Implementation Tutorial : Image Generation using GAN

Jin Myung Kwak
MLAI, KAIST
2020/07/27

Overview

In this tutorial, we will learn how to implement:

Image Generation using GAN

- Deep Convolutional Generative Adversarial Networks (***DCGAN***)

*Both the codes and the dataset for this tutorial will be provided by the instructor.

*The provided version may have been *slightly* modified from the original codes.

Environments

Prerequisites

- Linux or macOS
- Python >= 3.7
- Tensorflow = 2.2.0

Github Repositories

- git clone https://github.com/jin8/gan.git

Image Generation using GAN (***DCGAN***)

Download Tutorial

You can download the DCGAN code as below:

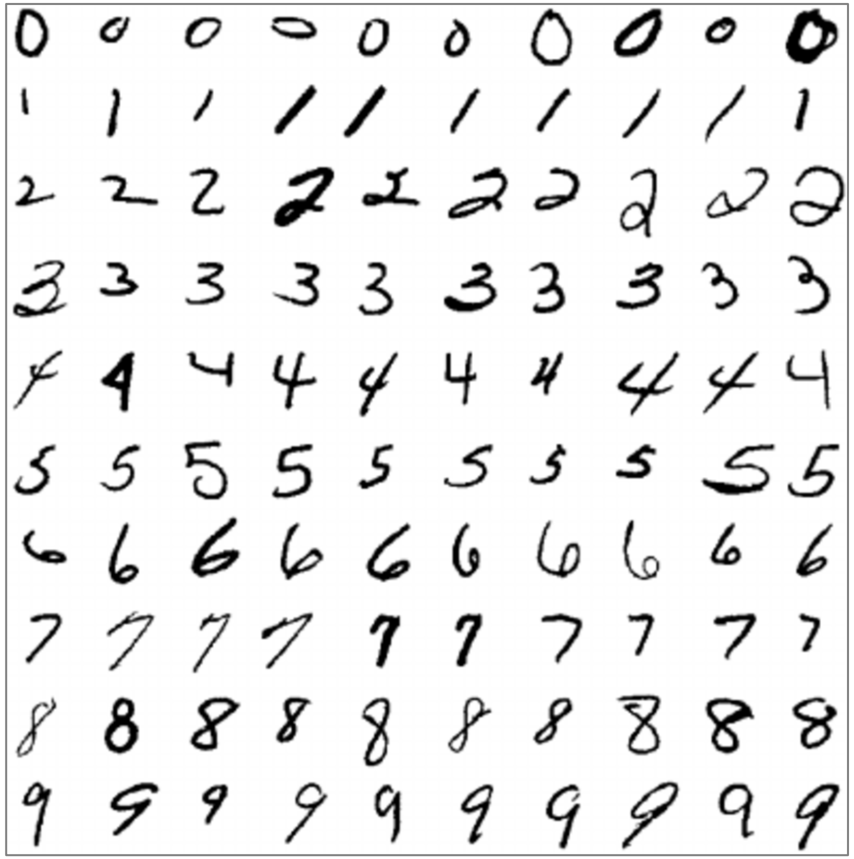
```
myung@ai2:/st2/myung/tutorial/$ git clone https://github.com/jin8/gan.git

Cloning into 'gan'...
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 9 (delta 1), reused 8 (delta 0), pack-reused 0
Unpacking objects: 100% (9/9), done.
```

This is what you will see after git clone.

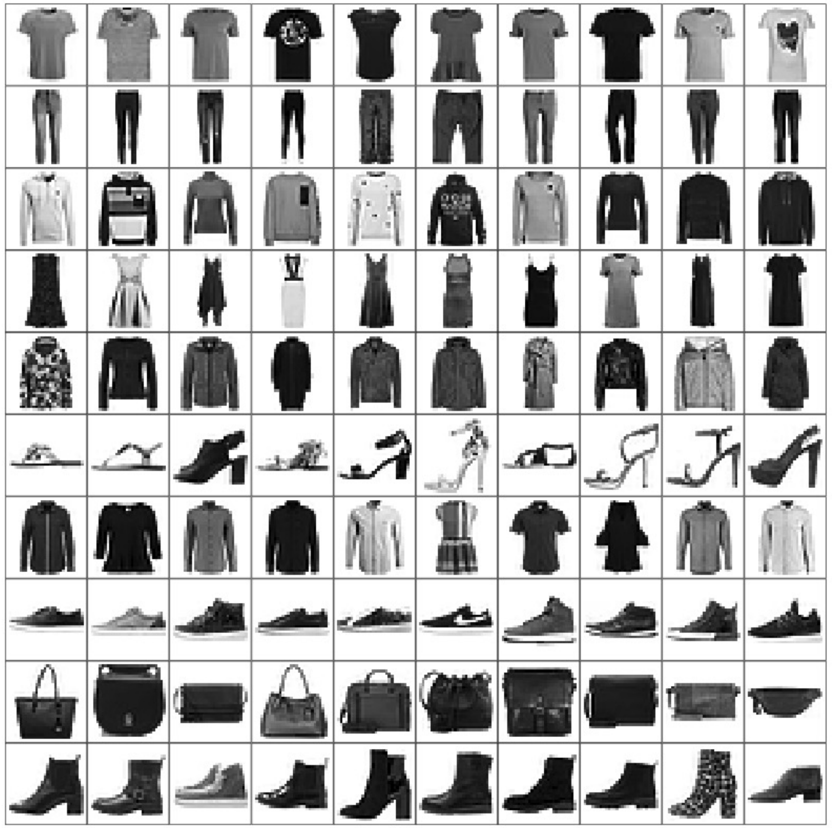
```
- gan/
  └dcgan.py
  └dcgan_sol.py
```

Datasets

Two popular image datasets



MNIST



Fashion MNIST

How to Execute the Code?

You can run the program with this command:

```
myung@ai2:/st2/myung/tutorial/gan$ python dcgan_sol.py --dataset mnist
```

Once you execute the code you will see new folders created in the directory

```
-gan/
 └ dcgan.py
 └ images/
    └ dcgan_sol_mnist/
       └ image_at_epoch_0.png
       └ image_at_epoch_1.png
       └ image_at_epoch_2.png
       └          ...
 └ logs
 └ dcgan_sol_mnist.gif
```

Code Structure

We need look no further than just 1 file: ***dcgan.py***

```
1. import packages

2. parser.add_argument()

3. Dataset

4. Models
   - def generator_model():
   - def discriminator_model():

5. Loss functions
   - def generator_loss():
   - def discriminator_loss():

6. Optimizer & Checkpoint

7. Training!
    for epoch in range(args.epoch):
      for i, data in enumerate(dataset):
        # train!
```

Code Structure

We need look no further than just 1 file: ***dcgan.py***

```
1. import packages

2. parser.add_argument()

3. Dataset

4. Models
   - def generator_model():
   - def discriminator_model():

5. Loss functions
   - def generator_loss():
   - def discriminator_loss():

6. Optimizer & Checkpoint

7. Training!
    for epoch in range(args.epoch):
      for i, data in enumerate(dataset):
        # train!
```

1. Import Packages

This are the following packages used to train the GAN

```
import os
import time
import argparse
import tensorflow as tf
from tensorflow.keras import layers
from utils import generate_and_save_images, generate_and_save_gif

# assign specific gpu
os.environ["CUDA_VISIBLE_DEVICES"] = "#"
```

→ We have installed all the packages you'll need for this project

→ Please refer environment.yml if you want to check which packages are used for this tutorial

→ Double check your `CUDA_VISIBLE_DEVICES` number

2. Define Argparser

```
parser = argparse.ArgumentParser()
parser.add_argument('--dataset', required=True, help='mnist | fashion')
              ...
parser.add_argument('--image_dir', type=str, default='./images’,
              help='directory for output images')
parser.add_argument('--log_dir', type=str, default='./logs’,
              help='directory for logging losses using tensorboard’)
              ...
parser.add_argument('--epochs', type=int, default=100,
              help='training epochs')
parser.add_argument('--batch_size', type=int, default=64, help='input batch size')
parser.add_argument('--latent_size', type=int, default=100,
              help='size of the latent z vector’)
```

We can assign parameters in command line:

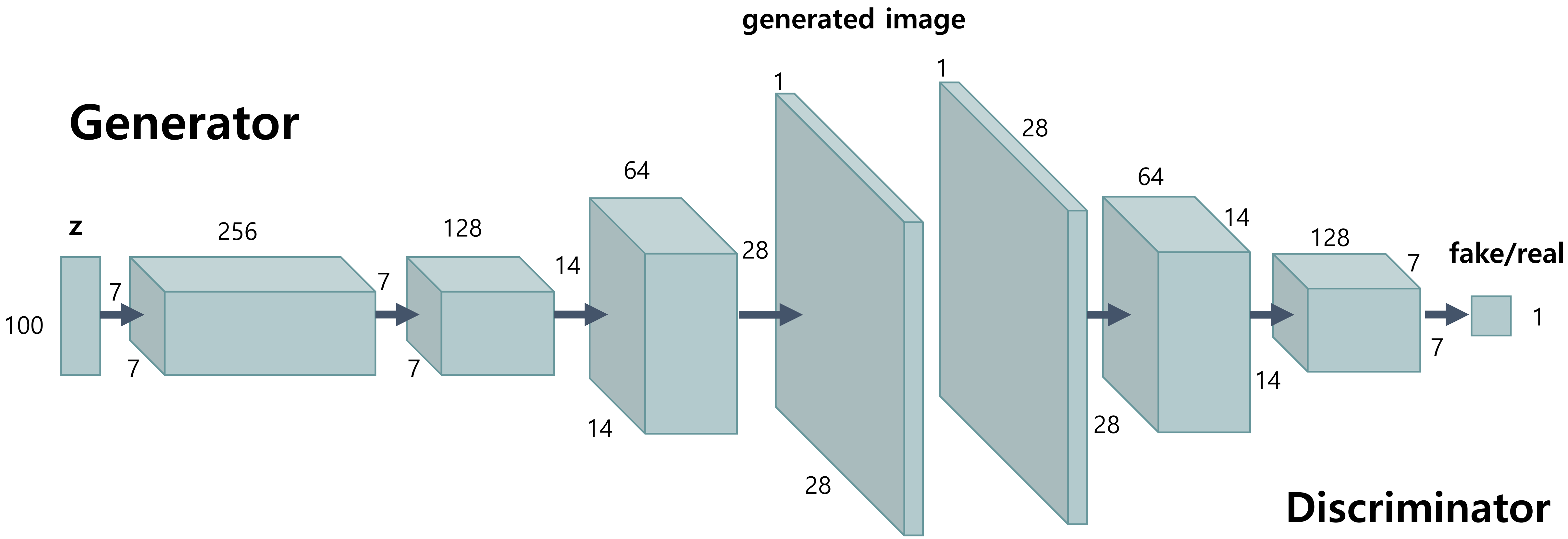
```
myung@ai2:/st2/myung/tutorial/gan$ python dcgan.py --dataset celeba
```

3. Dataset Loading & Preprocessing

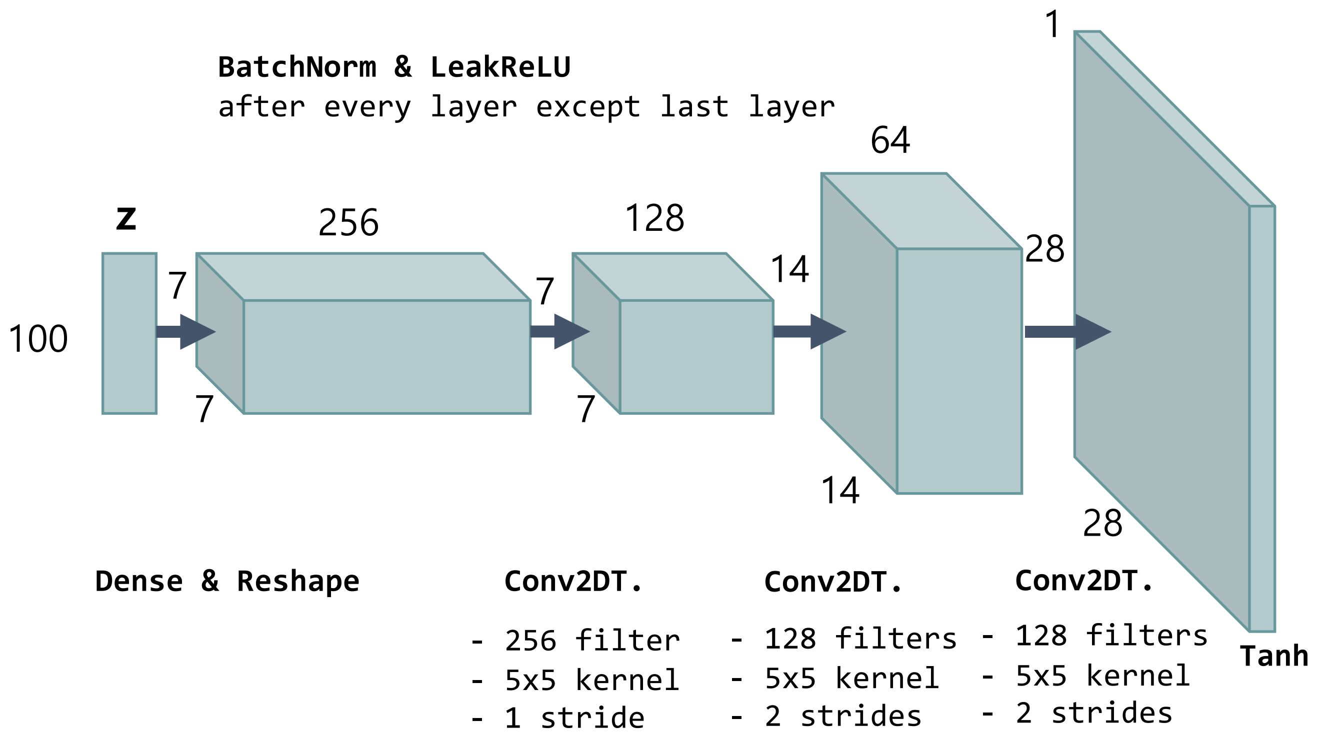
Datasets are load depending on the `args.dataset`

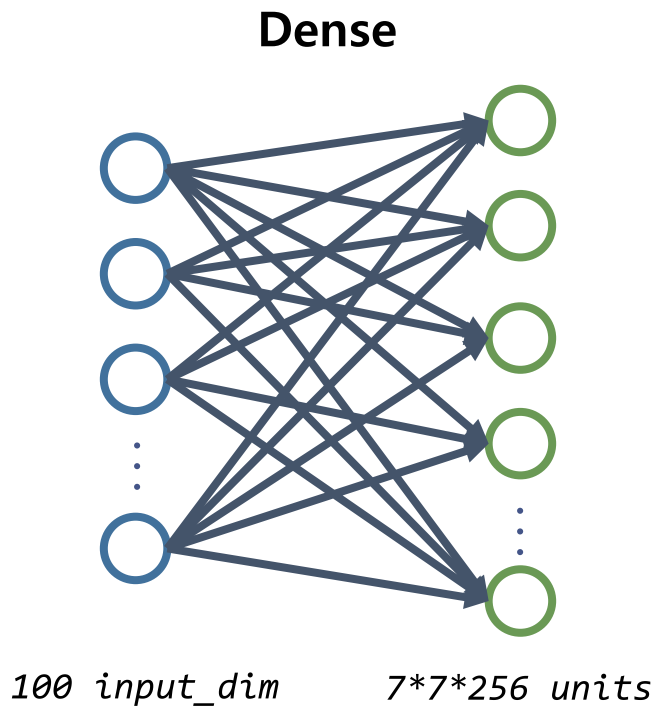
```
if args.dataset == 'mnist':
    (train_images, train_labels), (_, _) = tf.keras.datasets.mnist.load_data()

elif args.dataset == 'fashion':
    (train_images, train_labels), (_, _) = tf.keras.datasets.fashion_mnist.load_data()
    ...

train_images = train_images.reshape(
                      train_images.shape[0],
                      IMAGE_SIZE, IMAGE_SIZE,
                      IMAGE_CHANNEL).astype('float32')
train_images = (train_images - 127.5) / 127.5
train_dataset = tf.data.Dataset.from_tensor_slices(train_images)\
                      .shuffle(train_images.shape[0], reshuffle_each_iteration=True)\
                      .batch(BATCH_SIZE)
```

4. Model : Deep Convolutional GAN (*DCGAN*)



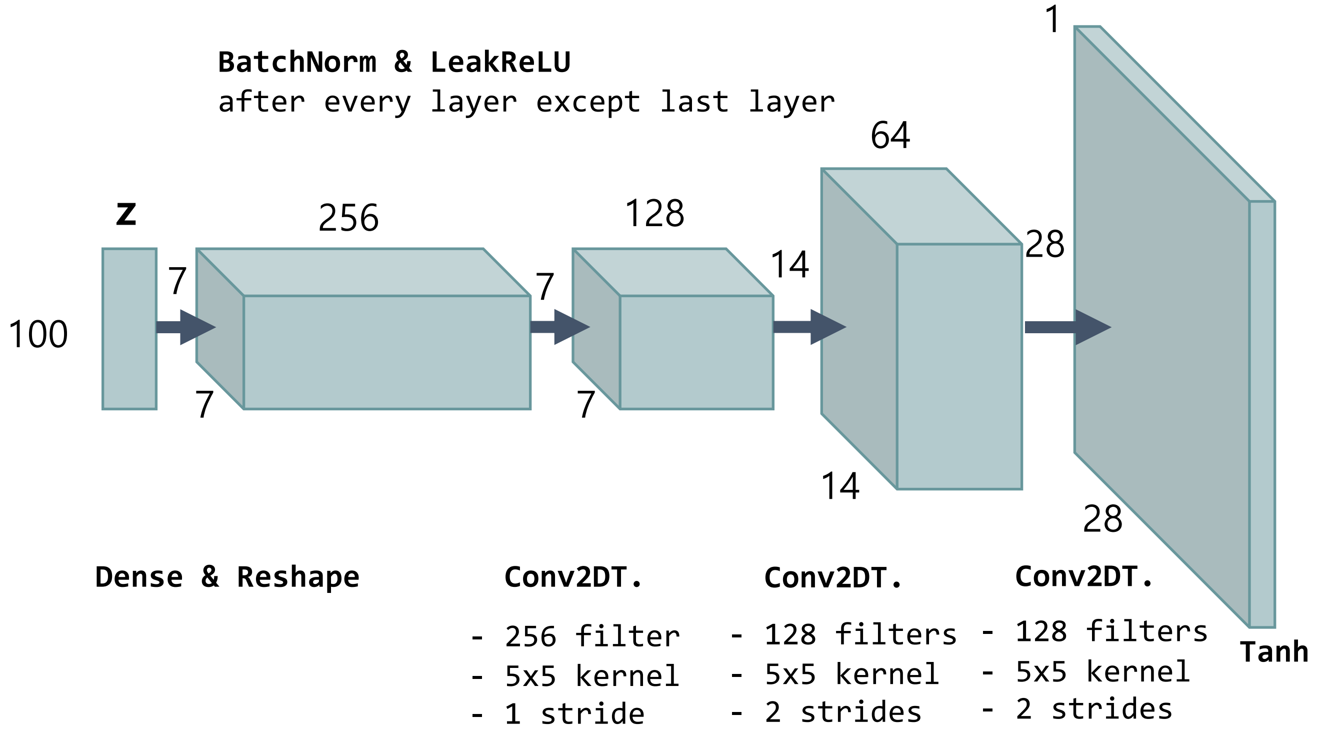
4. Implement **def generator_model():**

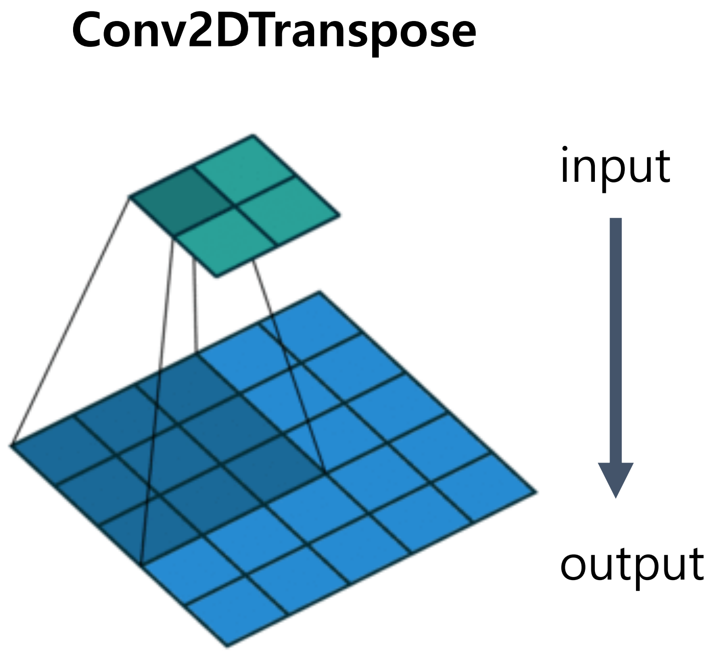




- layers.**Dense**(*units*, use_bias=False, *input_shape=(input_dim,)*)
- layers.**Reshape**(*target_shape*)
- layers.**Conv2DTranspose**(*filters*, *kernel_size*, *strides=(1, 1)*, padding='same', use_bias=False)
- layers.**BatchNormalization()**
- layers.**LeakyReLU()**

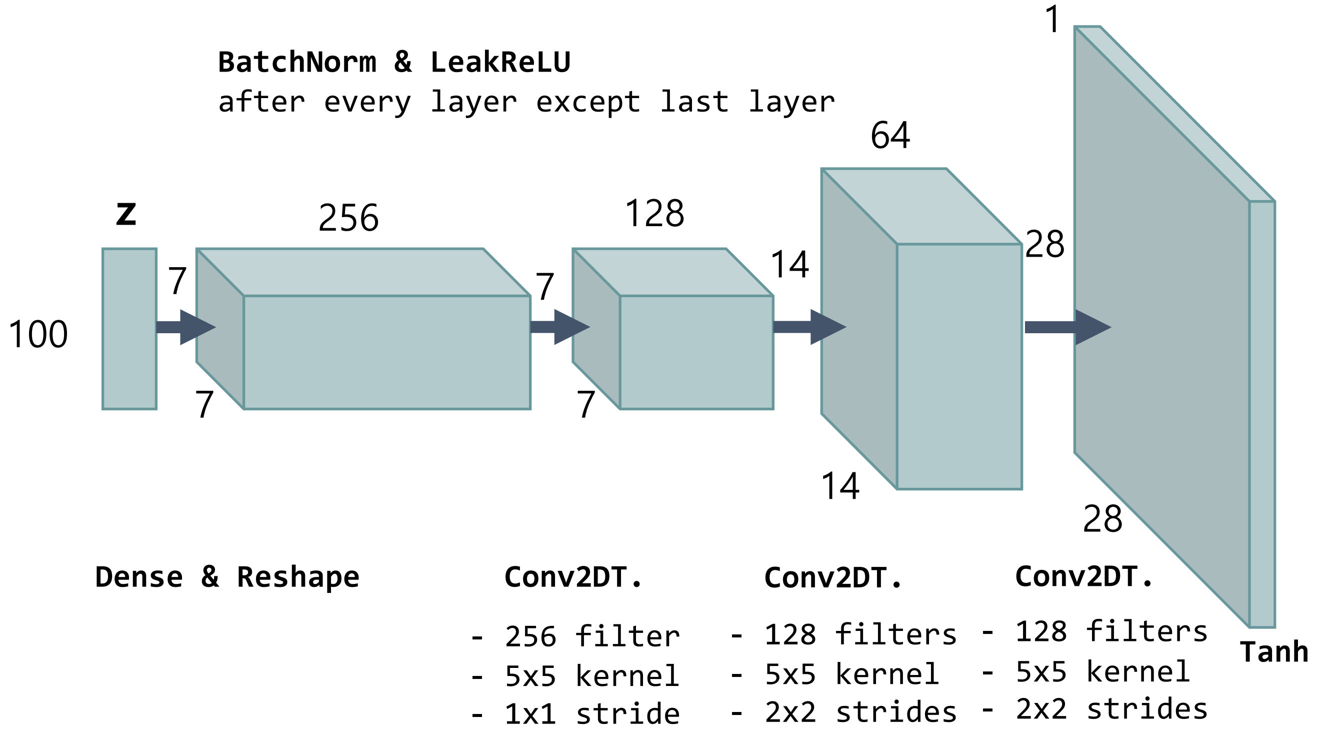
4. Implement **def generator_model():**





- layers.**Dense**(*units*, use_bias=False, *input_shape=(input_dim,)*)
- layers.**Reshape**(*target_shape*)
- layers.**Conv2DTranspose**(*filters*, *kernel_size*, *strides=(1, 1)*, padding='same', use_bias=False)
- layers.**BatchNormalization()**
- layers.**LeakyReLU()**

4. Implement **def generator_model():**



BatchNorm

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

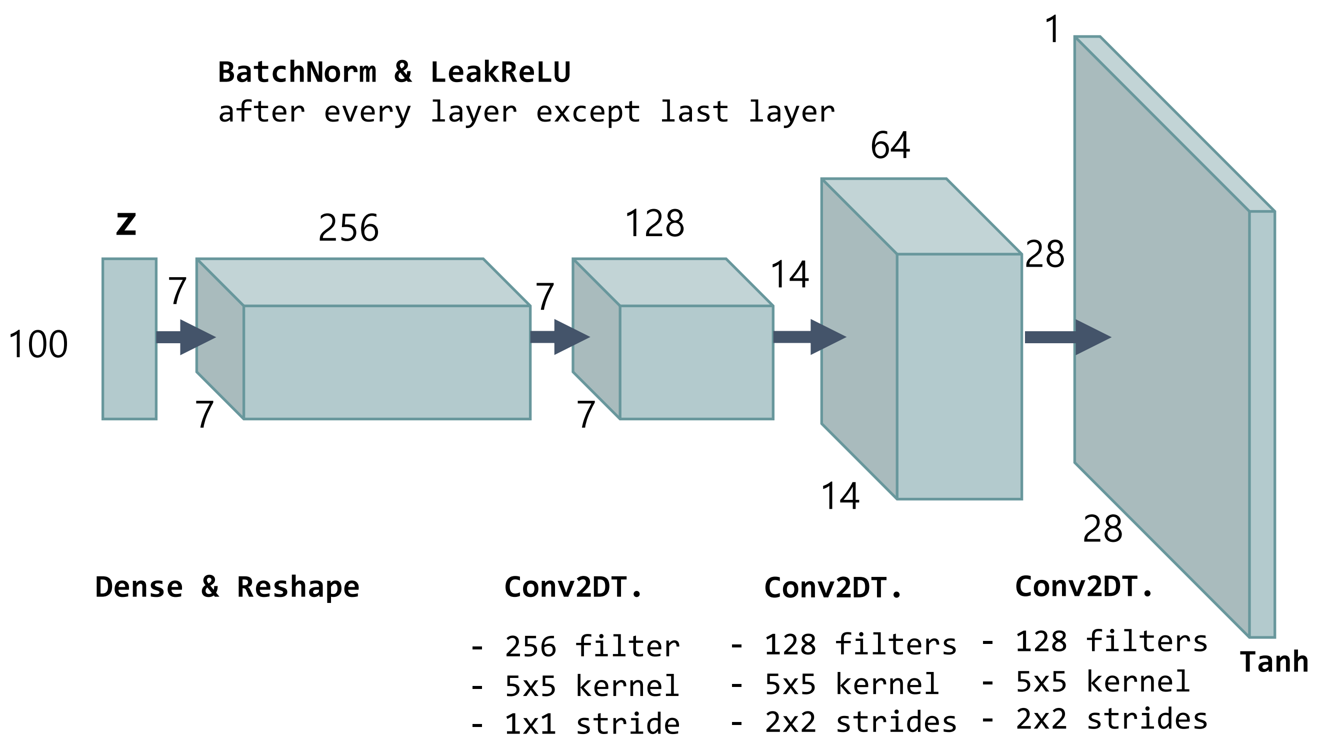
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

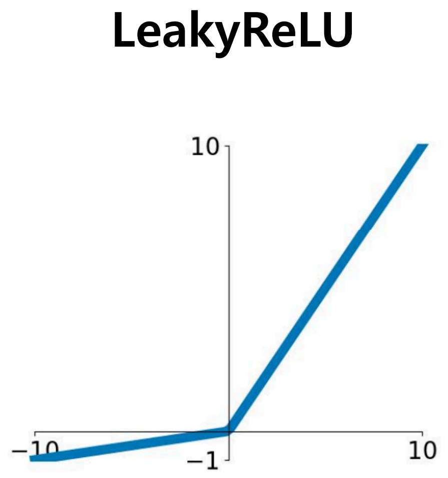
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- `layers.Dense(units, use_bias=False, input_shape=(input_dim,))`
- `layers.Reshape(target_shape)`
- `layers.Conv2DTranspose(filters, kernel_size, strides=(1, 1), padding='same', use_bias=False)`
- `layers.BatchNormalization()`
- `layers.LeakyReLU()`

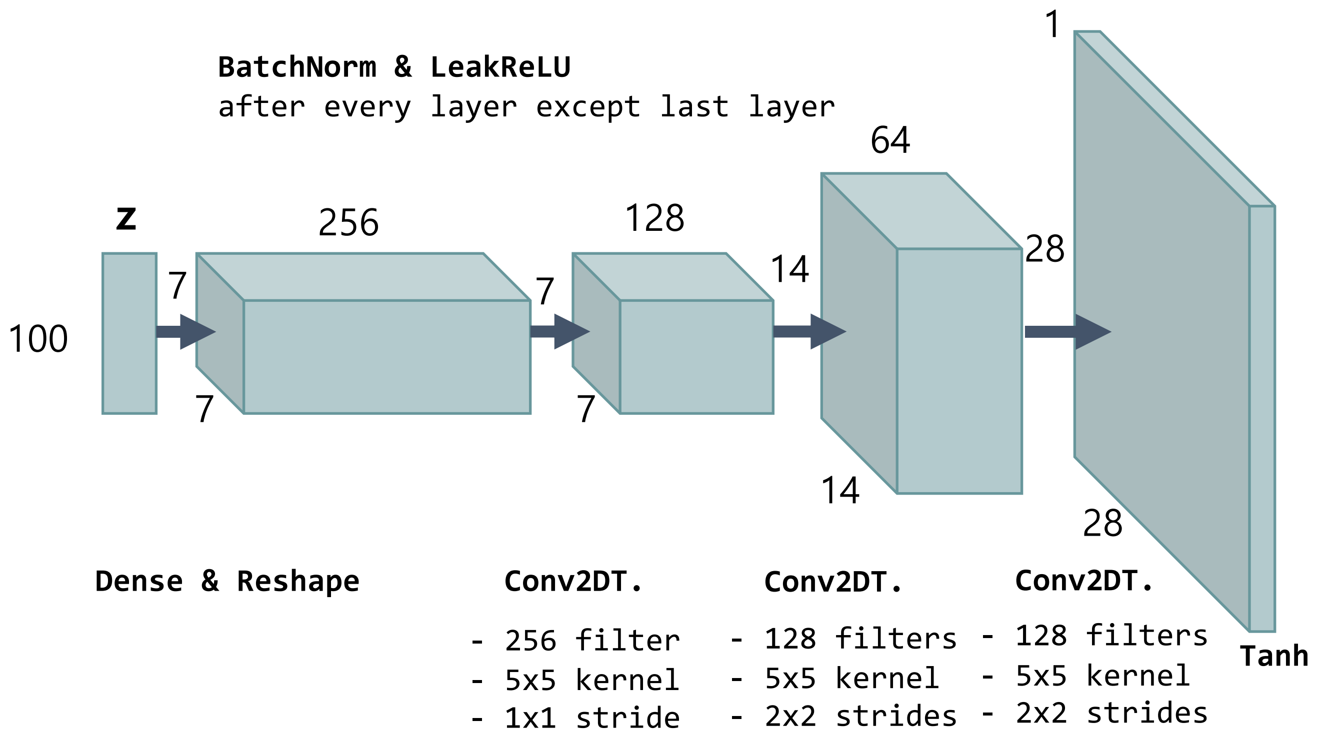
4. Implement **def generator_model():**

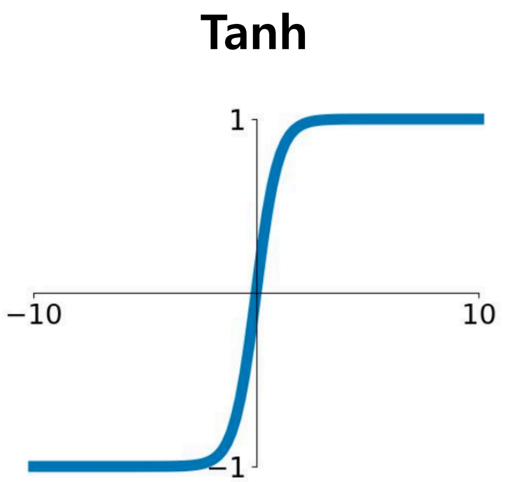




- layers.**Dense**(*units*, use_bias=False, *input_shape=(input_dim,)*)
- layers.**Reshape**(*target_shape*)
- layers.**Conv2DTranspose**(*filters*, *kernel_size*, *strides=(1, 1)*, padding='same', use_bias=False)
- layers.**BatchNormalization()**
- layers.**LeakyReLU()**

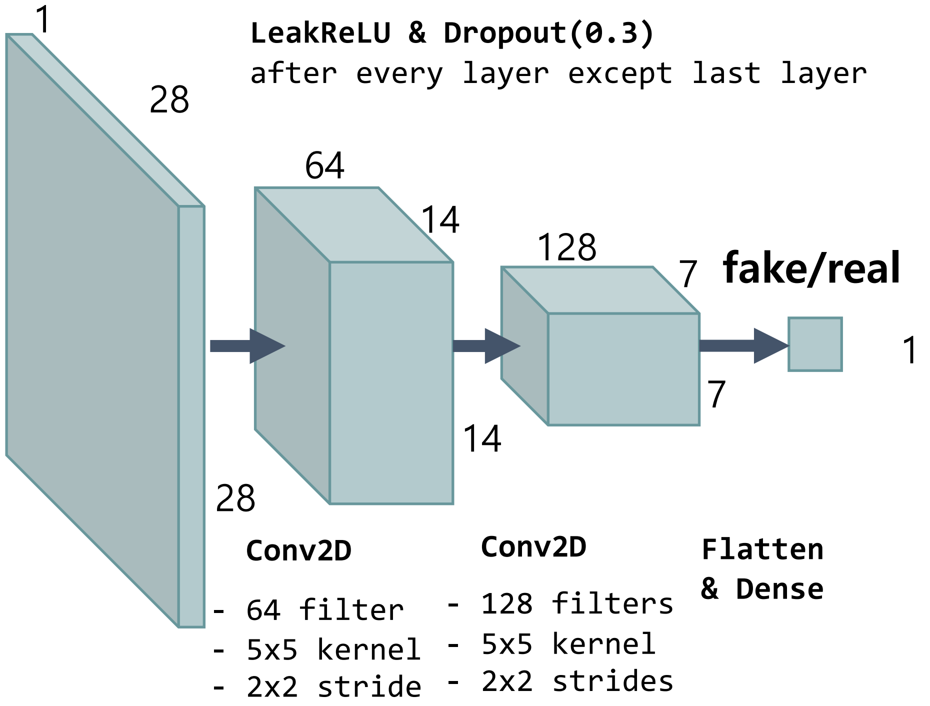
4. Implement **def generator_model():**

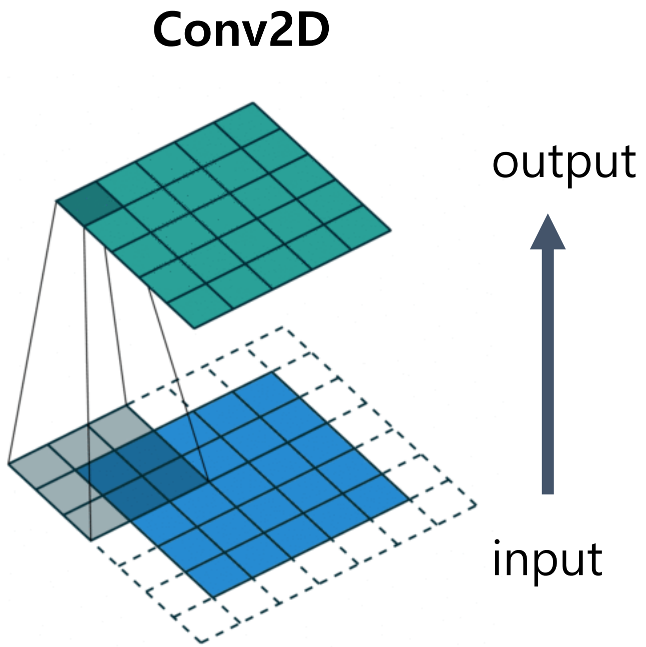




- layers.**Dense**(*units*, use_bias=False, *input_shape=(input_dim,)*)
- layers.**Reshape**(*target_shape*)
- layers.**Conv2DTranspose**(*filters*, *kernel_size*, *strides=(1, 1)*, padding='same', use_bias=False)
- layers.**BatchNormalization()**
- layers.**LeakyReLU()**

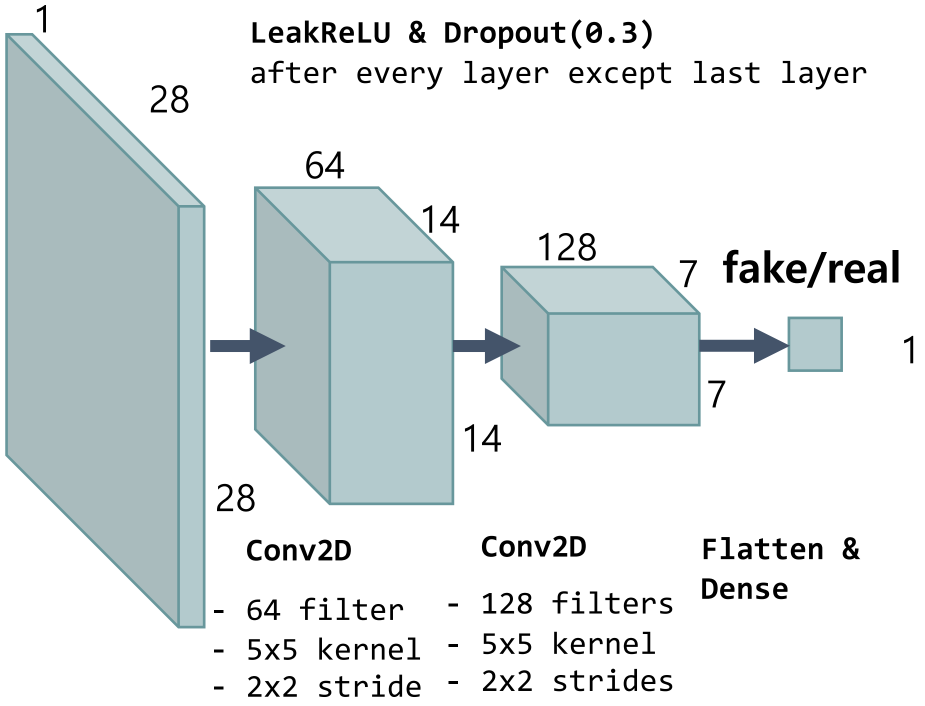
4. Implement **def discriminator_model():**



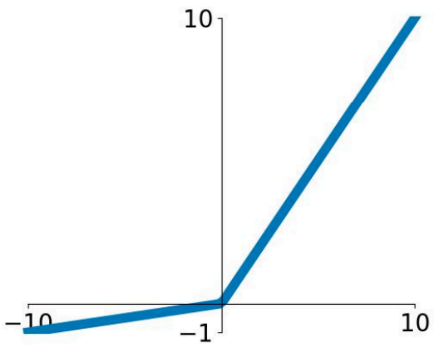


- layers.**Conv2D**(*filters*, *kernel_size*, *strides=(1, 1)*, padding='same', use_bias=False)
- layers.**LeakyReLU()**
- layers.**Dropout**(*rate*)
- layers.**Flatten**()
- layers.**Dense**(*units*, use_bias=False, *input_shape=(input_dim,)*)

4. Implement def **discriminator_model():**

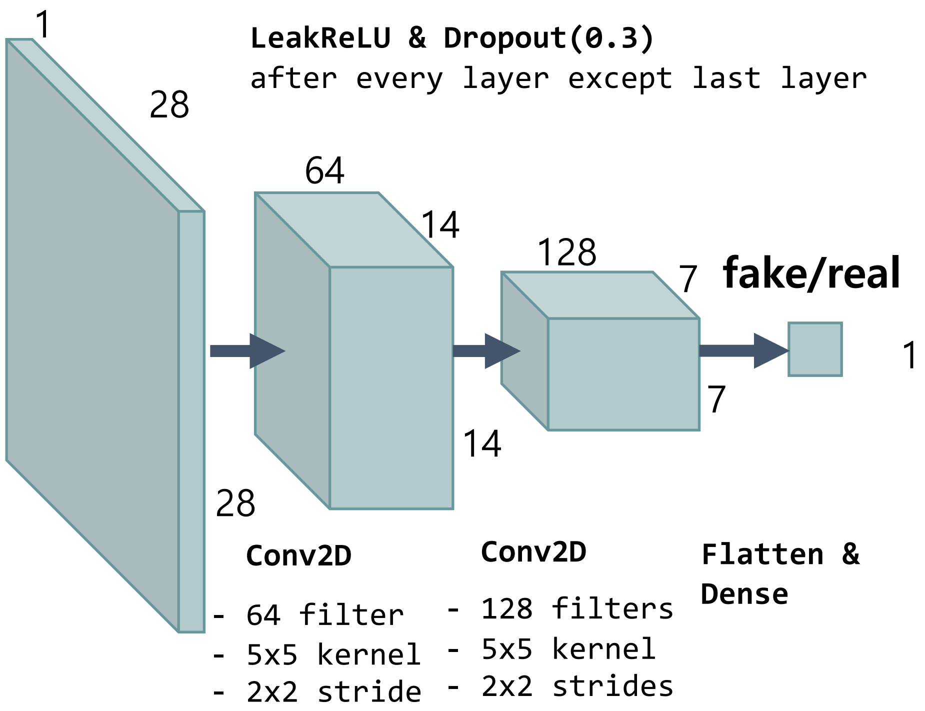


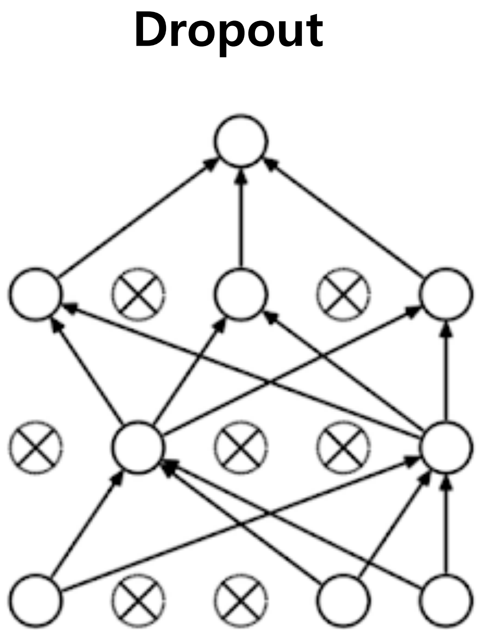
LeakyReLU



- layers.**Conv2D**(*filters*, *kernel_size*, *strides=(1, 1)*, padding='same', use_bias=False)
- layers.**LeakyReLU()**
- layers.**Dropout**(*rate*)
- layers.**Flatten**()
- layers.**Dense**(*units*, use_bias=False, *input_shape=(input_dim,)*)

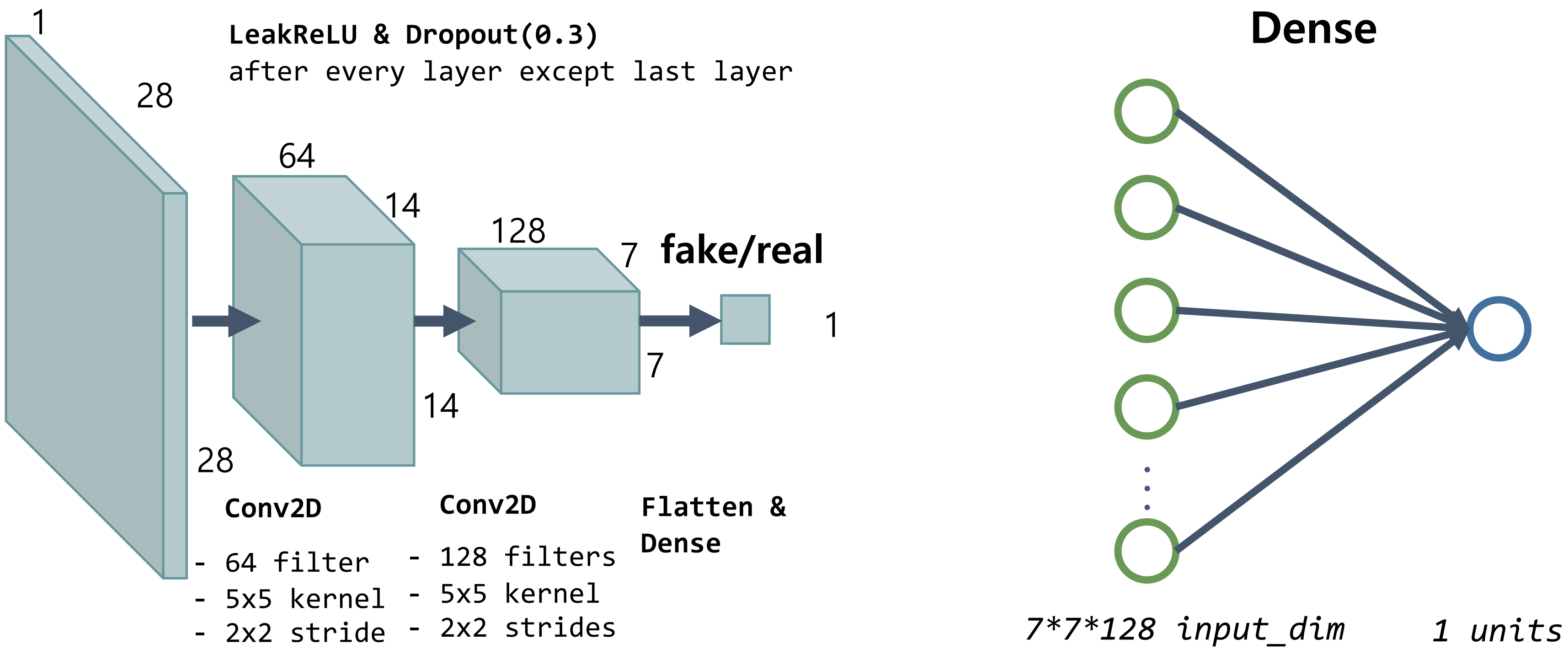
4. Implement **def discriminator_model():**





- layers.**Conv2D**(*filters*, *kernel_size*, *strides=(1, 1)*, padding='same', use_bias=False)
- layers.**LeakyReLU()**
- layers.**Dropout**(*rate*)
- layers.**Flatten**()
- layers.**Dense**(*units*, use_bias=False, *input_shape=(input_dim,)*)

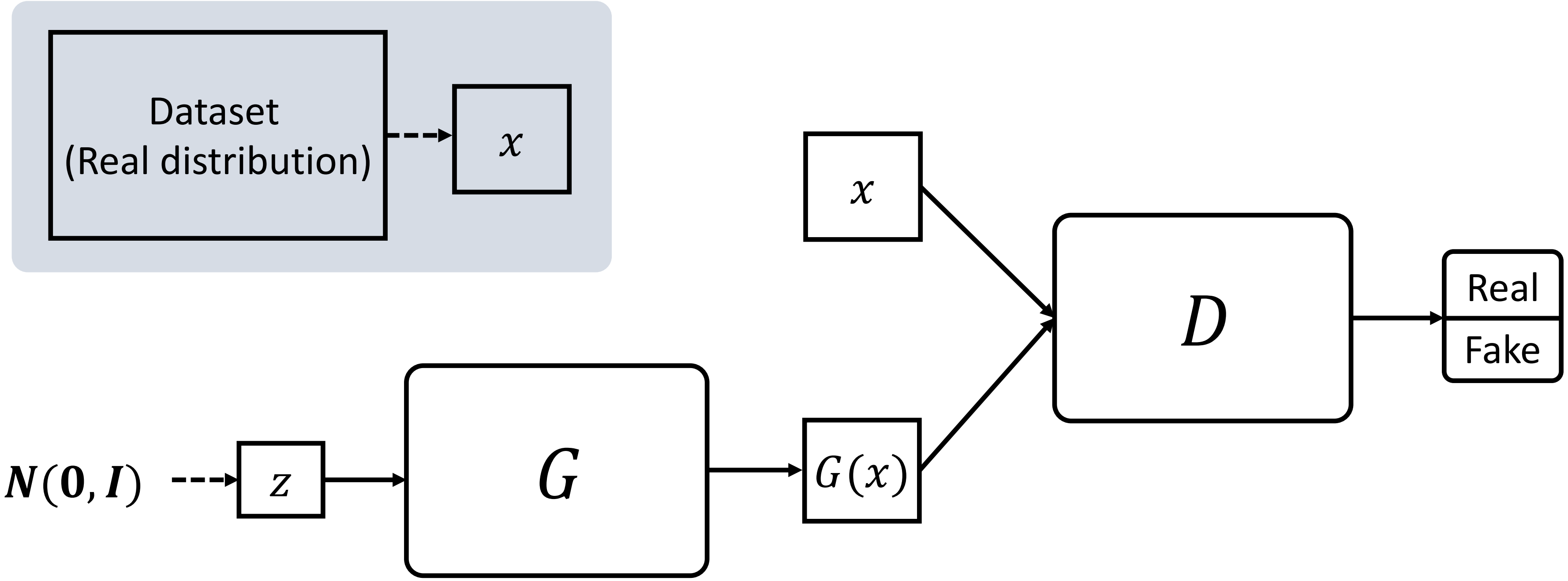
4. Implement **def discriminator_model():**



- layers.**Conv2D**(*filters*, *kernel_size*, *strides=(1, 1)*, padding='same', use_bias=False)
- layers.**LeakyReLU()**
- layers.**Dropout**(*rate*)
- layers.**Flatten**()
- layers.**Dense**(*units*, use_bias=False, *input_shape=(input_dim,)*)

5. Loss Function

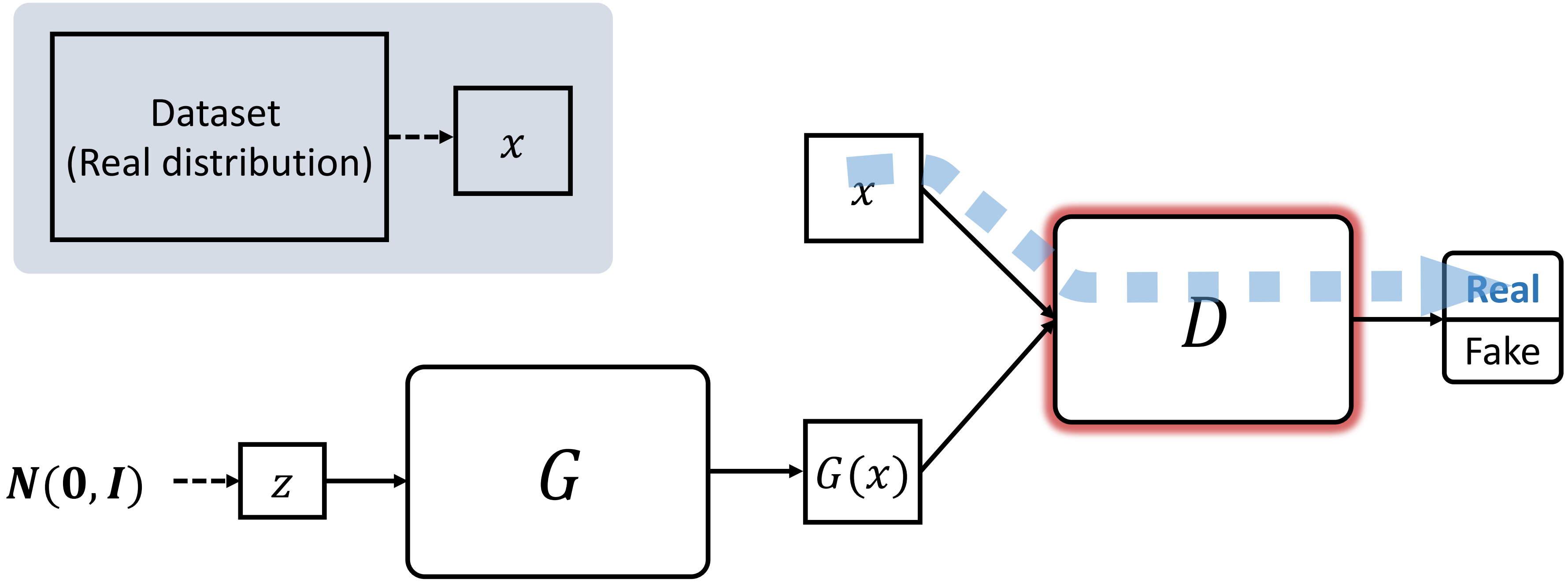
Recap: Overview



$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

5. Loss Function

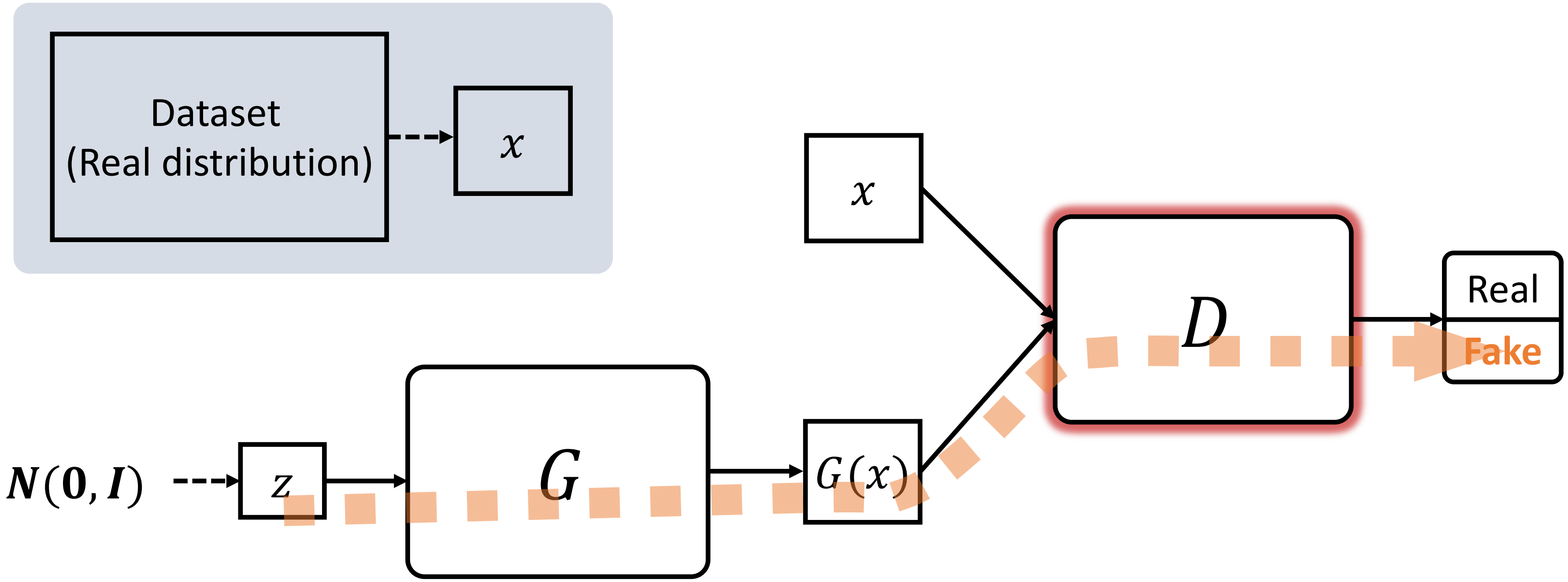
Recap: Training of *Discriminator* *(Real → Real)*



$$\min_{G} \boxed{\max_{D}} V(D, G) = \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

5. Loss Function

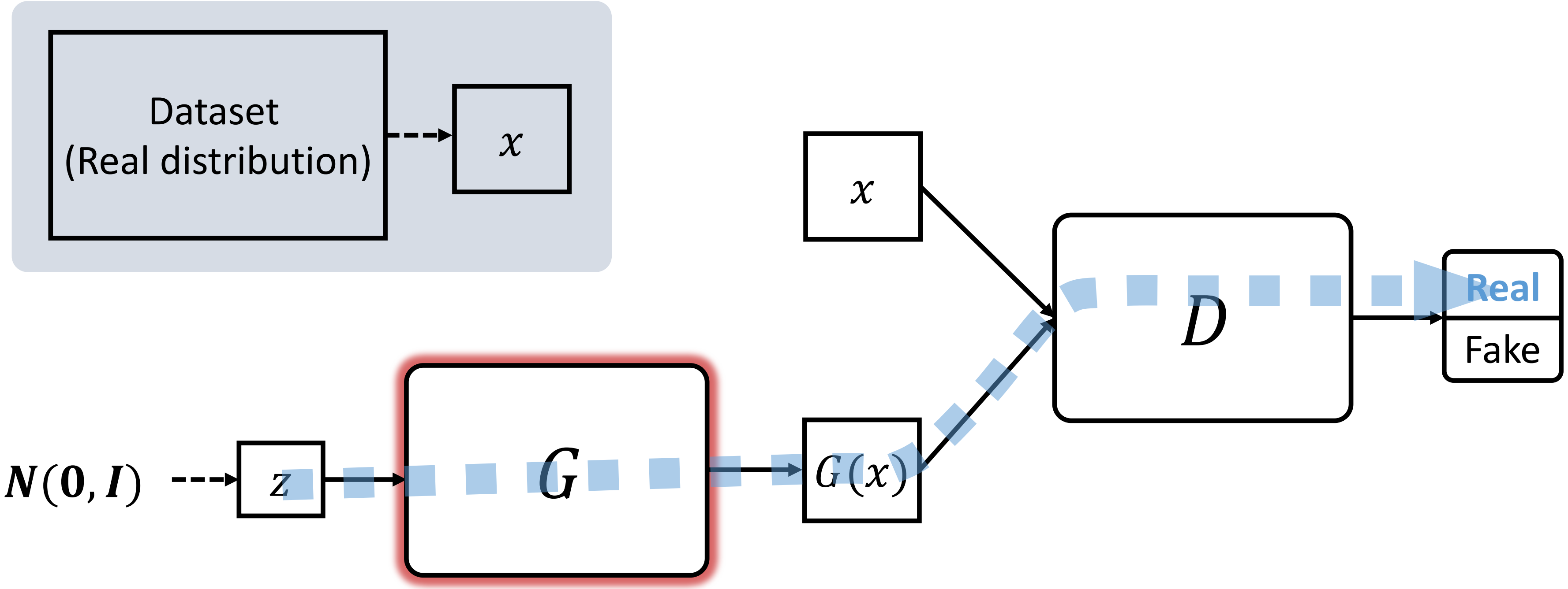
Recap: Training of *Discriminator* *(Fake → Fake)*



$$\min_G \boxed{\max_D} V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

5. Loss Function

Recap: Training of *Generator* *(Fake → Real)*



$$\boxed{\min_G} \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

5. Loss Function

Loss function and Optimization

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

Discriminator

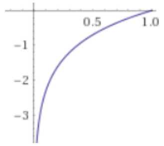
prob. of D predicting that Real data is genuine.

*prob. of D predicting that fake data is **NOT** genuine.*

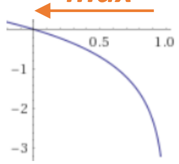
Maximize : $\log(D_1(x)) + \log(1 - D_2(G(z)))$

$\log(x)$

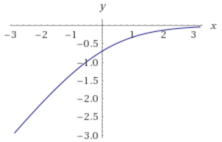
max

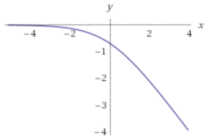


max



$\log(sgm(x))$



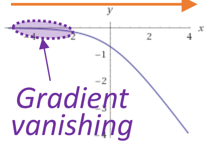


Generator

*prob. of D predicting that fake data is **NOT** genuine.*

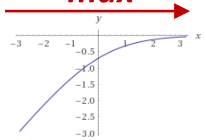
Minimize : $\log(1 - D_2(G(z)))$

min



Maximize : $\log(D_2(G(z)))$

max



prob. of D predicting that fake data is genuine.

: D rejecting G(x) with high confidence.

5. Loss Function

Loss function and Optimization

We will use the Binary Cross Entropy loss function.

```
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)
loss = cross_entropy(y_true, y_pred)
```

We can specify which part of the equation to use with the label $\boldsymbol{y}$.

$$l_n = -\left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n)\right]$$

Real label: y = 1
Fake label: y = 0

We use this part,
when label is ***real***

We use this part,
when label is ***fake***

6. Optimizer & Checkpoint

Since two networks are trained separately, two optimizers are defined.

```
generator_optimizer = tf.keras.optimizers.Adam(LR)
discriminator_optimizer = tf.keras.optimizers.Adam(LR)
```

We can save and restore the model using `tf.train.Checkpoint`

```
checkpoint_prefix = os.path.join(CKPT_DIR, "ckpt")
checkpoint = tf.train.Checkpoint(
                     generator_optimizer=generator_optimizer,
                     discriminator_optimizer=discriminator_optimizer,
                     generator=generator, discriminator=discriminator)

checkpoint.save(file_prefix = checkpoint_prefix) # save model
...
checkpoint.restore(tf.train.latest_checkpoint(CKPT_DIR)) # restore model
```

7. Implement **def train_step(images):**

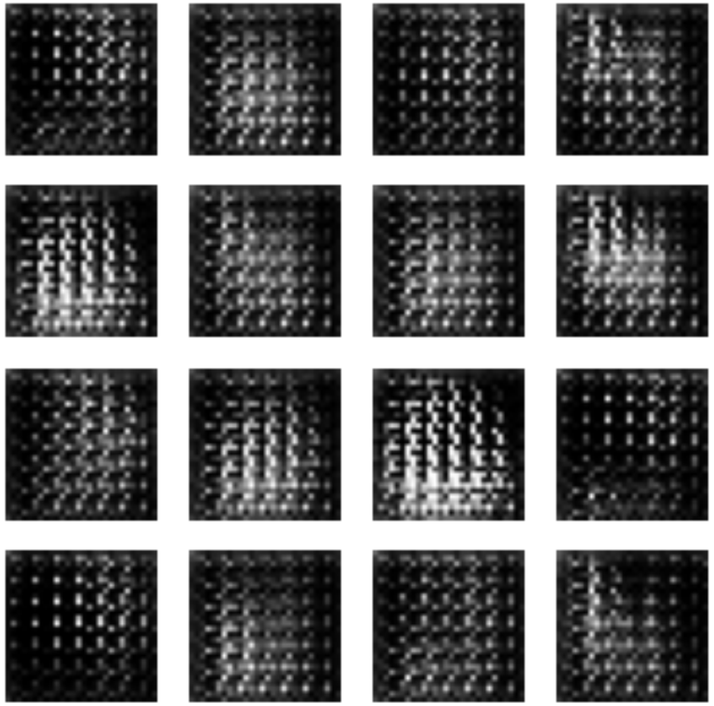
```
@tf.function
def train_step(images):
    # 1. sample noise z from normal distribution
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
      # 2. generator generates fake image
      # 3. discriminator discriminates real image
      # 4. discriminator discriminates fake image
      # 5. compute discriminator loss
      # 6. compute generator loss

    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))
```

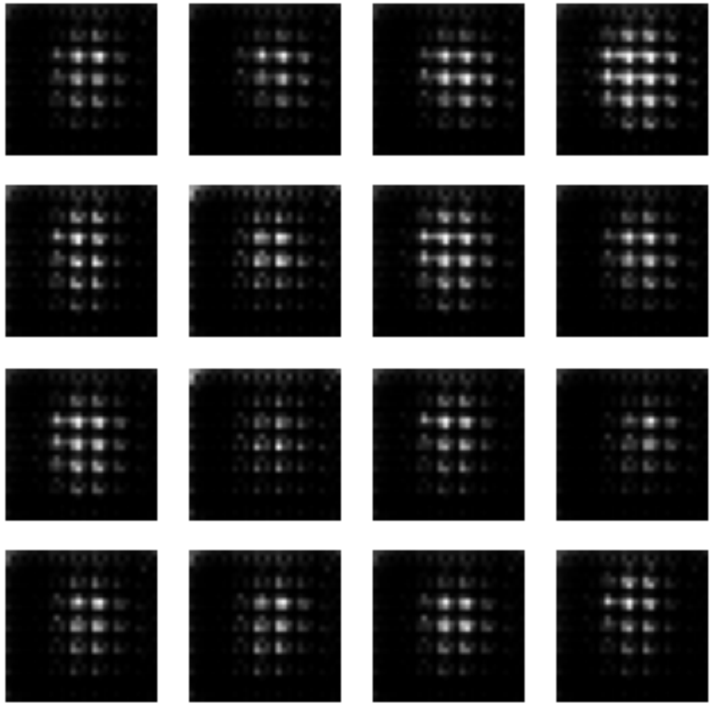
7. Training

```
for epoch in range(EPOCHS):
    start = time.time()

    for image_batch in train_dataset:
        train_step(image_batch)

    with train_summary_writer.as_default():
        tf.summary.scalar('gen_loss', train_gen_loss.result(), step=epoch)
        tf.summary.scalar('disc_loss', train_disc_loss.result(), step=epoch)
    generate_and_save_images(generator, epoch + 1, seed_z, IMAGE_DIR)

    if (epoch + 1) % 15 == 0:
        checkpoint.save(file_prefix = checkpoint_prefix)
        print ('Time for epoch {} is {} sec'.format(epoch + 1, time.time()-start))

generate_and_save_images(generator, EPOCHS, seed_z, IMAGE_DIR)
generate_and_save_gif(IMAGE_DIR, GIF_FILE)
```

Result

After running dcgan.py, you will have gif files.





Any questions?