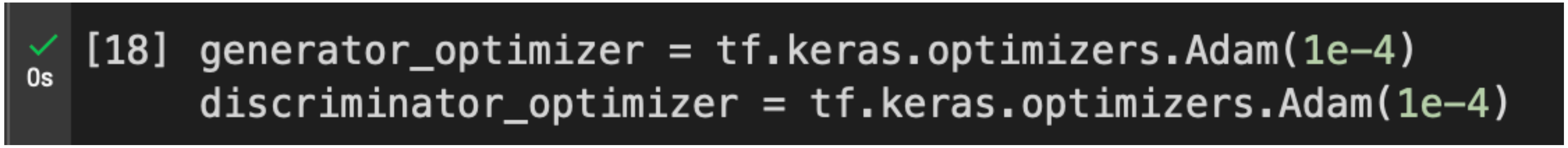
DCGAN - Loss and optimisers - Generator loss & optimisers

- The generator's loss quantifies how well it was able to trick the discriminator. Intuitively, if the generator is performing well, the discriminator will classify the fake images as real (or 1). Here, compare the discriminators' decisions on the generated images to an array of 1s.



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

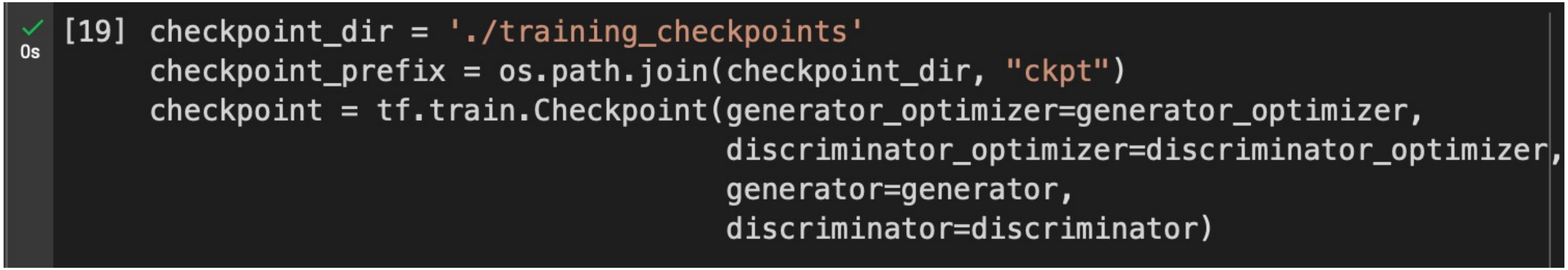
- The loss function used in DCGAN is typically the binary cross-entropy loss, which measures the difference between the predicted probability and the true label (0 for generated images, 1 for real images). The discriminator and the generator optimizers are different since you will train two networks separately.



```
[18] generator_optimizer = tf.keras.optimizers.Adam(1e-4)
     discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
```

DCGAN - Use of checkpoints

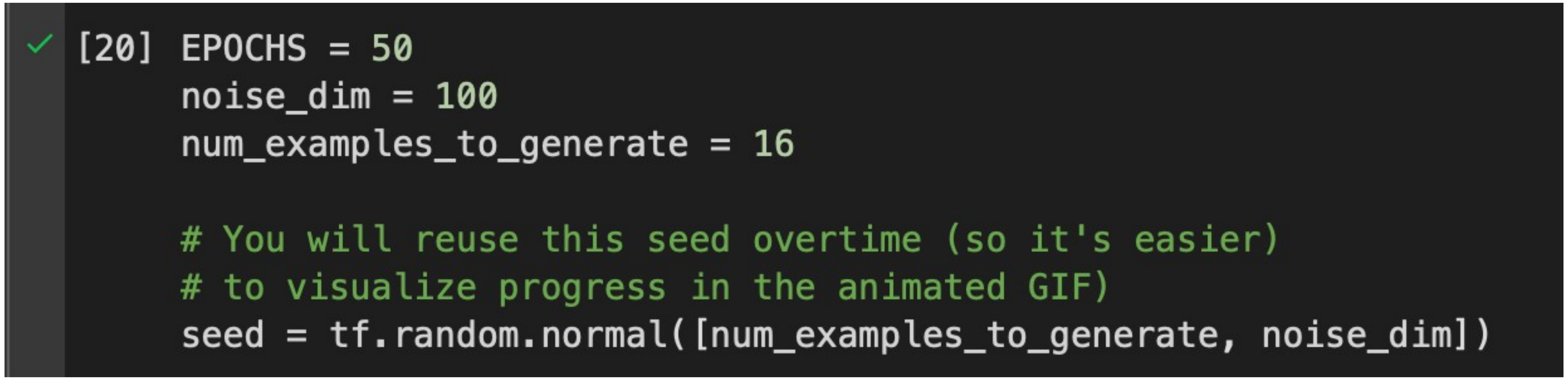
- Here we use checkpoints to save and restore models, which can be helpful in case, a long-running training task is interrupted.



```
[19] checkpoint_dir = './training_checkpoints'
     checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
     checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
                                      discriminator_optimizer=discriminator_optimizer,
                                      generator=generator,
                                      discriminator=discriminator)
```

DCGAN - Training Loop

- The training loop begins with the generator receiving a random seed as input. That seed is used to produce an image. The discriminator is then used to classify real images (drawn from the training set) and fake images (produced by the generator).



```
[20] EPOCHS = 50
     noise_dim = 100
     num_examples_to_generate = 16

     # You will reuse this seed overtime (so it's easier)
     # to visualize progress in the animated GIF)
     seed = tf.random.normal([num_examples_to_generate, noise_dim])
```

DCGAN - Training Loop Code

```
[21] # Notice the use of `tf.function`
     # This annotation causes the function to be "compiled".
     @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))
```

DCGAN - Training Loop Code

- Overall, the following code represents one iteration of the training loop for a DCGAN model on the MNIST dataset. During each iteration, a batch of real images is passed through the discriminator, and a batch of random noise is passed through the generator to produce a batch of generated images.

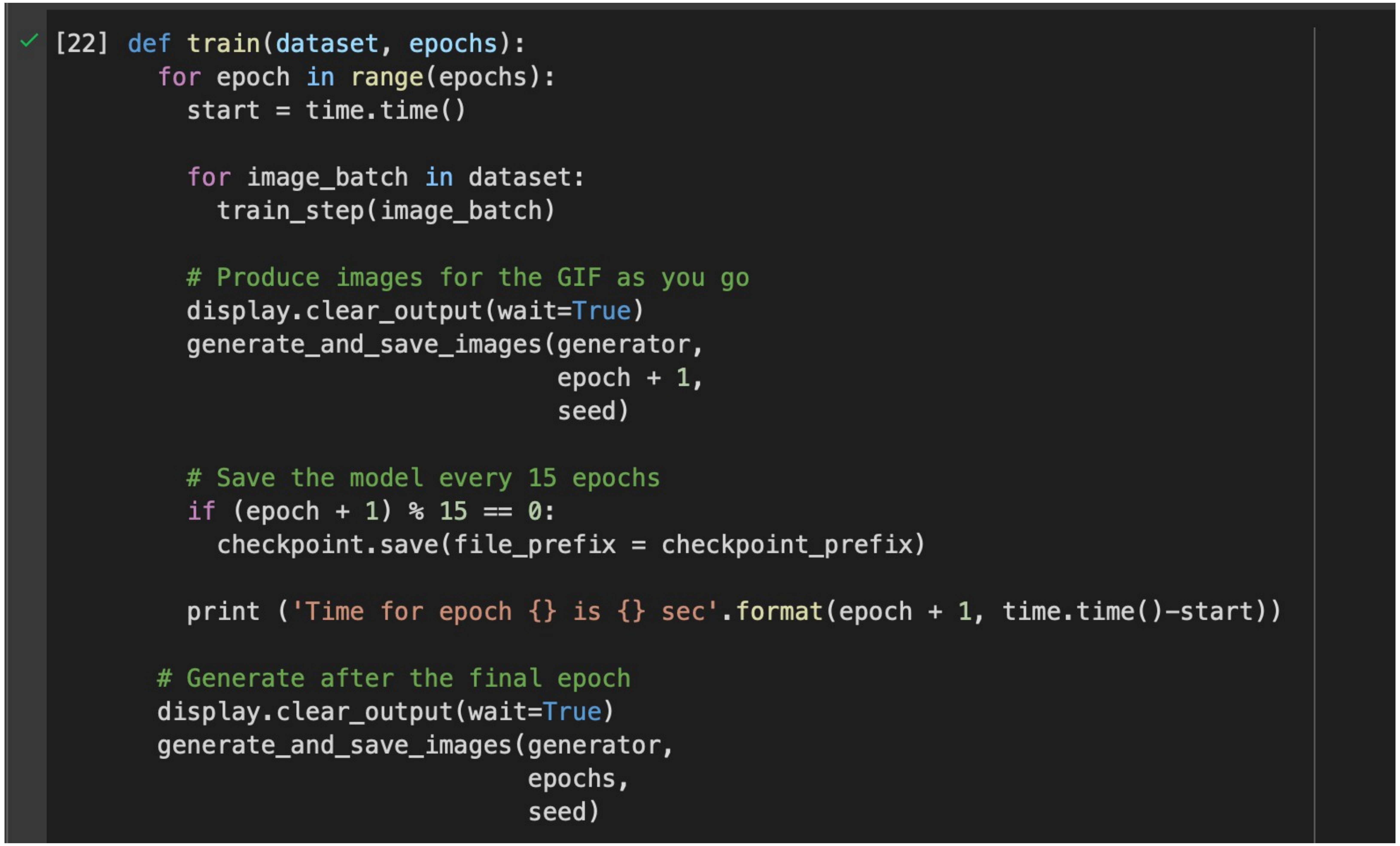
- Two gradient tapes are created, one for the generator and one for the discriminator. Gradient tapes are used to record gradients during the forward pass, which can be used to compute gradients for backpropagation during training. The real and generated images are passed through the discriminator to produce a batch of outputs.

- The losses for both the generator and discriminator are computed based on the discriminator's output for the real and generated images, and the gradients of the losses with respect to the trainable variables for each network are computed using gradient tapes.

- These gradients are then used to update the trainable variables of the generator and discriminator using the optimizer. The process is repeated for a specified number of iterations or epochs to train the model.

DCGAN - Train function code

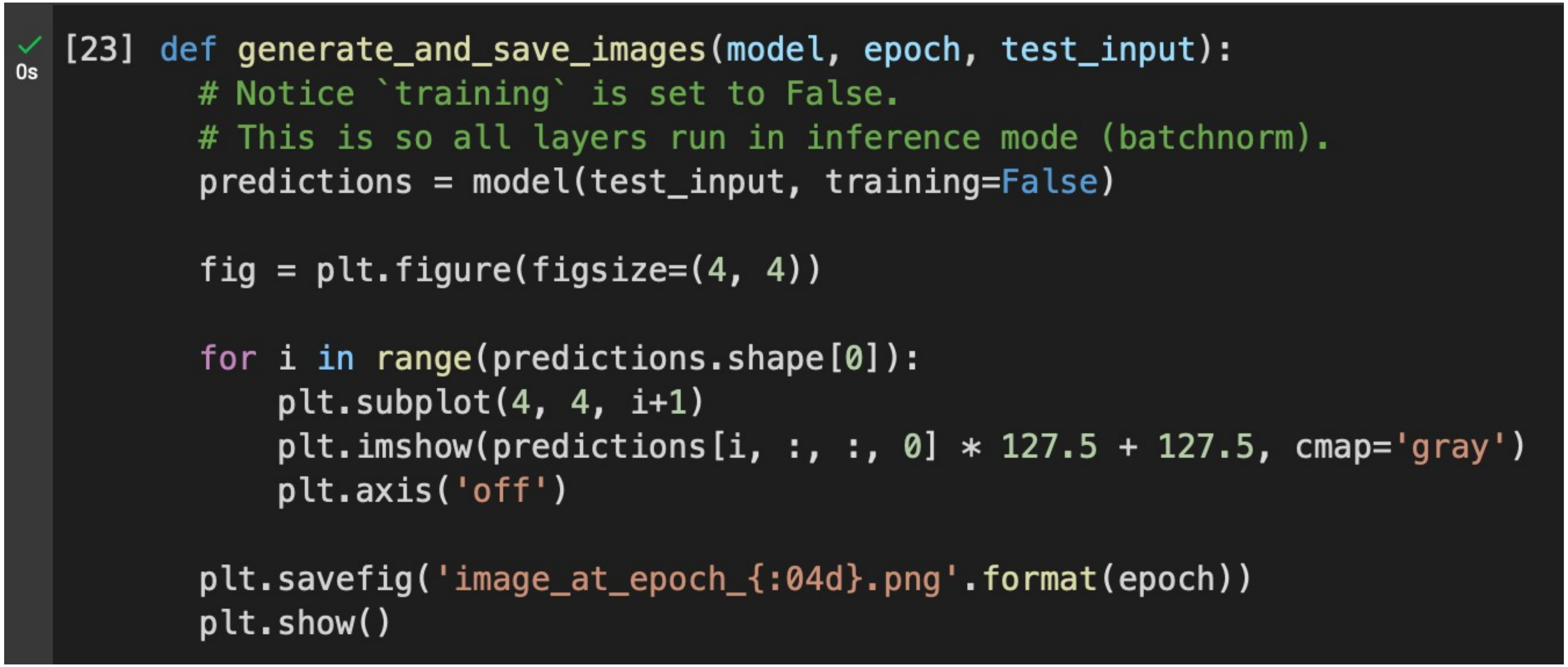
- Training of the DCGAN starts, checkpoint happens every 15 epochs and produces images for GIF as the process goes through.



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

         for image_batch in dataset:
           train_step(image_batch)

         # Produce images for the GIF as you go
         display.clear_output(wait=True)
         generate_and_save_images(generator,
                                  epoch + 1,
                                  seed)

         # Save the model every 15 epochs
         if (epoch + 1) % 15 == 0:
           checkpoint.save(file_prefix = checkpoint_prefix)

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

       # Generate after the final epoch
       display.clear_output(wait=True)
       generate_and_save_images(generator,
                                epochs,
                                seed)
```

Topics in Deep Learning
DCGAN - Generate and save images

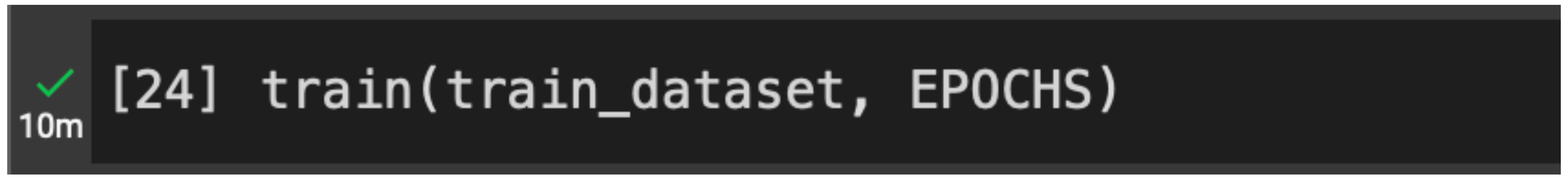
- The function displays the images generated by the model passed as a parameter.



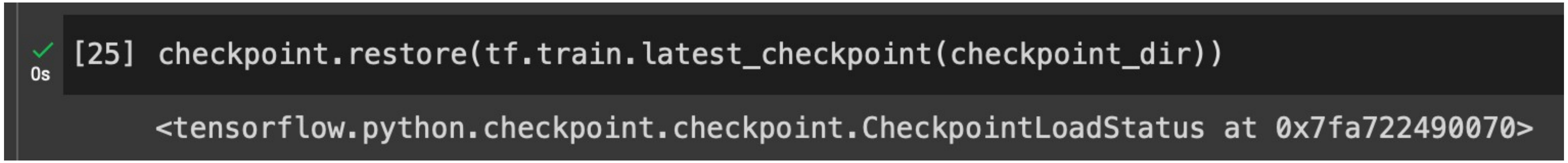
```
[23] def generate_and_save_images(model, epoch, test_input):
       # Notice `training` is set to False.
       # This is so all layers run in inference mode (batchnorm).
       predictions = model(test_input, training=False)

       fig = plt.figure(figsize=(4, 4))

       for i in range(predictions.shape[0]):
           plt.subplot(4, 4, i+1)
           plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5, cmap='gray')
           plt.axis('off')

       plt.savefig('image_at_epoch_{:04d}.png'.format(epoch))
       plt.show()
```

DCGAN - Train the model

- Call the train( ) method defined above to train the generator and discriminator simultaneously. Note, training GANs can be tricky. It's important that the generator and discriminator do not overpower each other (e.g., that they train at a similar rate).
- At the beginning of the training, the generated images look like random noise. As training progresses, the generated digits will look increasingly real. After about 50 epochs, they resemble MNIST digits. This may take about one minute/epoch with the default settings on Colab.
- Call the train function and restore the checkpoint.

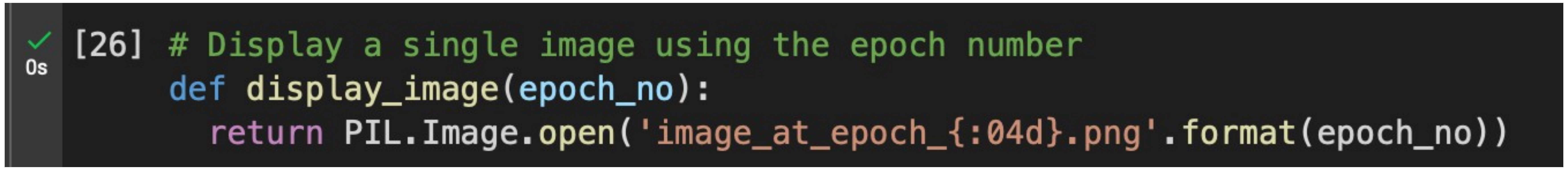


```
10m [24] train(train_dataset, EPOCHS)
```

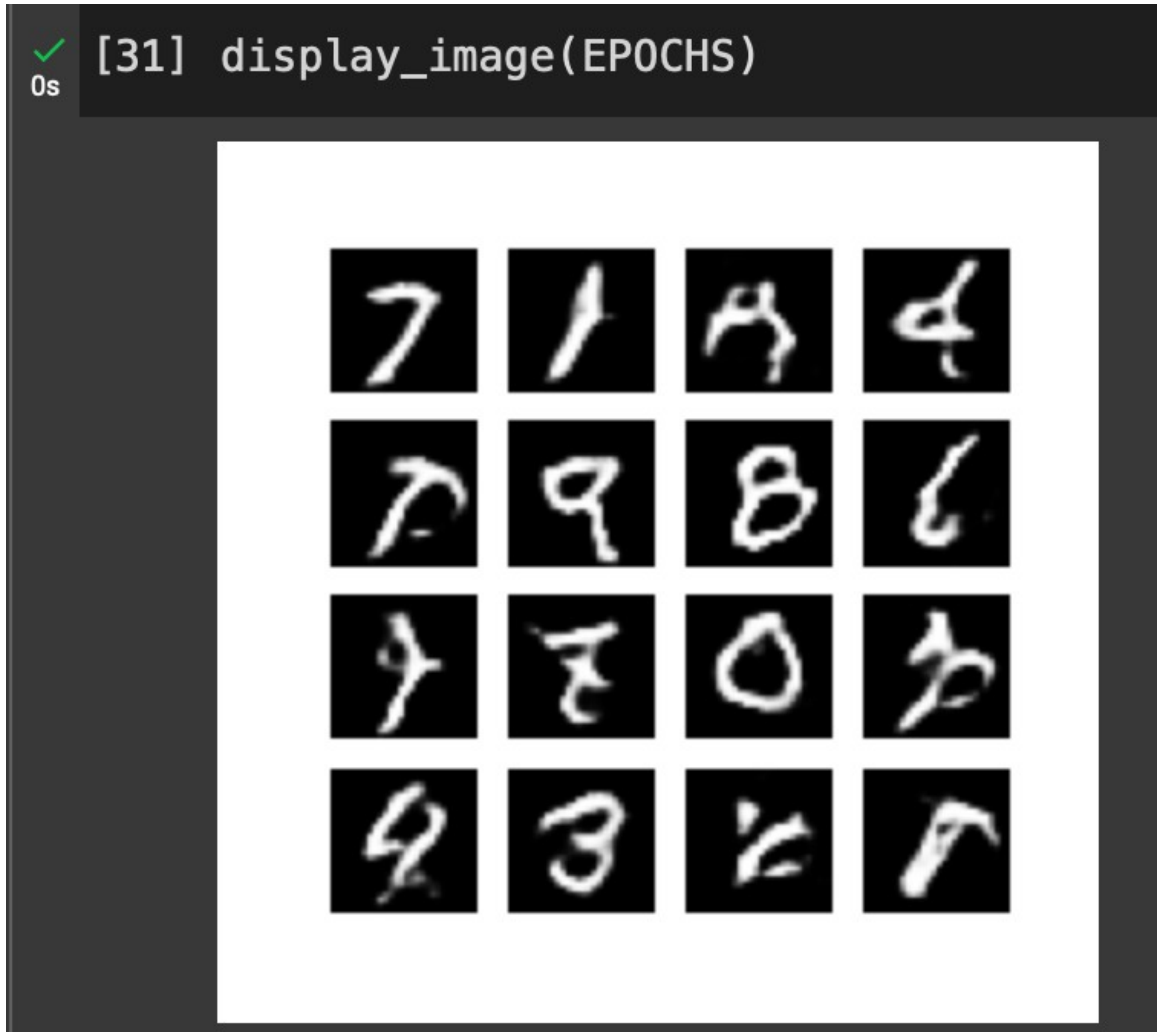


```
0s [25] checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))

        <tensorflow.python.checkpoint.checkpoint.CheckpointLoadStatus at 0x7fa722490070>
```

Topics in Deep Learning
DCGAN - Display image at final epoch



```
[26] # Display a single image using the epoch number
     def display_image(epoch_no):
       return PIL.Image.open('image_at_epoch_{:04d}.png'.format(epoch_no))
```



```
[31] display_image(EPOCHS)
```

Topics in Deep Learning
Link to View Entire Code

- Find the link to the entire code below:

https://drive.google.com/file/d/11ADG0TF6HK5DUAhwqxwW9Yl25MTpcibg/view?usp=sharing

Key differences/additions - DCGAN

Here is the summary of DCGAN:

- Replace all max pooling with convolutional stride
- Use transposed convolution for up sampling.
- Eliminate fully connected layers.
- Use Batch normalization except for the output layer for the generator and the input layer of the discriminator.
- Use ReLU in the generator except for the output which uses tanh.
- Both Generator and Discriminator do not use a Max pooling.
- The Generator uses Leaky Relu as the activation function for all layers except the output, and the Discriminator uses Leaky Relu for all layers.

DCGAN - Key problem and summary

- One of the key challenges in training DCGANs is achieving stable convergence. Because the generator and discriminator networks are trained simultaneously, they can sometimes become stuck in a cycle where the generator produces the same images repeatedly, and the discriminator learns to distinguish these images from the real images. To prevent this from happening, several techniques have been proposed, including batch normalization, weight initialization, and gradient penalty regularization.

- In summary, DCGAN is a type of generative model that uses deep convolutional neural networks to generate new images. By learning hierarchical representations of the image data, DCGANs can capture both low-level and high-level features, producing images that are visually similar to those in the training set. While DCGANs are powerful models, they also require careful tuning and attention to detail to achieve stable convergence and good-quality image generation

Acknowledgements

https://www.tensorflow.org/tutorials/generative/dcgan

https://machinelearningmastery.com/how-to-develop-a-generative-adversarial-network-for-an-mnist-handwritten-digits-from-scratch-in-keras/

THANK YOU

Dr. Shylaja S S
Director of Cloud Computing & Big Data (CCBD), Centre for Data Sciences & Applied Machine Learning (CDSAML)
Department of Computer Science and Engineering
shylaja.sharath@pes.edu

Ack: Divya K,
Teaching Assistant