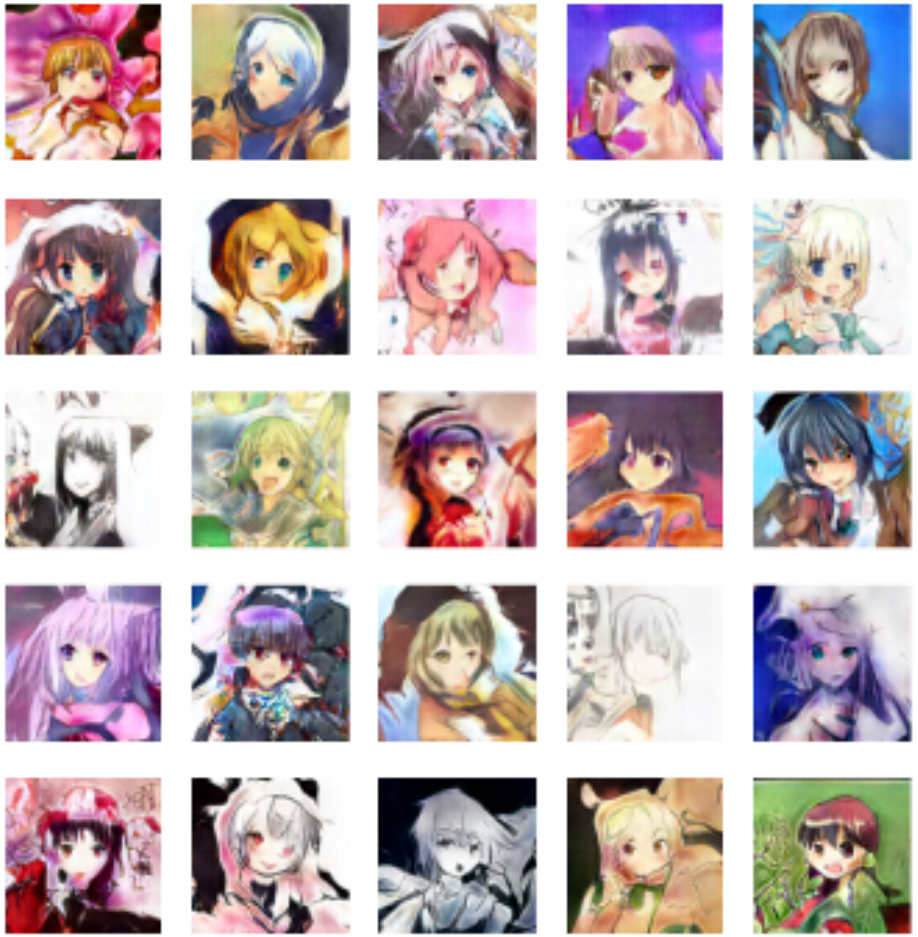
Generative Adversarial Networks (GANs)

Shusen Wang

Image Generation

real images for training

generated images



Reference: https://qiita.com/mattya/items/e5bfe5e04b9d2f0bbd47

GAN: Main Idea

- **Generator**: generates fake images to *fool* the discriminator.
- **Discriminator**: tries to *distinguish* between real and fake images.

- Train them against each other.
- Finally, the discriminator cannot distinguish between real and fake.
- ➔ It means the fake images look like real.

GAN: Main Idea

- **Generator**: a forger who wants to create a fake Picasso painting.
- **Discriminator**: an art dealer.

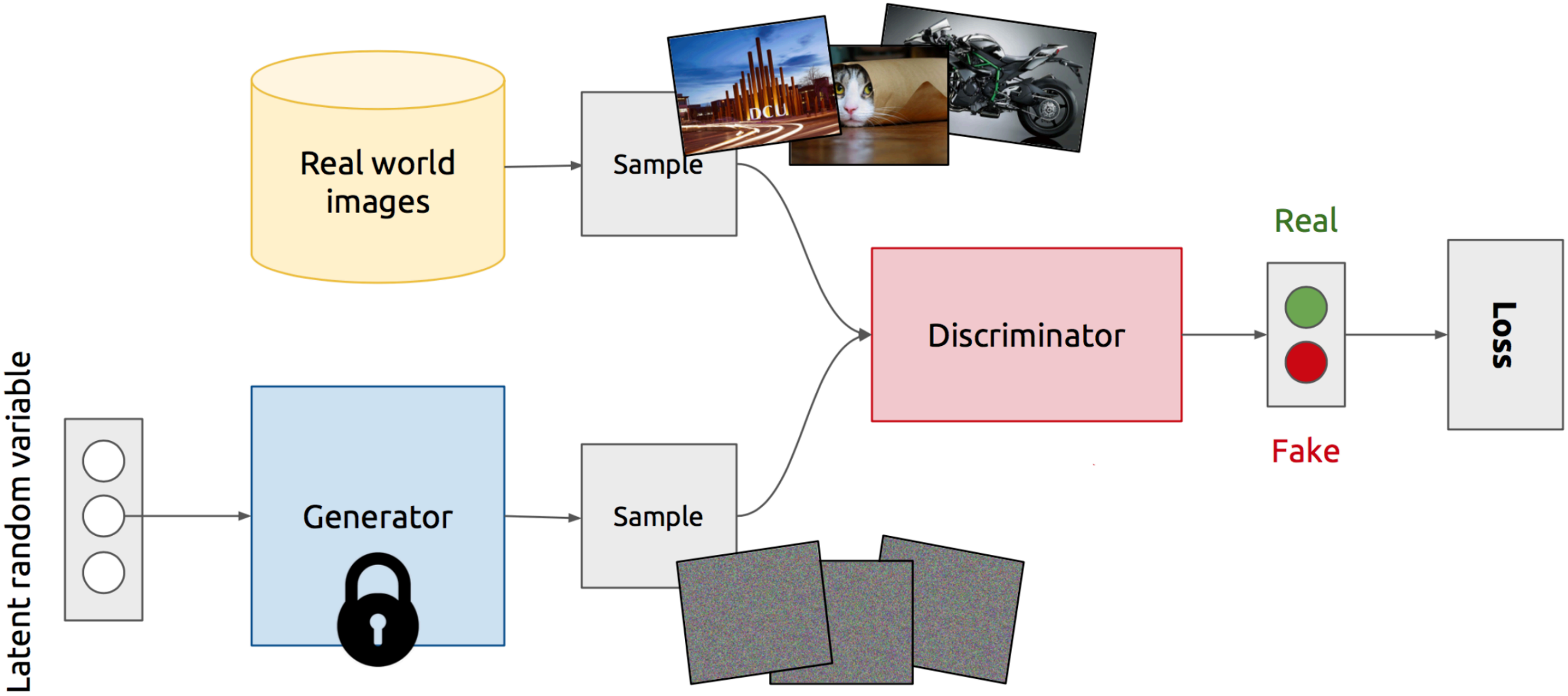
Which is real?

GAN: Main Idea

- **Generator**: a forger who wants to create a fake Picasso painting.
- **Discriminator**: an art dealer.

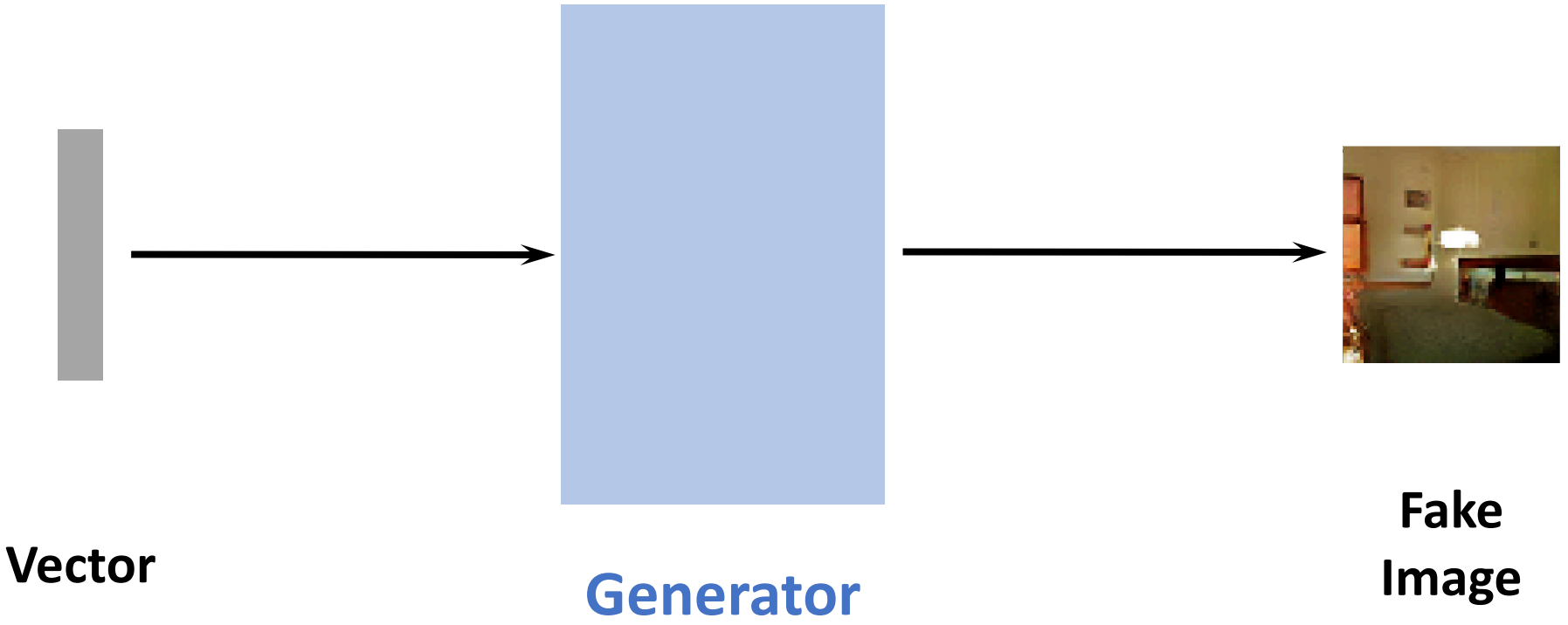
- Train them against each other.
  - Forger mixes his fake with authentic paintings and ask art dealer to assess them.
  - Art dealer gives feedback about what makes a Picasso look like a Picasso.
  - Forger improves his competence at imitating the style of Picasso.
  - Art dealer improves his competence at distinguishing real and fake Picasso.
- Finally, the art dealer cannot distinguish between real and fake.

GAN: Model Overview



Build Generator and Discriminator Networks

Generator: From Vector to Image



Generator: From Vector to Image

```
from keras.layers import Input, Activation, BatchNormalization, Dropout
from keras.layers import Dense, Reshape, UpSampling2D, Conv2DTranspose

depth = 256
dim = 7

input_vec = Input(shape=(100,), name='input_vec')

dense1 = Dense(dim*dim*depth, name='dense1')(input_vec)
bn1 = BatchNormalization(momentum=0.9, name='bn1')(dense1)
act1 = Activation('relu', name='act1')(bn1)
dropout1 = Dropout(rate=0.4, name='dropout1')(act1)
reshape1 = Reshape((dim, dim, depth), name='reshape1')(dropout1)

us2 = UpSampling2D(name='upsampling2')(reshape1)
convt2 = Conv2DTranspose(int(depth/2), 5, padding='same', name='convt2')(us2)
bn2 = BatchNormalization(momentum=0.9, name='bn2')(convt2)
act2 = Activation('relu', name='act2')(bn2)

us3 = UpSampling2D(name='upsampling3')(act2)
convt3 = Conv2DTranspose(int(depth/4), 5, padding='same', name='convt3')(us3)
bn3 = BatchNormalization(momentum=0.9, name='bn3')(convt3)
act3 = Activation('relu', name='act3')(bn3)

convt4 = Conv2DTranspose(int(depth/8), 5, padding='same', name='convt4')(act3)
bn4 = BatchNormalization(momentum=0.9, name='bn4')(convt4)
act4 = Activation('relu', name='act4')(bn4)

convt5 = Conv2DTranspose(1, 5, padding='same', activation='sigmoid', name='convt5')(act4)
```

Block 1:
from 100-dim to 7×7×256

Block 2:
from 7×7×256 to 14×14×128

Block 3:
From 14×14×128 to 28×28×64

Block 4:
From 28×28×64 to 28×28×32

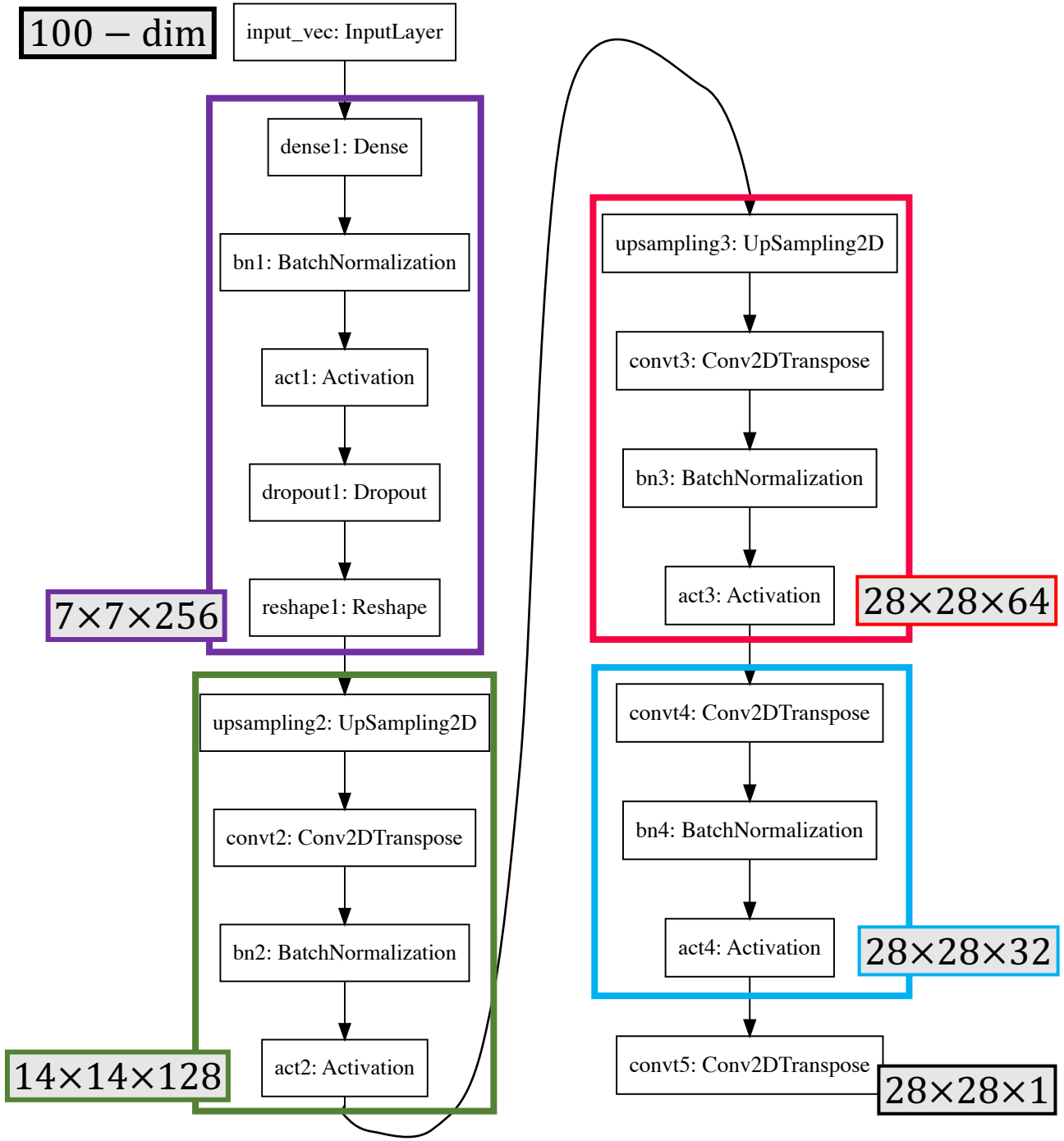
Generator: From Vector to Image

```
from keras.models import Model

generator = Model(inputs=input_vec, outputs=convt5, name='generator')
```

Layer (type)	Output Shape	Param #
input_vec (InputLayer)	(None, 100)	0
dense1 (Dense)	(None, 12544)	1266944
bn1 (BatchNormalization)	(None, 12544)	50176
act1 (Activation)	(None, 12544)	0
dropout1 (Dropout)	(None, 12544)	0
reshape1 (Reshape)	(None, 7, 7, 256)	0
upsampling2 (UpSampling2D)	(None, 14, 14, 256)	0
convt2 (Conv2DTranspose)	(None, 14, 14, 128)	819328
bn2 (BatchNormalization)	(None, 14, 14, 128)	512
act2 (Activation)	(None, 14, 14, 128)	0
upsampling3 (UpSampling2D)	(None, 28, 28, 128)	0
convt3 (Conv2DTranspose)	(None, 28, 28, 64)	204864
bn3 (BatchNormalization)	(None, 28, 28, 64)	256
act3 (Activation)	(None, 28, 28, 64)	0
convt4 (Conv2DTranspose)	(None, 28, 28, 32)	51232
bn4 (BatchNormalization)	(None, 28, 28, 32)	128
act4 (Activation)	(None, 28, 28, 32)	0
convt5 (Conv2DTranspose)	(None, 28, 28, 1)	801

Total params: 2,394,241
Trainable params: 2,368,705
Non-trainable params: 25,536



Discriminator: From Image to Binary

```
from keras.layers import Input, LeakyReLU, Dropout
from keras.layers import Dense, Flatten, Conv2D

depth = 64

input_img = Input(shape=(28, 28, 1), name='input_img')

conv1 = Conv2D(depth, 5, strides=2, padding='same', name='conv1')(input_img)
act1 = LeakyReLU(alpha=0.2, name='act1')(conv1)
dropout1 = Dropout(0.4, name='dropout1')(act1)

conv2 = Conv2D(depth*2, 5, strides=2, padding='same', name='conv2')(dropout1)
act2 = LeakyReLU(alpha=0.2, name='act2')(conv2)
dropout2 = Dropout(0.4, name='dropout2')(act2)

conv3 = Conv2D(depth*4, 5, strides=2, padding='same', name='conv3')(dropout2)
act3 = LeakyReLU(alpha=0.2, name='act3')(conv3)
dropout3 = Dropout(0.4, name='dropout3')(act3)

conv4 = Conv2D(depth*8, 5, strides=1, padding='same', name='conv4')(dropout3)
act4 = LeakyReLU(alpha=0.2, name='act4')(conv4)
dropout4 = Dropout(0.4, name='dropout4')(act4)

flat5 = Flatten(name='flat5')(dropout4)
dense5 = Dense(1, activation='sigmoid', name='dense5')(flat5)
```

Block 1:
from $28 \times 28 \times 1$ to $14 \times 14 \times 64$

Block 2:
from $14 \times 14 \times 64$ to $7 \times 7 \times 128$

Block 3:
From $7 \times 7 \times 128$ to $4 \times 4 \times 256$

Block 4:
From $4 \times 4 \times 256$ to $4 \times 4 \times 512$

Discriminator: From Image to Binary

```
from keras.models import Model

discriminator = Model(inputs=input_img, outputs=dense5, name='discriminator')
```

Training

Training of GAN

Alternating minimization

Repeat the 2 steps:

1. Update the discriminator network;
2. Update the generator network.

Update the Discriminator

Train a classifier

1. Generate a batch of fake images by the generator;
2. Randomly sample a batch of real images;
3. Inputs: $\mathbf{X} = [\text{real\_images}, \text{fake\_images}]$;
4. Targets: $\mathbf{y} = [\text{True}, \cdots, \text{True}, \text{False}, \cdots, \text{False}]$;
5. Update the discriminator network using $\mathbf{X}$ and $\mathbf{y}$.

Update the Discriminator

```
from keras.optimizers import RMSprop

optimizer = RMSprop(lr=0.0002, decay=6e-8)
discriminator.compile(loss='binary_crossentropy',
                      optimizer=optimizer,
                      metrics=['accuracy'])
```

Update the Discriminator

Uniformly sample a batch of real images

```
# train the discriminator
discriminator.trainable = True
rand_idx = np.random.randint(0, n, size=batch_size)
images_real = x_train[rand_idx]
latent_vecs = np.random.uniform(-1.0, 1.0, size=[batch_size, 100])
images_fake = generator.predict(latent_vecs)
x = np.concatenate((images_real, images_fake), axis=0)
y = np.concatenate([np.ones((batch_size, 1)),
                    np.zeros((batch_size, 1))])
d_loss = discriminator.train_on_batch(x, y)
```

Update the Discriminator

```
# train the discriminator
discriminator.trainable = True
rand_idx = np.random.randint(0, n, size=batch_size)
images_real = x_train[rand_idx]
latent_vecs = np.random.uniform(-1.0, 1.0, size=[batch_size, 100])
images_fake = generator.predict(latent_vecs)
x = np.concatenate((images_real, images_fake), axis=0)
y = np.concatenate([np.ones((batch_size, 1)),
                    np.zeros((batch_size, 1))])
d_loss = discriminator.train_on_batch(x, y)
```

Generate fake images

Update the Discriminator

```
# train the discriminator
discriminator.trainable = True
rand_idx = np.random.randint(0, n, size=batch_size)
images_real = x_train[rand_idx]
latent_vecs = np.random.uniform(-1.0, 1.0, size=[batch_size, 100])
images_fake = generator.predict(latent_vecs)
x = np.concatenate((images_real, images_fake), axis=0)
y = np.concatenate([np.ones((batch_size, 1)),
                    np.zeros((batch_size, 1))])
d_loss = discriminator.train_on_batch(x, y)
```

Set input and targets

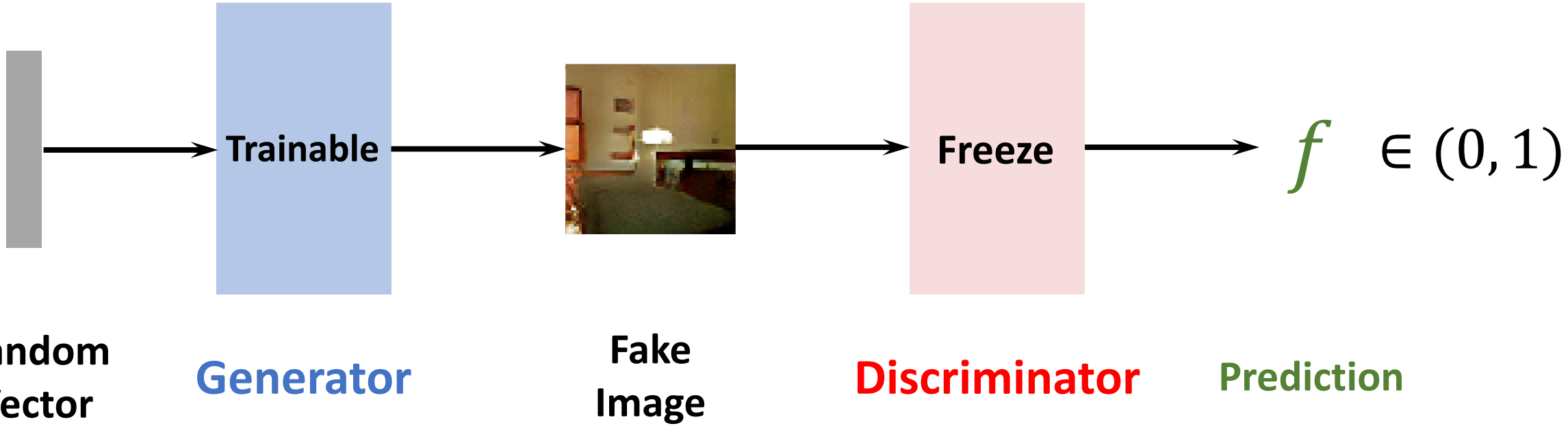
Update the Discriminator

```
# train the discriminator
discriminator.trainable = True
rand_idx = np.random.randint(0, n, size=batch_size)
images_real = x_train[rand_idx]
latent_vecs = np.random.uniform(-1.0, 1.0, size=[batch_size, 100])
images_fake = generator.predict(latent_vecs)
x = np.concatenate((images_real, images_fake), axis=0)
y = np.concatenate([np.ones((batch_size, 1)),
                    np.zeros((batch_size, 1))])
d_loss = discriminator.train_on_batch(x, y)
```

One update of the discriminator's parameters

Update the Generator

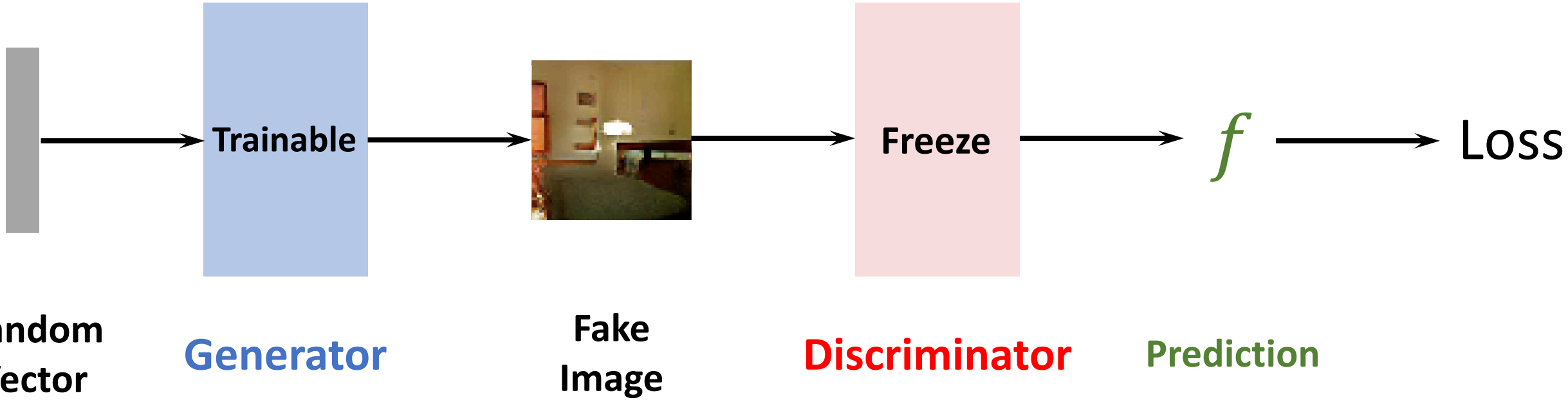
Connect the generator and discriminator (freeze discriminator's parameters).



Update the Generator

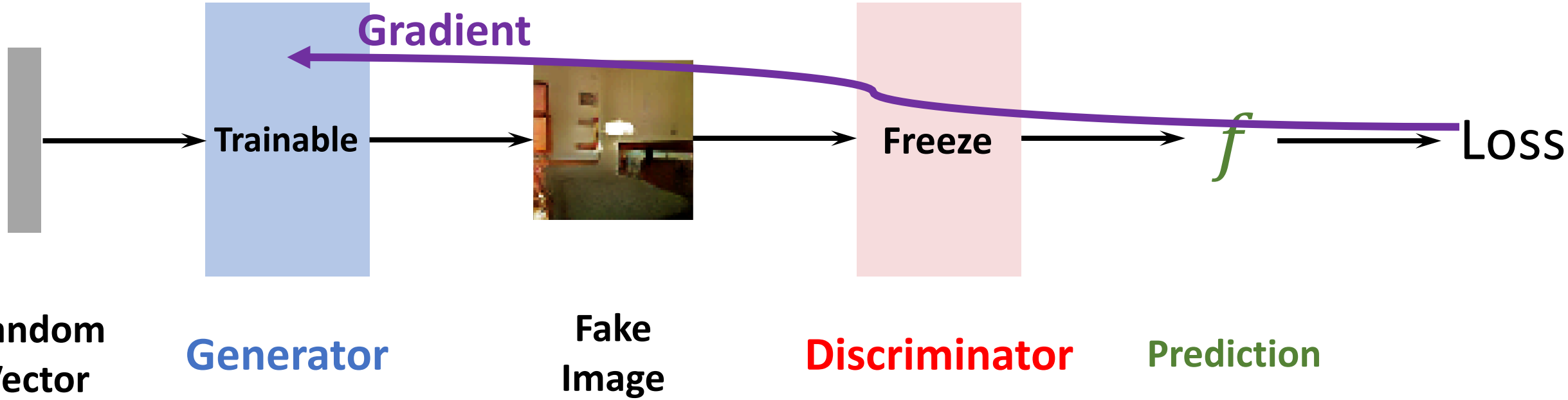
Connect the generator and discriminator (freeze discriminator's parameters).

Minimize $\text{Loss} = \text{Dist}(\text{True}, f)$ w.r.t. generator. (Encourage f be **True.**)



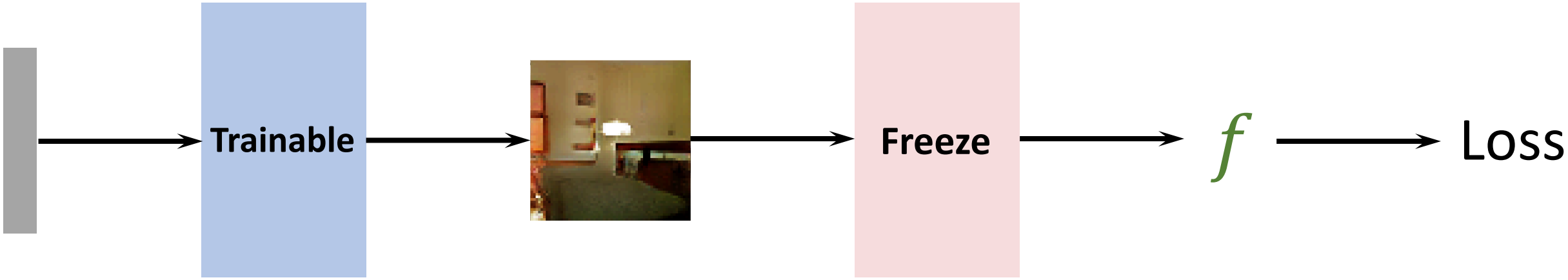
Update the Generator

- $\mathbf{W_G}$: parameters in the generator.
- Gradient: $\text{Grad} = \frac{\partial\ \text{Loss}}{\partial\ \mathbf{W_G}}$.



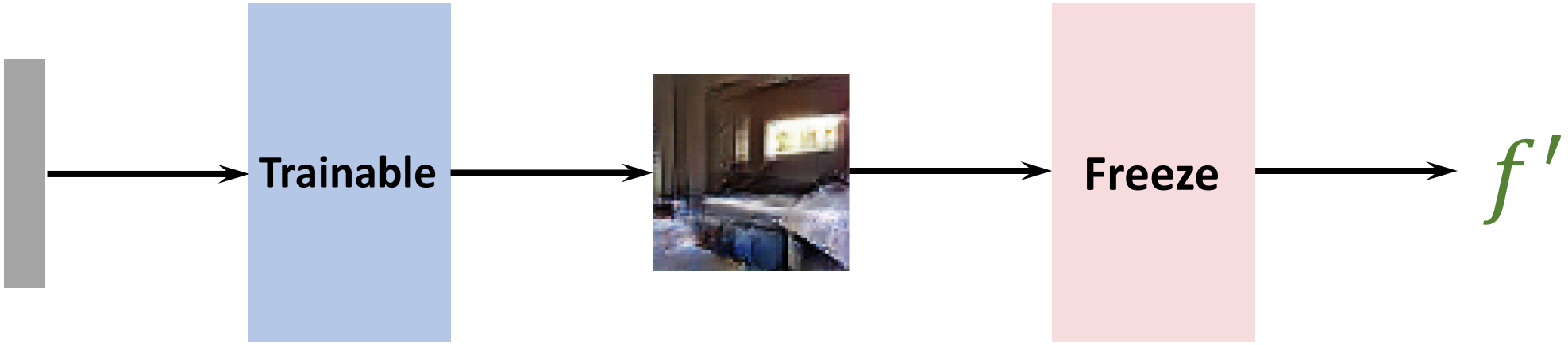
Update the Generator

- $\mathbf{W}_{\mathrm{G}}$: parameters in the generator.
- Gradient: $\mathrm{Grad} = \frac{\partial\ \mathrm{Loss}}{\partial\ \mathbf{W}_{\mathrm{G}}}$.
- Gradient descent: $\mathbf{W}_{\mathrm{G}} \leftarrow \mathbf{W}_{\mathrm{G}} - \alpha \cdot \mathrm{Grad}$.



Update the Generator

- Use the same **vector**.
- Generate fake image using the updated generator.



Random Vector

Generator (updated)

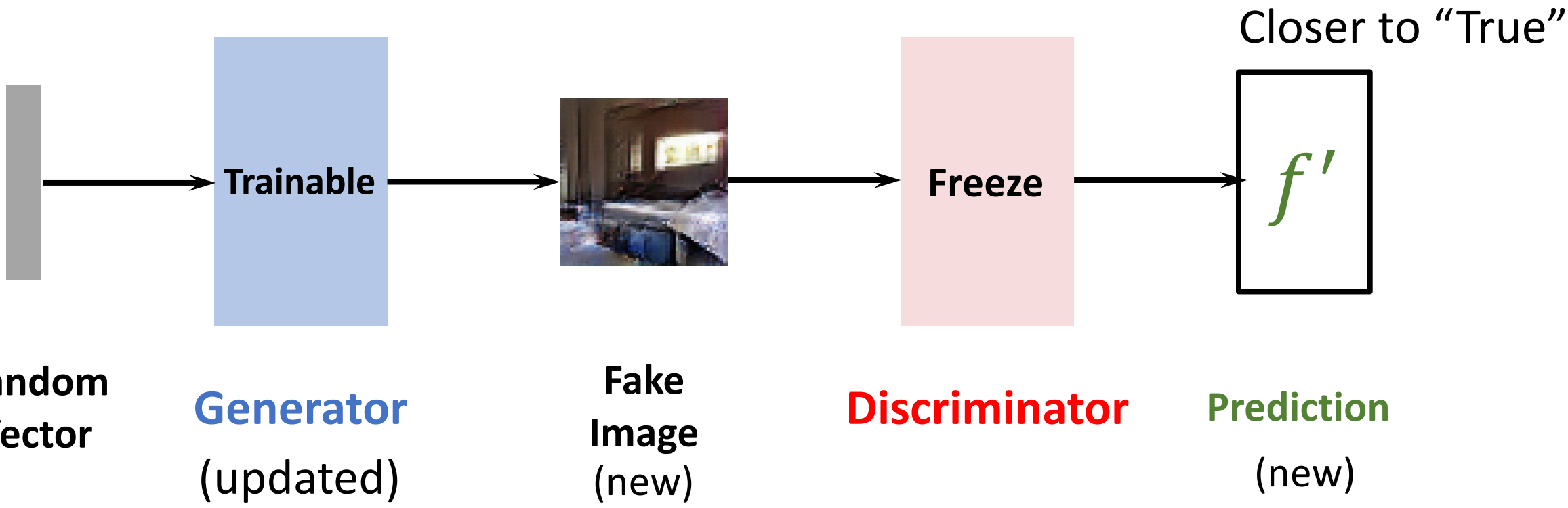
Fake Image (new)

Discriminator

Prediction (new)

Update the Generator

- Use the same **vector**.
- Generate fake image using the updated generator.
- The discriminator thinks the new fake image more “real” than before.



Update the Generator

Connect the generator and discriminator (freeze discriminator's parameters).

```
from keras.layers import Input
from keras.models import Model

discriminator.trainable = False
gan_input = Input(shape=(100,), name='gan_input')
gan_generator = generator(gan_input)
gan_output = discriminator(gan_generator)
gan = Model(inputs=gan_input, outputs=gan_output, name='gan')
```

Update the Generator

Connect the generator and discriminator (freeze discriminator's parameters).

Minimize $\text{Loss} = \text{Dist}(\text{True}, f)$ w.r.t. generator. (Encourage f be **True.**)

```
from keras.optimizers import RMSprop

gan_optimizer = RMSprop(lr=0.0001, decay=3e-8)
gan.compile(optimizer=gan_optimizer,
            loss='binary_crossentropy',
            metrics=['accuracy'])
```

Update the Generator

Connect the generator and discriminator (freeze discriminator's parameters).

Minimize $\text{Loss} = \text{Dist}(\text{True}, f)$ w.r.t. generator. (Encourage f be **True.**)

```
# train the generator
discriminator.trainable = False
fake_targets = np.ones([batch_size, 1]) # pretend the images are real
latent_vecs = np.random.uniform(-1.0, 1.0, size=[batch_size, 100])
a_loss = gan.train_on_batch(latent_vecs, fake_targets)
```

$\text{Loss} = \text{Dist}(\text{True}, f)$

Random vectors

All true

Summary of Training

```
for t in range(train_steps):
    # train the discriminator
    discriminator.trainable = True
    rand_idx = np.random.randint(0, n, size=batch_size)
    images_real = x_train[rand_idx]
    latent_vecs = np.random.uniform(-1.0, 1.0, size=[batch_size, 100])
    images_fake = generator.predict(latent_vecs)
    x = np.concatenate((images_real, images_fake), axis=0)
    y = np.concatenate([np.ones((batch_size, 1)),
                        np.zeros((batch_size, 1))])
    d_loss = discriminator.train_on_batch(x, y)

    # train the generator
    discriminator.trainable = False
    fake_targets = np.ones([batch_size, 1]) # pretend the images are real
    latent_vecs = np.random.uniform(-1.0, 1.0, size=[batch_size, 100])
    a_loss = gan.train_on_batch(latent_vecs, fake_targets)
```

Difficulties in Training GAN

Discriminator Shouldn't Be Too Good

- **Generator**: a forger who wants to create a fake Picasso painting.
- **Discriminator**: an art dealer providing feedbacks.

What if the art dealer is 100% correct at judging Picasso painting?

Discriminator Shouldn't Be Too Good

- **Generator**: a forger who wants to create a fake Picasso painting.
- **Discriminator**: an art dealer providing feedbacks.

What if the art dealer is 100% correct at judging Picasso painting?

- Whatever forged painting sent to the art dealer is recognized as fake.
- The forger cannot learn anything from the *feedback*.
  - No positive case to follow.

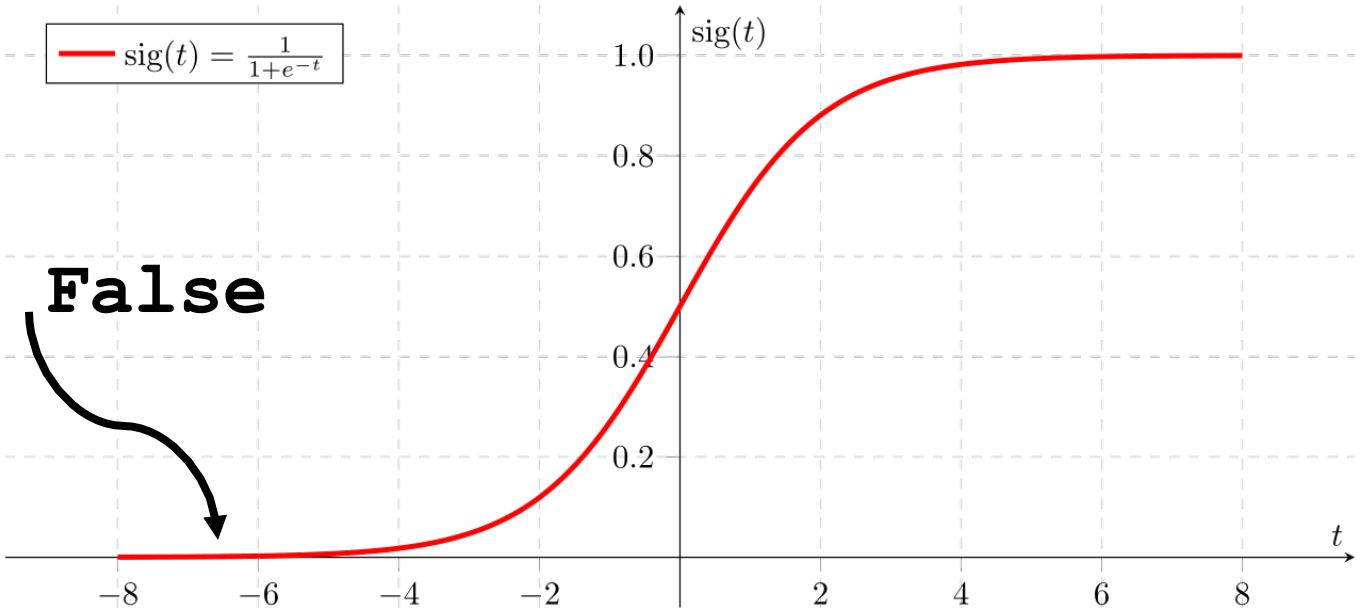
- The forger need some success.
  - So he will know what kind of fake painting can fool the dealer.

Discriminator Shouldn't Be Too Good

Explanation: vanishing gradient

- Suppose the discriminator is perfect.
- Whatever the generators forged is recognized fake by the discriminator.
- ➔ The gradient (averaged over a batch of samples) is near zero.

Prediction = False



Discriminator Shouldn't Be Too Bad

- **Generator**: a forger who wants to create a fake Picasso painting.
- **Discriminator**: an art dealer providing feedbacks.

What if the art dealer is cannot distinguish between real and fake paintings?

- The art dealer's judgement is almost *random guess*.
- The forger cannot learn anything from the *feedback*.
- ➔ When the forger's skill is good, getting amateurish art dealer's feedback is a not helpful.

Useful Tricks

1. Carefully tune the learning rates.

- If the discriminator improves too fast,
  - ➔ classification accuracy can be near 100%,
  - ➔ vanishing gradient,
  - ➔ the generator is dead.

- If the generator improves too fast,
  - ➔ the discriminator cannot provide useful feedback,
  - ➔ the generator has to wait for the discriminator,
  - ➔ slow convergence.

Useful Tricks

2. Add noise to the real and fake images; decay the noise over time.

- When training the discriminator, perturb the inputs (both real and fake images).

Useful Tricks

2. Add noise to the real and fake images; decay the noise over time.

- When training the discriminator, perturb the inputs (both real and fake images).

3. Add noise to the labels.

- When training the discriminator, perturb the labels (both real and fake images).
- `Real = 1` → `Real ~ Uniform(0.7, 1.2).`
- `Fake = 0` → `Fake ~ Uniform(0.0, 0.3).`

Useful Tricks

Many other tricks…

- Further reading:
- https://github.com/soumith/ganhacks