Lecture 12

Tian Han

Outline

- **Variational autoencoder**

- **Generative Adversarial Net**

Variational Autoencoder for Image Generation

Preliminary: Distance between Two Probability Distributions

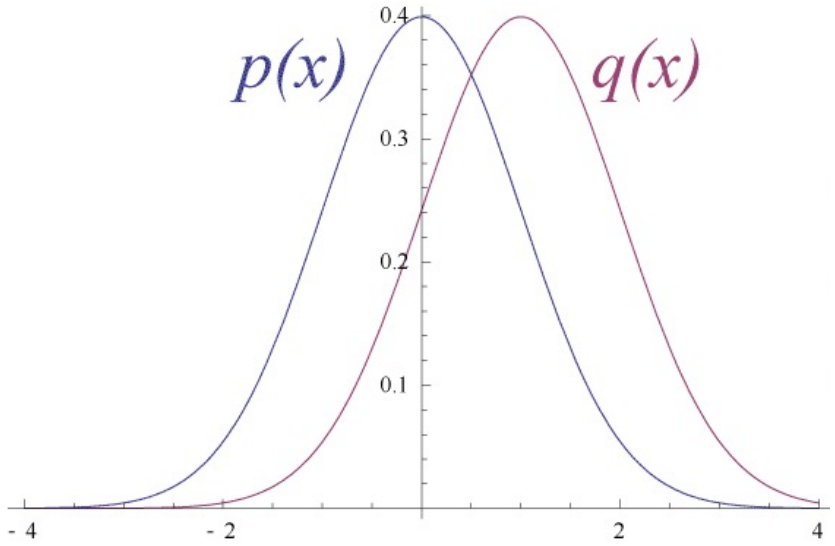
Distance between Two Vectors

- Distance between two vectors is measured by vector norms.
- Examples:
  - ℓ_1 distance: $\left|\left|\mathbf{a}-\mathbf{b}\right|\right|_1 = \sum_j \left|a_j - b_j\right|$.
  - ℓ_2 distance: $\left|\left|\mathbf{a}-\mathbf{b}\right|\right|_2 = \sqrt{\sum_j \left(a_j - b_j\right)^2}$.
  - ℓ_∞ distance: $\left|\left|\mathbf{a}-\mathbf{b}\right|\right|_\infty = \max_j \left|a_j - b_j\right|$.
  - ℓ_p distance: $\left|\left|\mathbf{a}-\mathbf{b}\right|\right|_p = \left[\sum_j \left(a_j - b_j\right)^p\right]^{1/p}$.

Distance between Two Probability Distributions

Question: What is the distance between $p(x)$ and $q(x)$?

- Let $p(x)$ and $q(x)$ be two Probability Density Functions (PDFs).
- E.g., the PDF of a Gaussian distribution is $p(x|\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$.



Distance between Two Probability Distributions

Question: What is the distance between $p(x)$ and $q(x)$?

- Let $p(x)$ and $q(x)$ be two Probability Density Functions (PDFs).
- E.g., the PDF of a Gaussian distribution is $p(x|\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$.

Definition: The Kullback–Leibler (KL) divergence $D_{\mathrm{KL}}(p||q)$.

- For discrete probability distributions, the KL divergence is

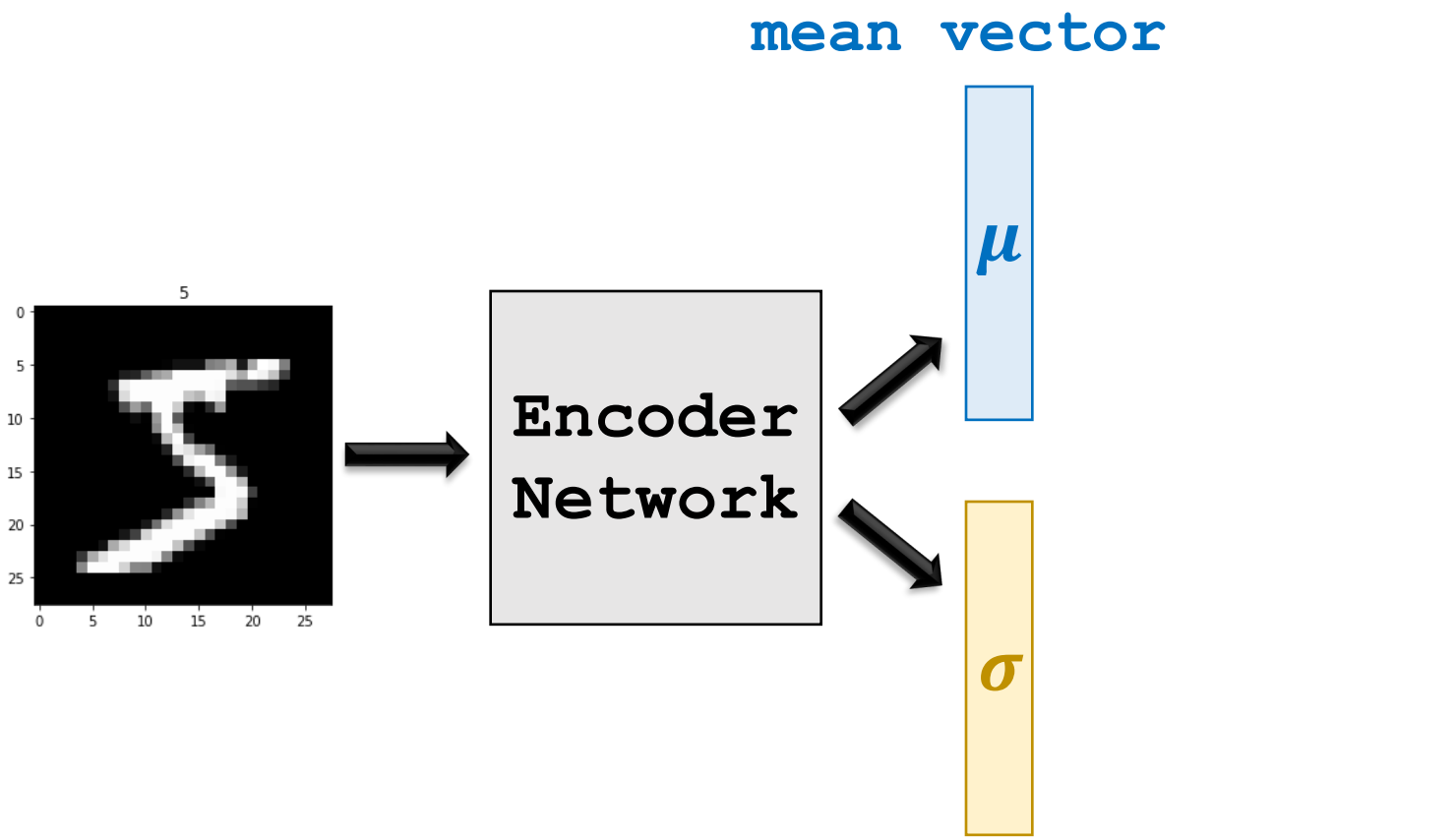
$$D_{\mathrm{KL}}(p||q) = -\sum_i p(i) \log \frac{q(i)}{p(i)}.$$

- For continuous probability distributions, the KL divergence is

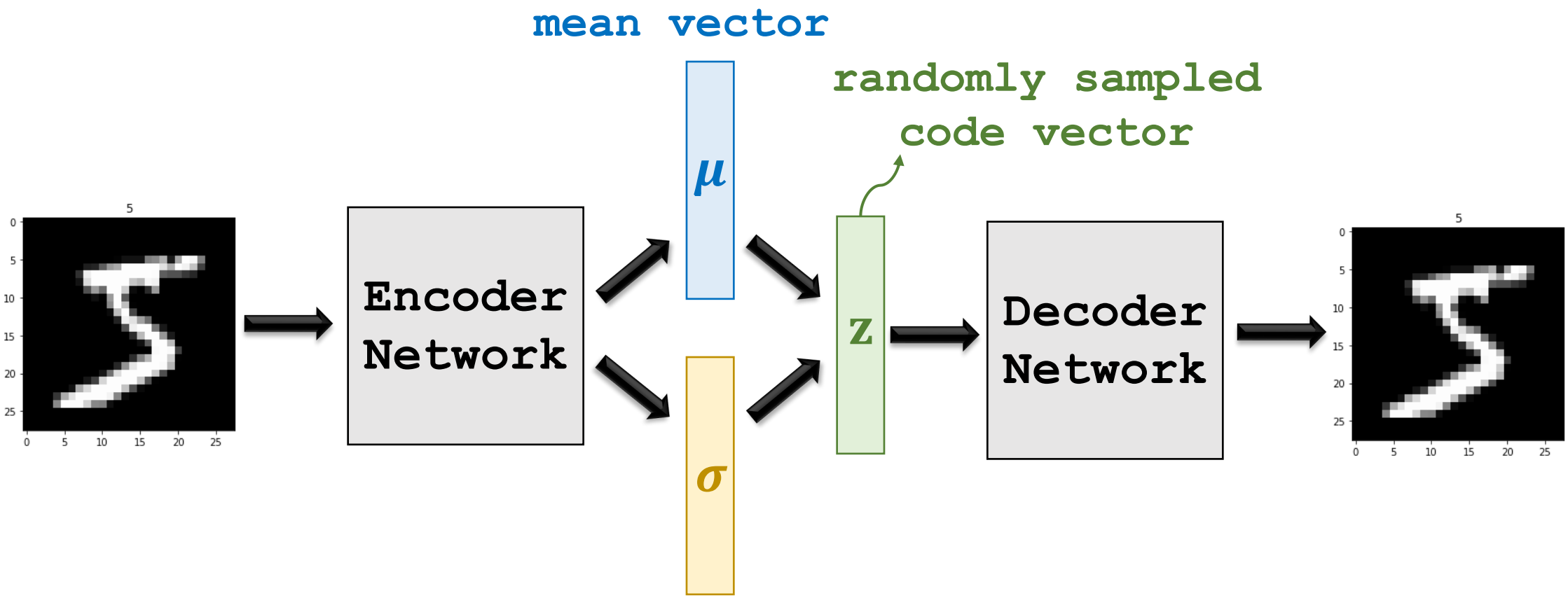
$$D_{\mathrm{KL}}(p||q) = -\int_{-\infty}^{+\infty} p(x) \log \frac{q(x)}{p(x)}\, dx.$$

Variational Autoencoder

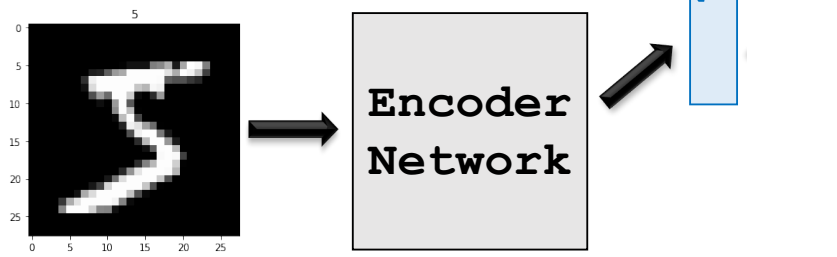
Variational Autoencoder (VAE)



Variational Autoencoder (VAE)



VAE Sampling

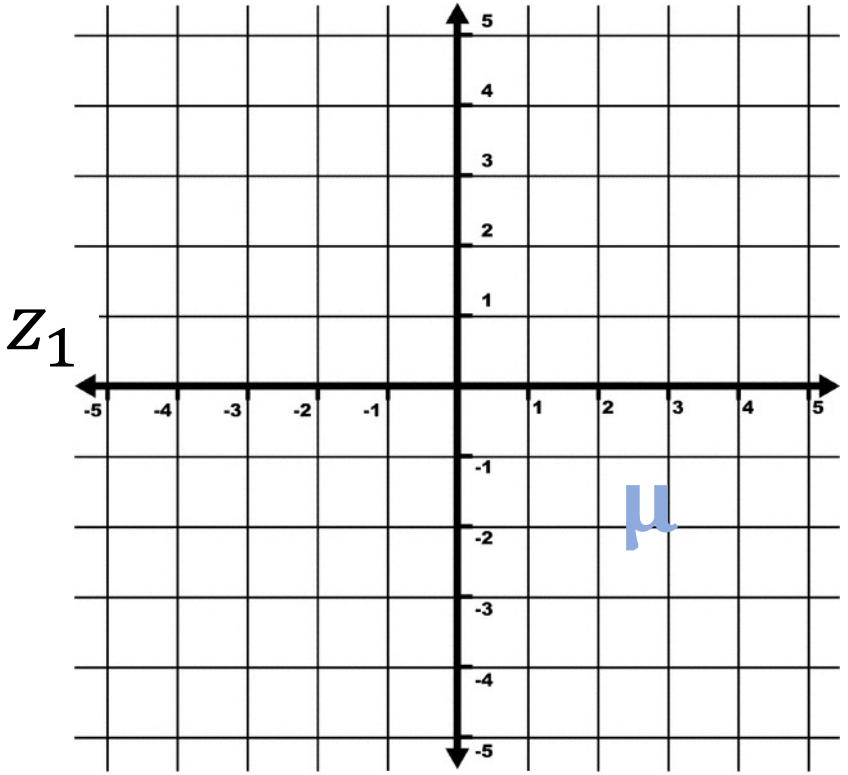


Encoder Network

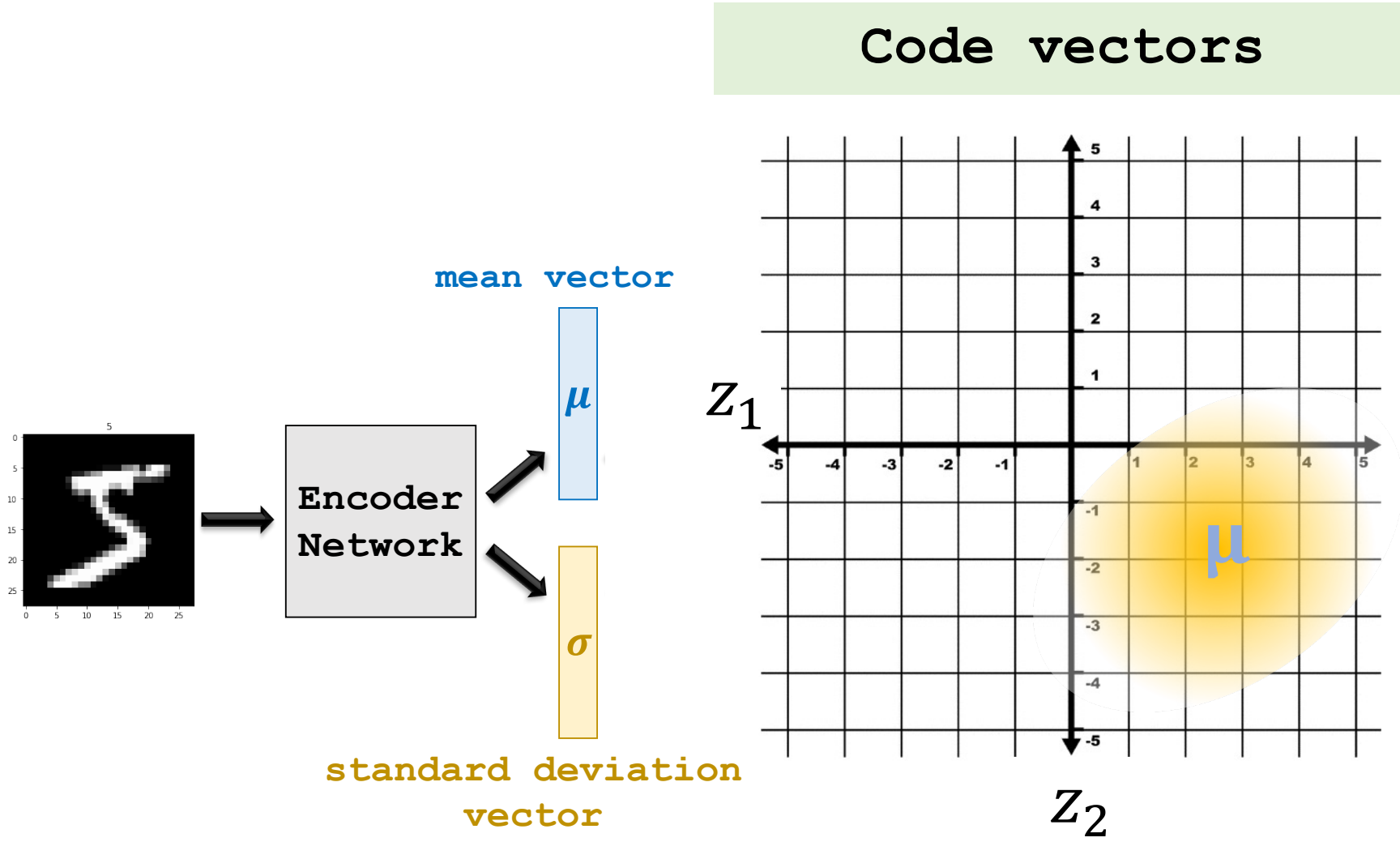
mean vector

μ

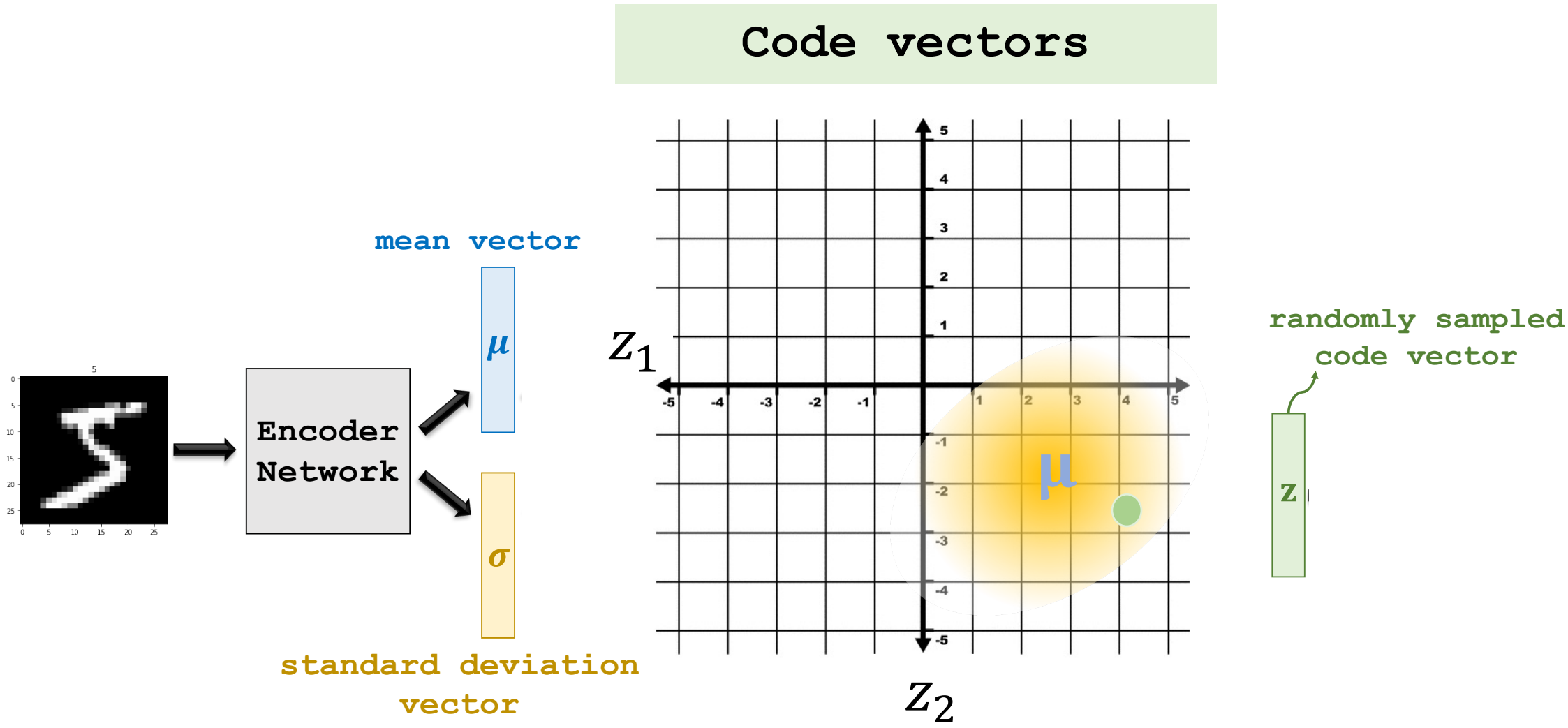
Code vectors



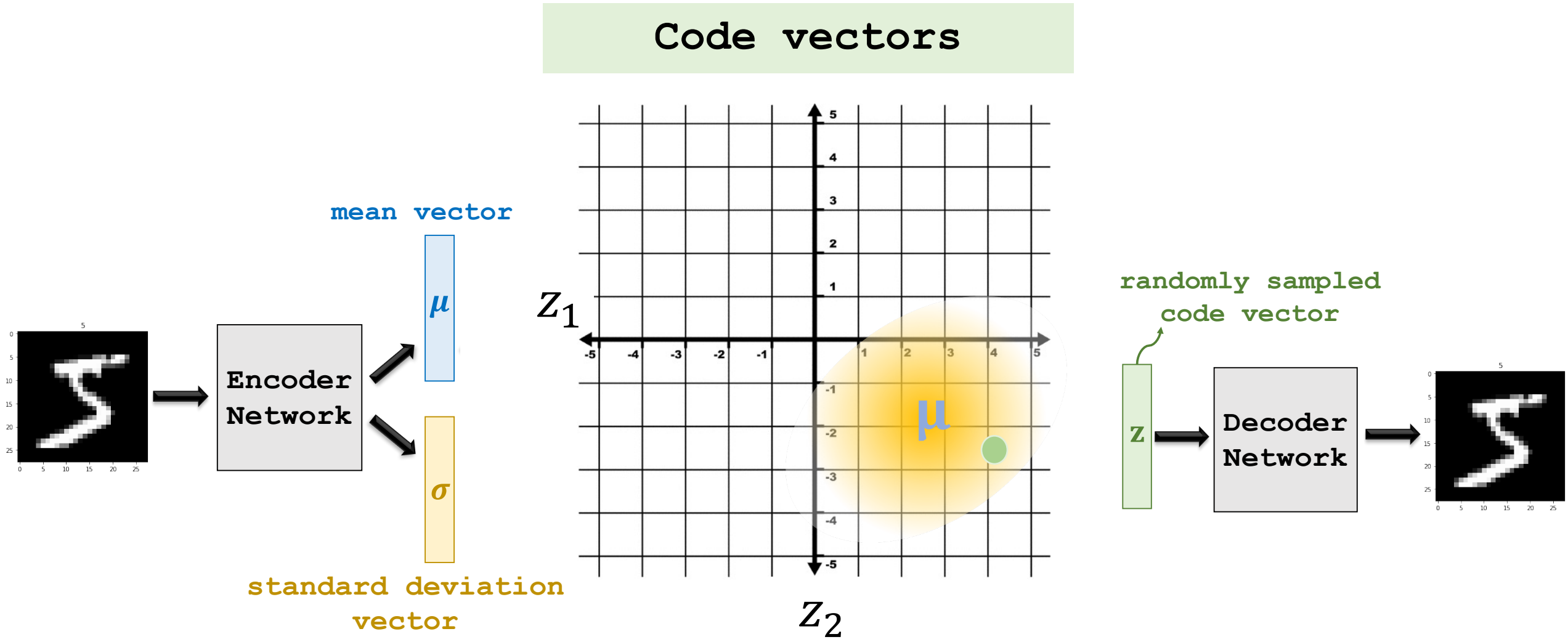
VAE Sampling



VAE Sampling



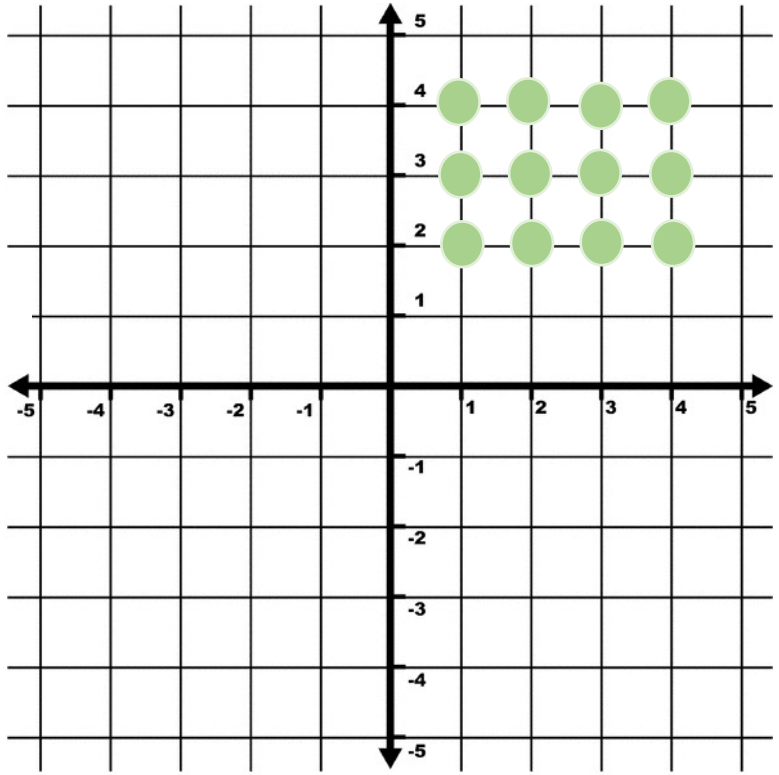
VAE Sampling



Visualize Code Vectors

Visualize code vectors.

Code vectors

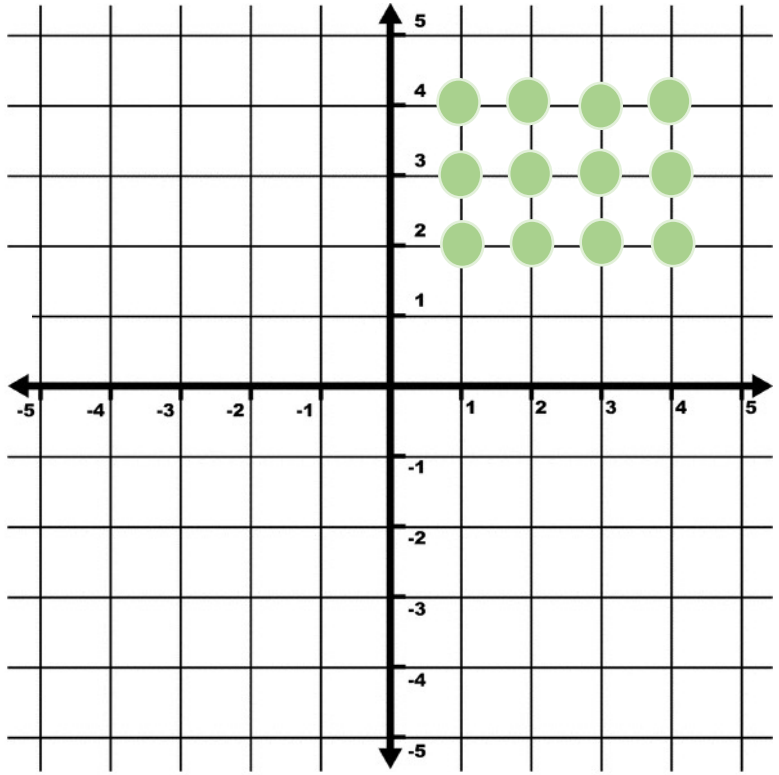


- Get a set of code vectors from the grid:

$$\mathbf{z}_1, \mathbf{z}_2, \mathbf{z}_3, \mathbf{z}_4, \mathbf{z}_5, \cdots$$

Visualize code vectors.

Code vectors



- Get a set of code vectors from the grid:

$$\mathbf{z}_1, \mathbf{z}_2, \mathbf{z}_3, \mathbf{z}_4, \mathbf{z}_5, \cdots$$

- For every code vector $\mathbf{z}_i$, map it to an image using the decoder:

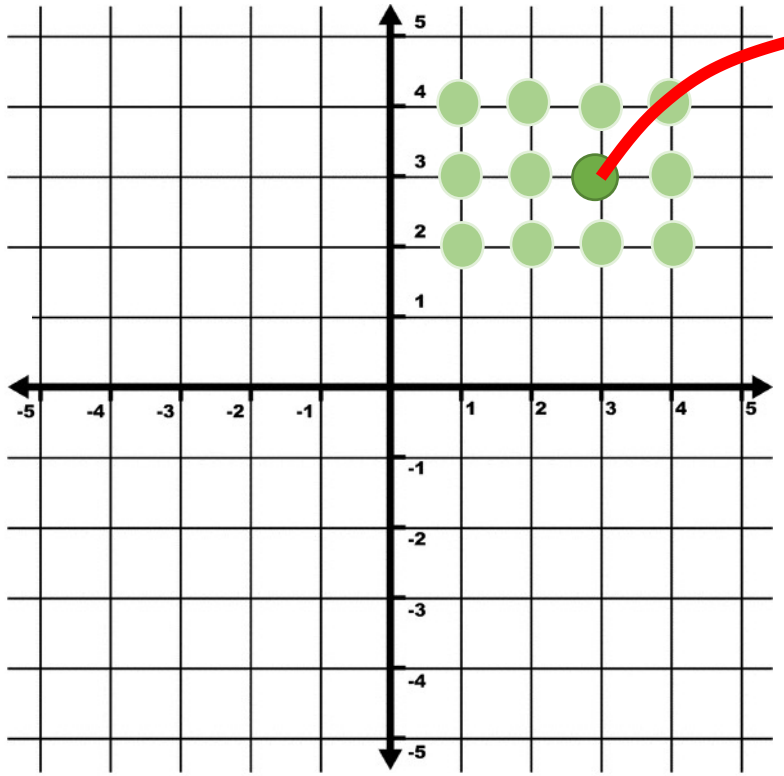
$$\text{image}_i = \text{Decoder}(\mathbf{z}_i).$$

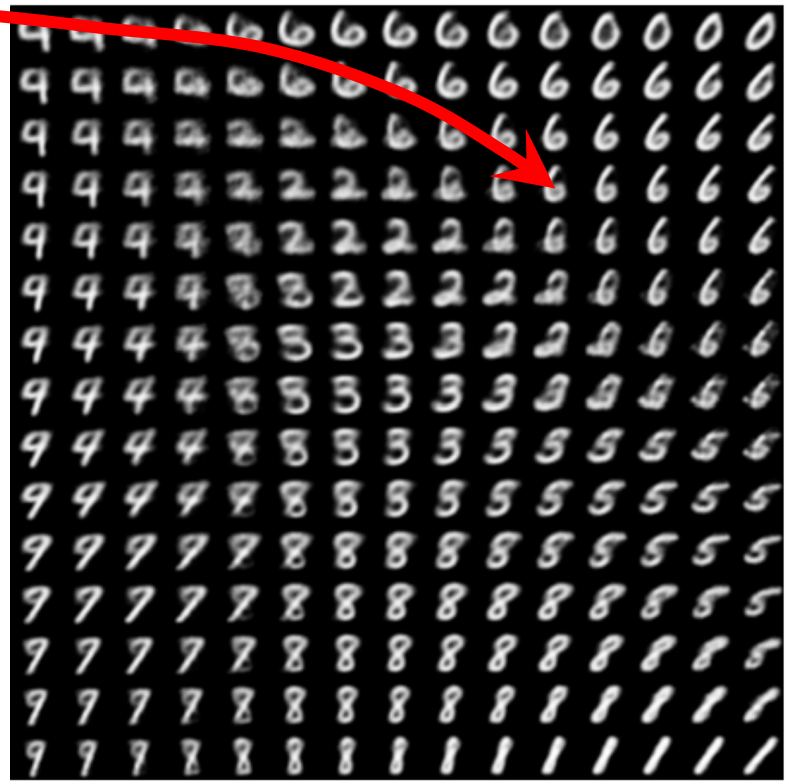
Visualize code vectors.

Code vectors

Images

Decoder





The decoder behaves like a continuous function.

- Function f is continuous if a small change in z (input) results in a small change in $f(z)$ (function value).

- The decoder network is trained to be (almost) continuous.
- If the code vectors $\mathbf{z}$ and $\mathbf{z}'$ are similar, then the images

$$\text{Decoder}(\mathbf{z}) \quad \text{and} \quad \text{Decoder}(\mathbf{z}')$$

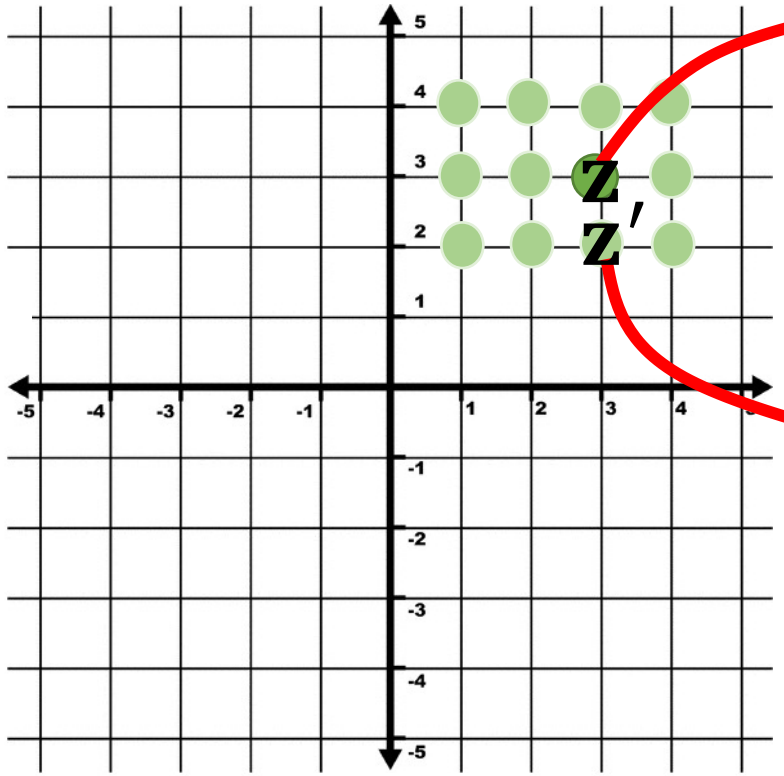
are similar as well.

The decoder behaves like a continuous function.

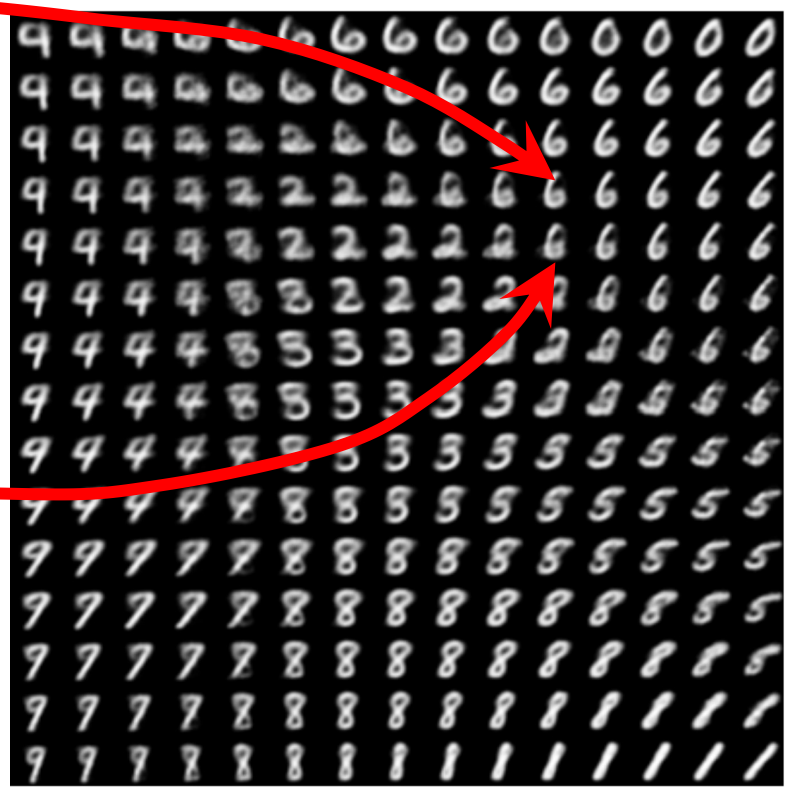
Code vectors

Images

Decoder



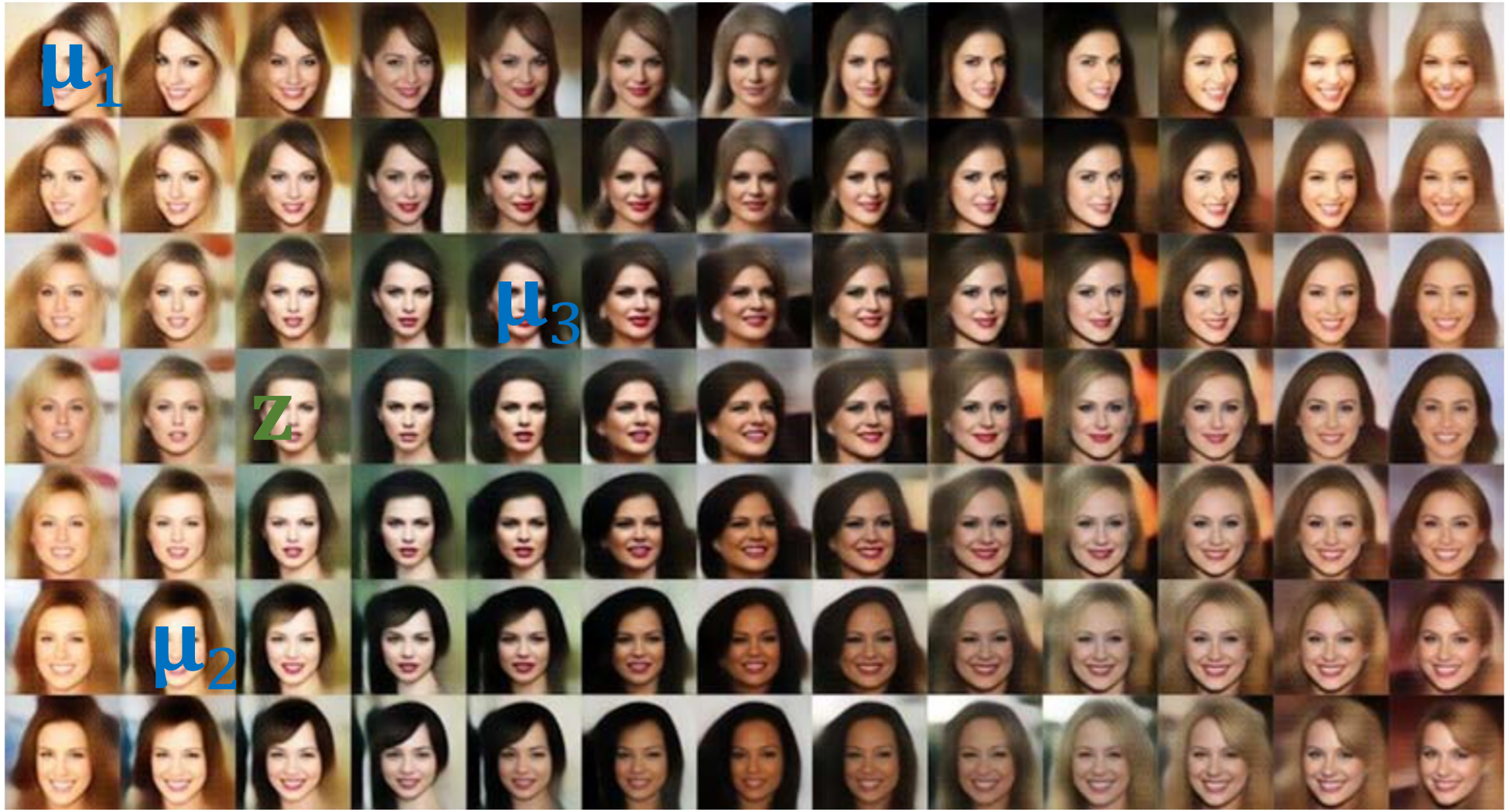
Decoder



The decoder behaves like a continuous function.

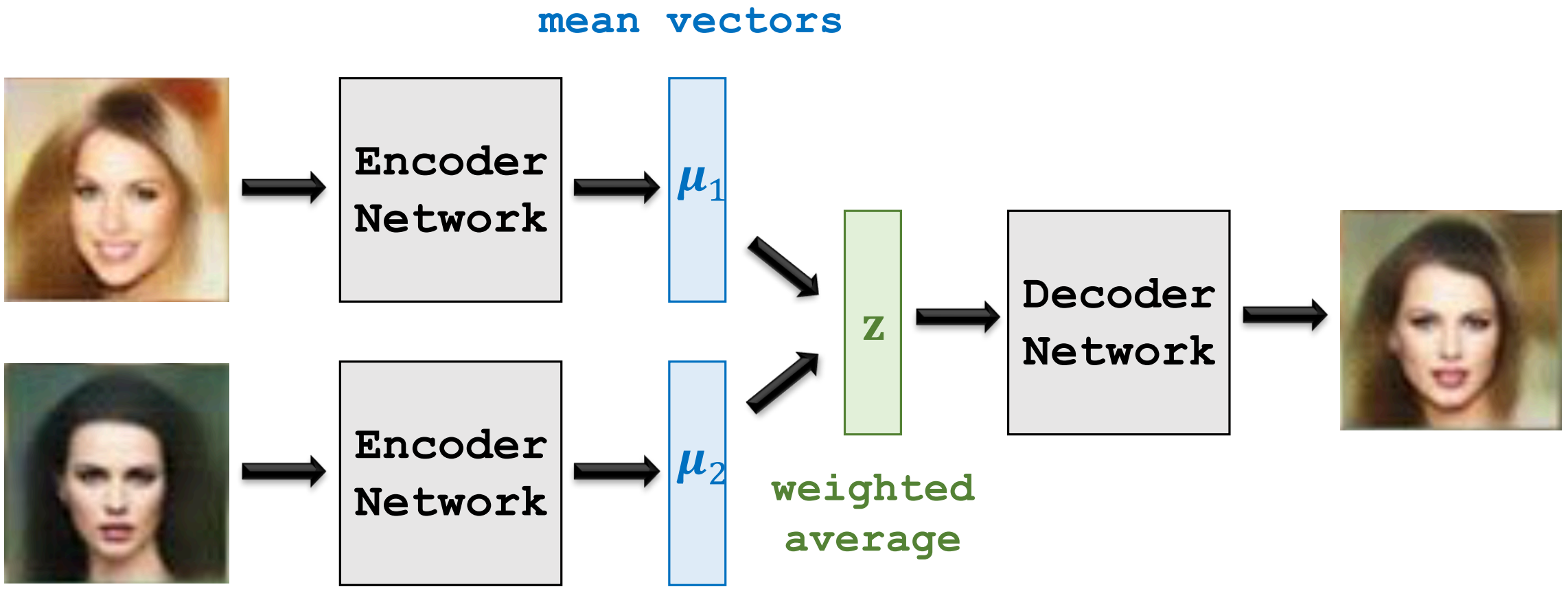
Visualization of the generated faces (by Tom White)

Averaging images via averaging code vectors

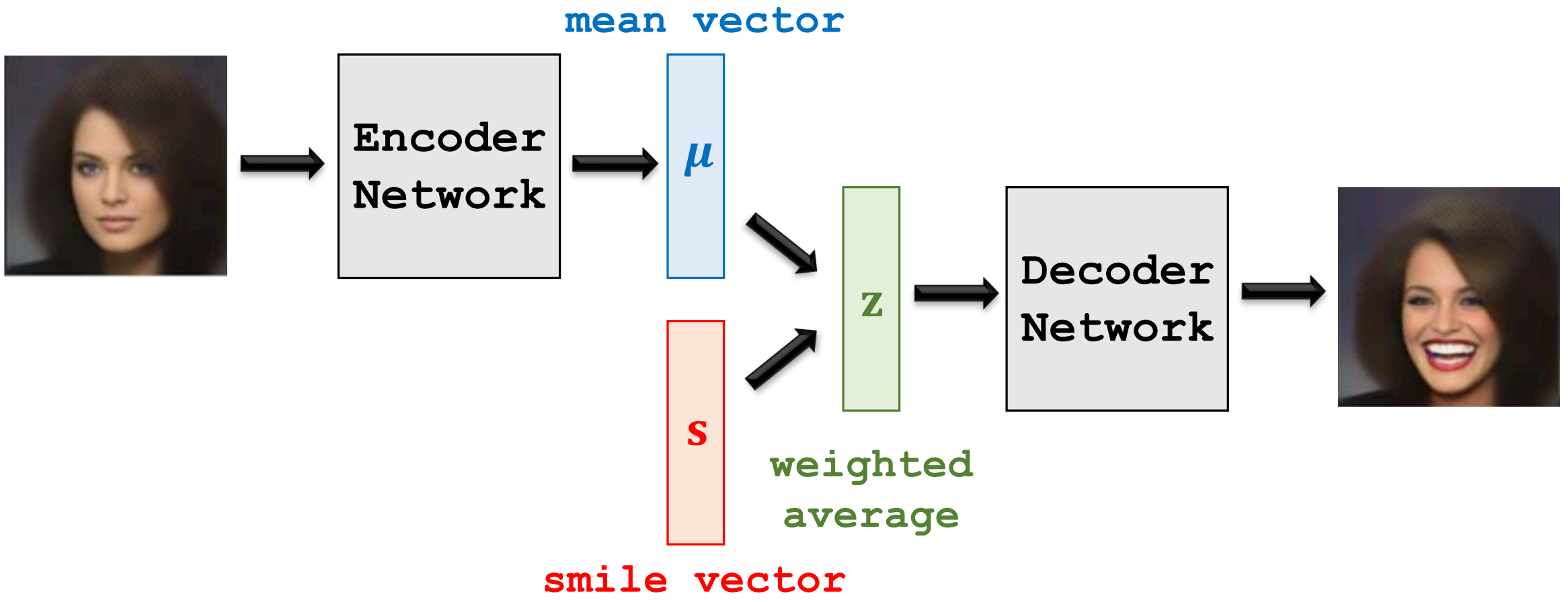


Visualization of the generated faces (by Tom White)

Averaging images via averaging code vectors

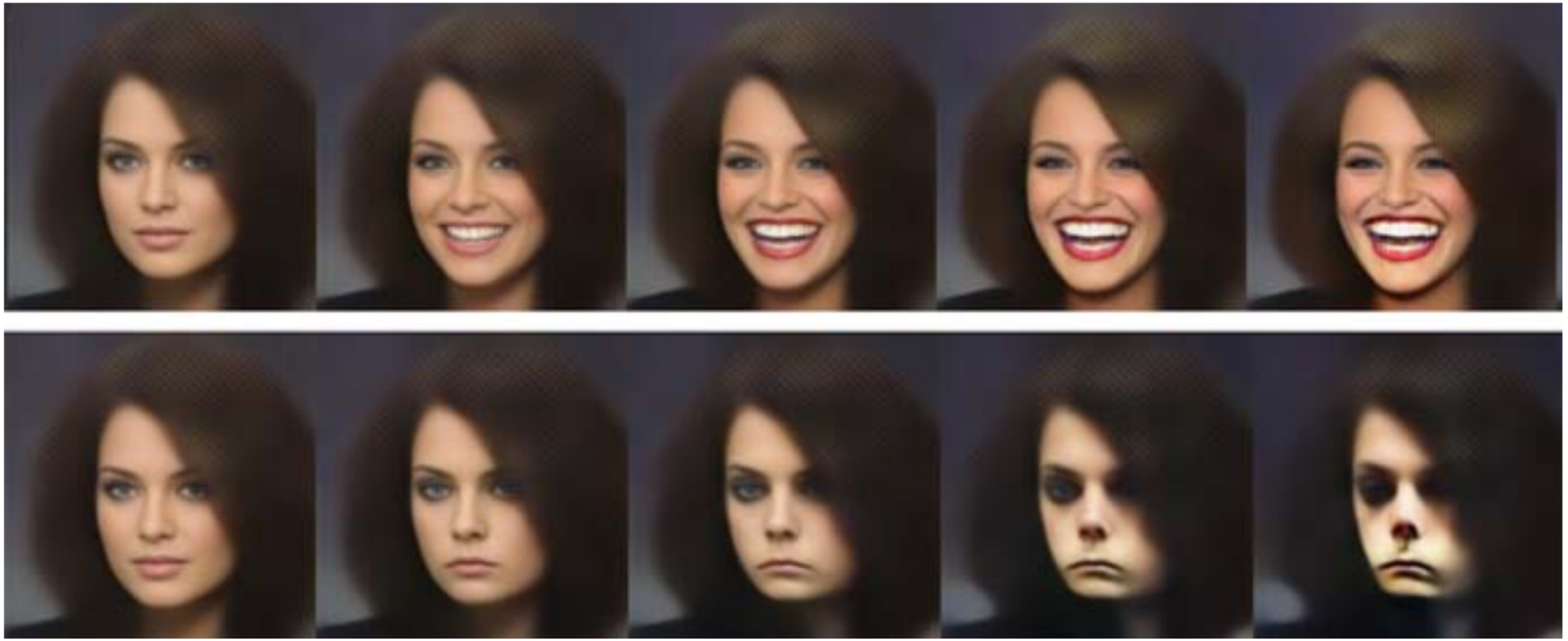


Editing images via editing code vectors



Editing images via editing code vectors

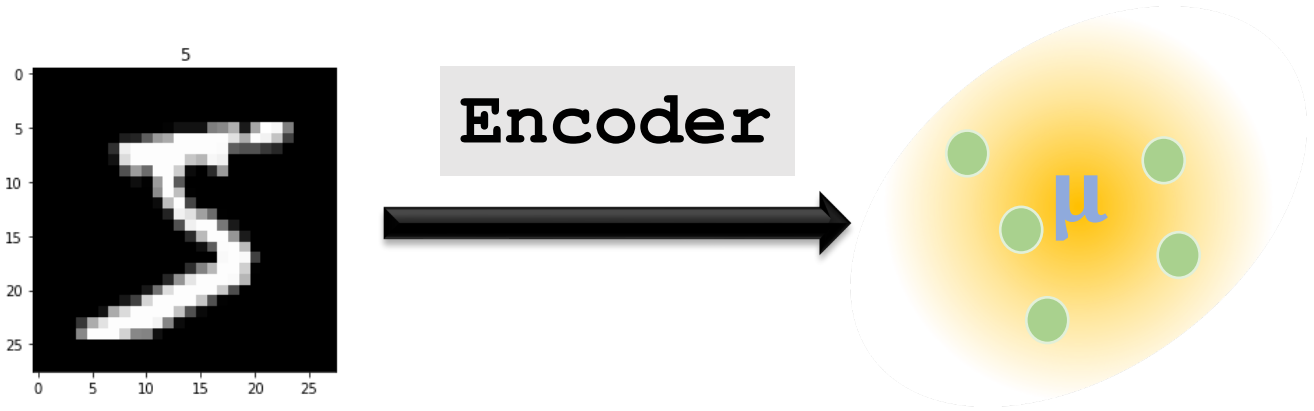
μ $\mu + s$ $\mu + 2s$ $\mu + 3s$ $\mu + 4s$



μ $\mu - s$ $\mu - 2s$ $\mu - 3s$ $\mu - 4s$

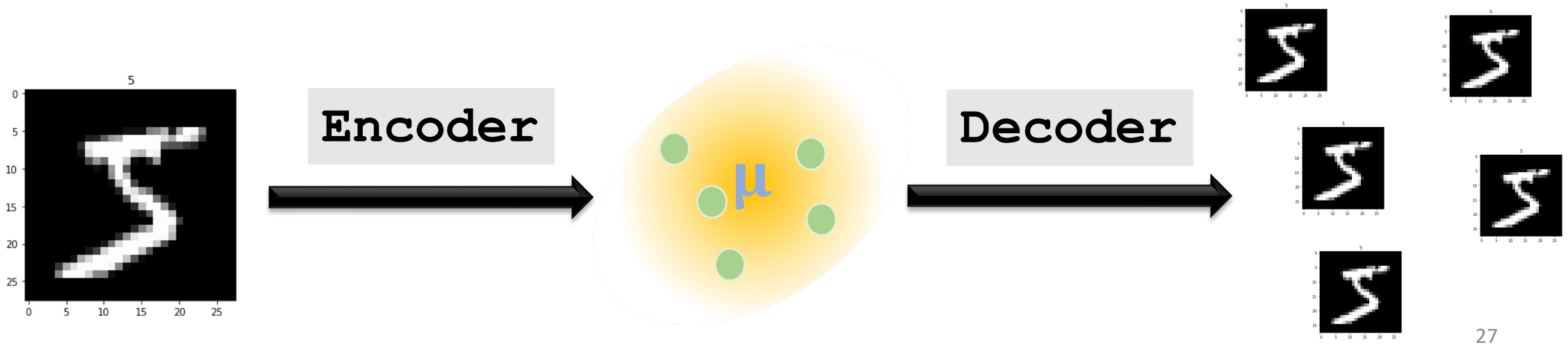
What makes the decoder continuous?

- Given $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$, sample $\mathbf{z}_i \sim \mathcal{N}\left(\boldsymbol{\mu}, \mathrm{diag}\left(\boldsymbol{\sigma}^2\right)\right)$, for $i = 1, 2, 3, \cdots$.



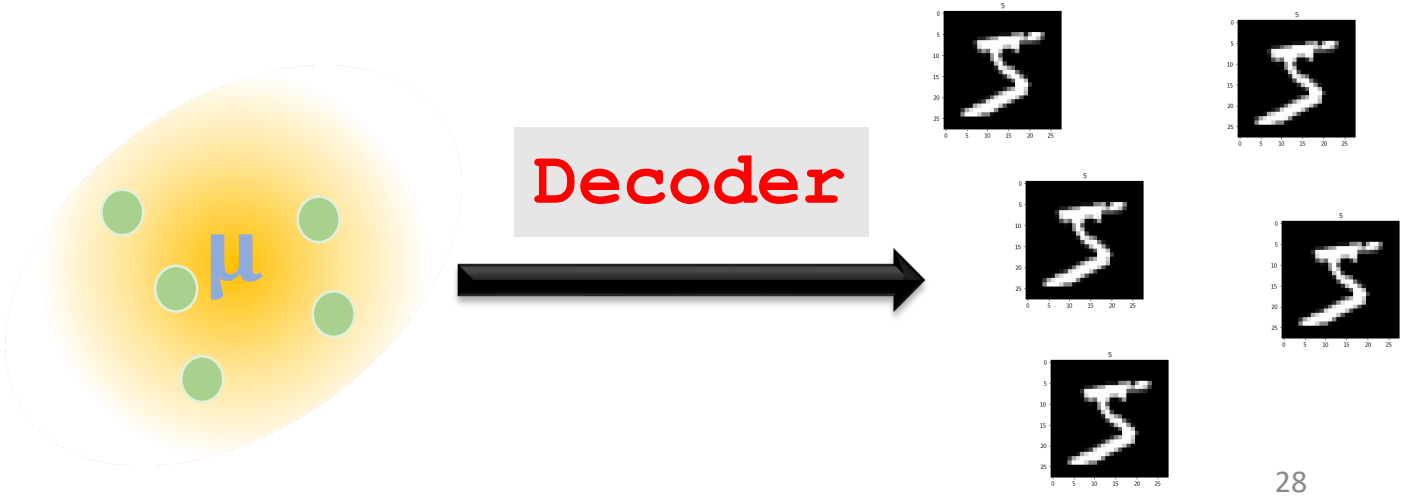
What makes the decoder continuous?

- Given $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$, sample $\mathbf{z}_i \sim \mathcal{N}\left(\boldsymbol{\mu}, \mathrm{diag}\left(\boldsymbol{\sigma}^2\right)\right)$, for $i = 1, 2, 3, \cdots$.
- Generate images, $\mathrm{Decoder}(\mathbf{z}_1), \mathrm{Decoder}(\mathbf{z}_2), \mathrm{Decoder}(\mathbf{z}_3), \cdots$, have the same target (which is the original image).



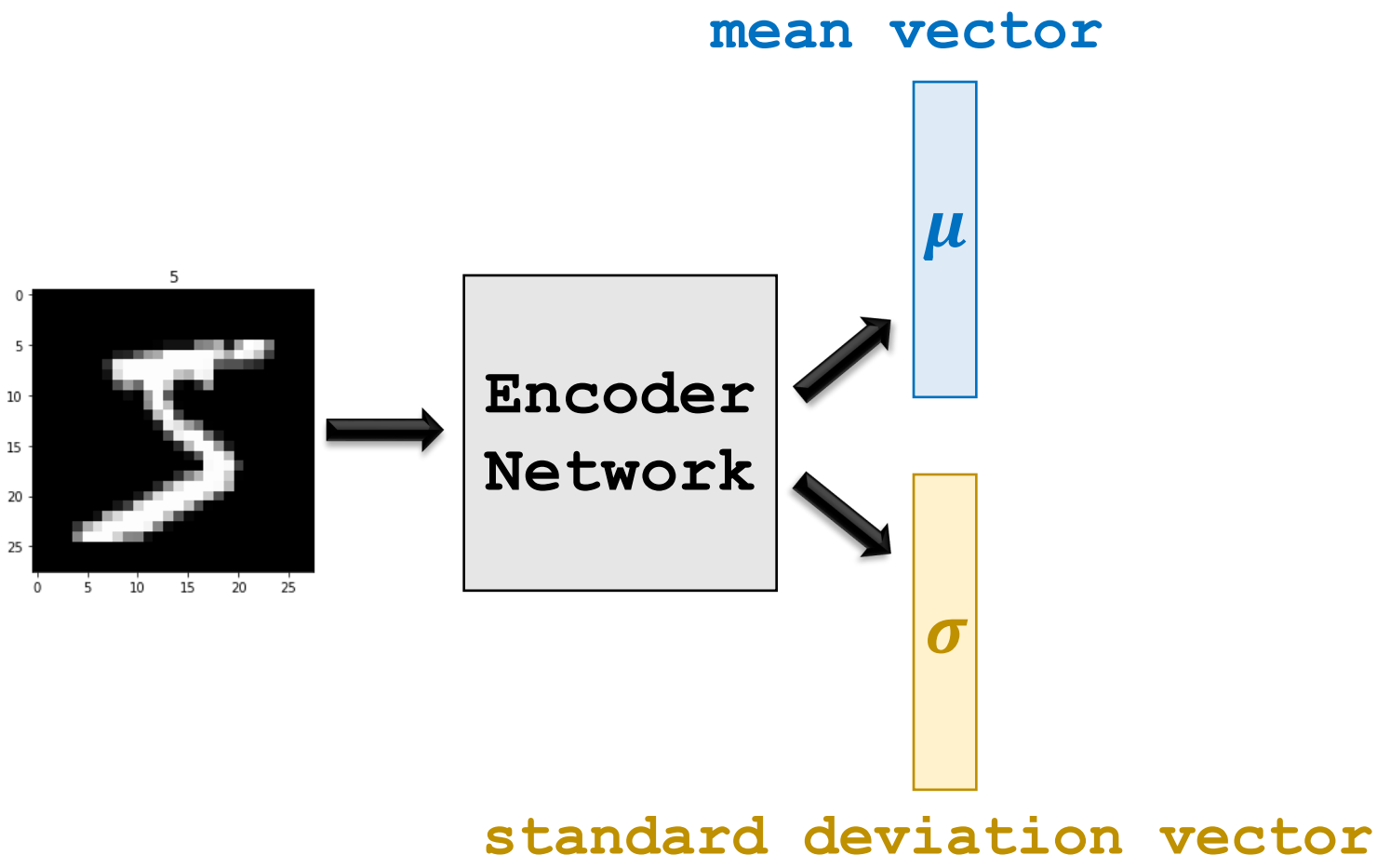
What makes the decoder continuous?

- Inputs (code vectors) are similar. (Because they are in a small neighborhood around μ.)
- Outputs (generated images) are encouraged to be similar. (Because they have the same target.)
- ➔ The decoder is trained to behave like a continuous function.



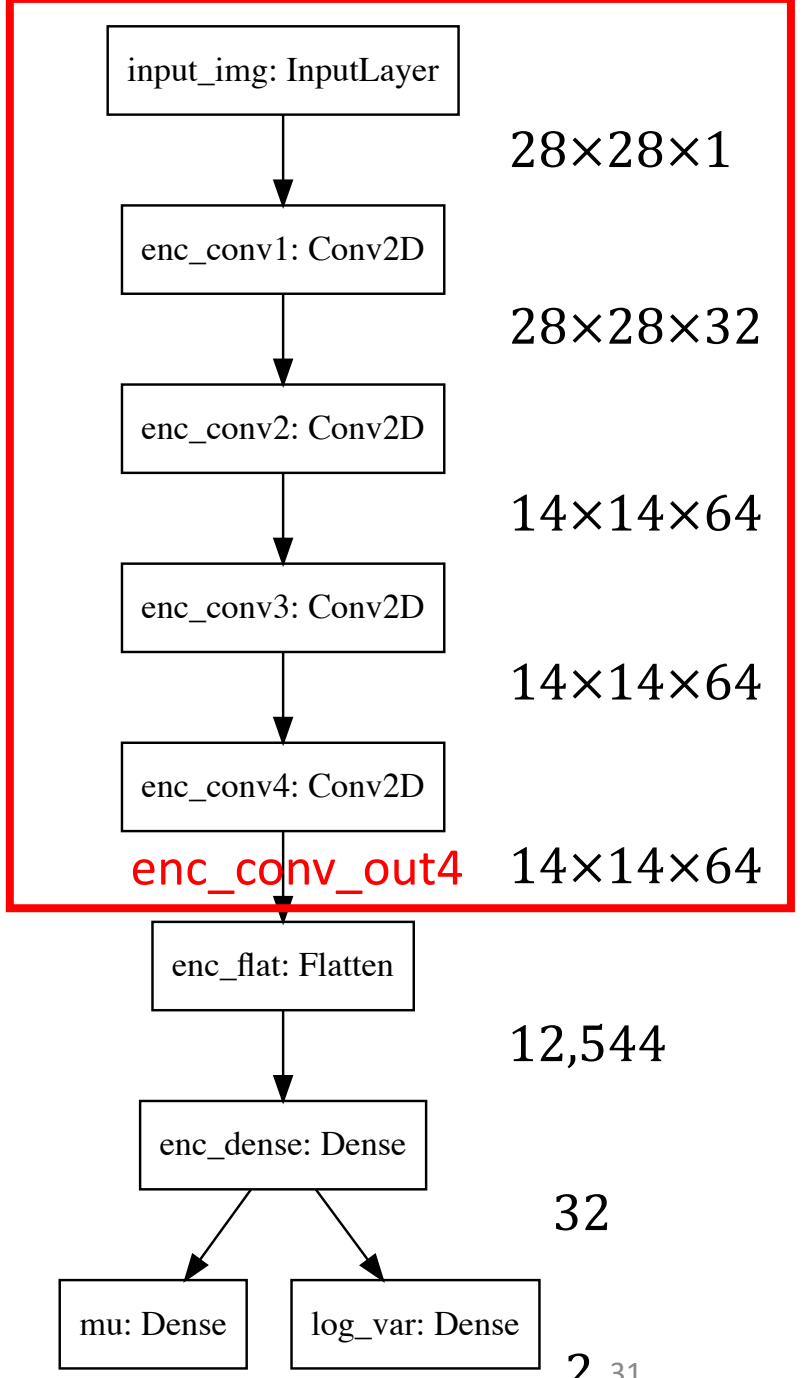
Build the Networks

1. The Encoder Network



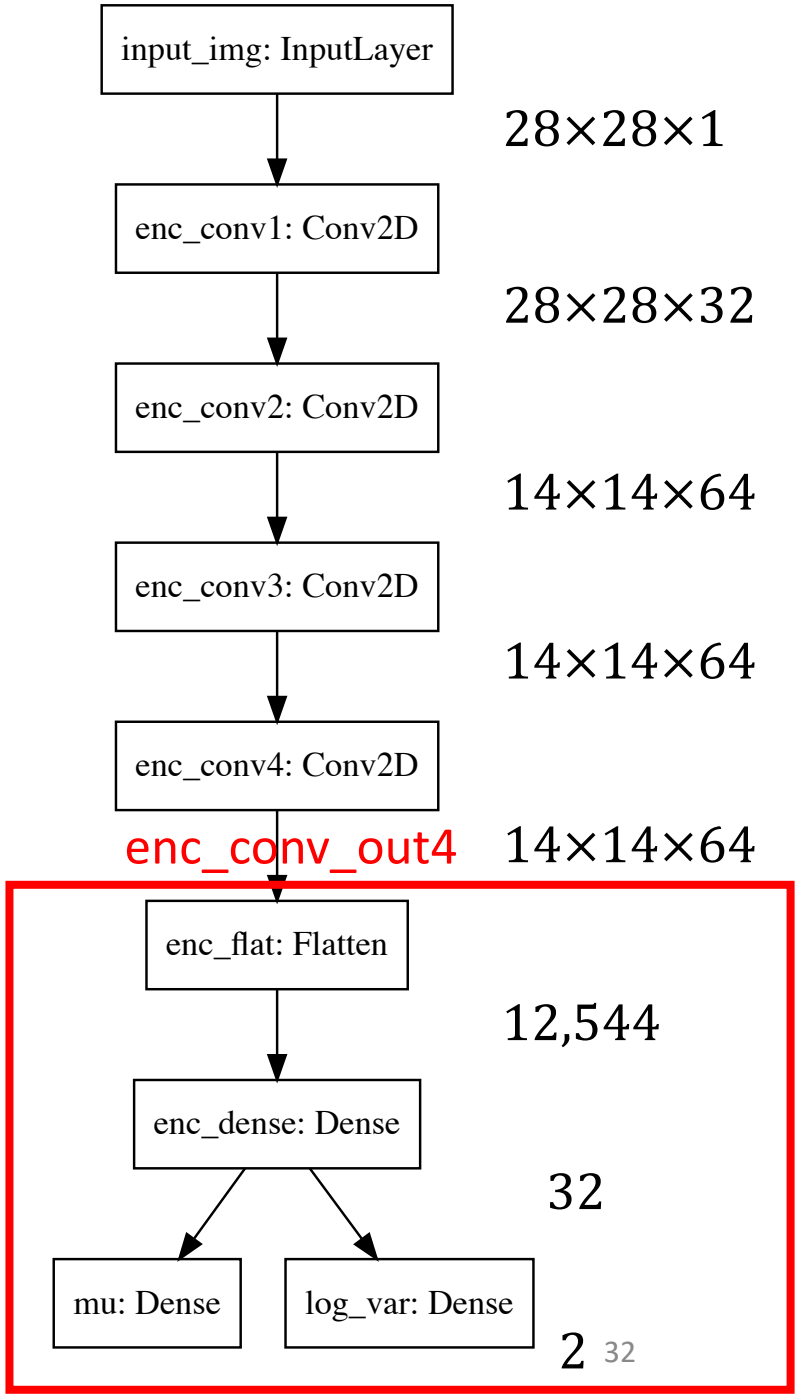
1. The Encoder Network

```
from keras.layers import Input, Conv2D, Flatten, Dense
from keras import models

shape_z = 2

# define convolutional layers
enc_conv1 = Conv2D(32, 3, padding='same',
                   activation='relu', name='enc_conv1')
enc_conv2 = Conv2D(64, 3, padding='same', activation='relu',
                      strides=(2, 2), name='enc_conv2')
enc_conv3 = Conv2D(64, 3, padding='same',
                   activation='relu', name='enc_conv3')
enc_conv4 = Conv2D(64, 3, padding='same',
                   activation='relu', name='enc_conv4')

input_img = Input(shape=(28, 28, 1), name='input_img')
enc_conv_out1 = enc_conv1(input_img)
enc_conv_out2 = enc_conv2(enc_conv_out1)
enc_conv_out3 = enc_conv3(enc_conv_out2)
enc_conv_out4 = enc_conv4(enc_conv_out2)
```



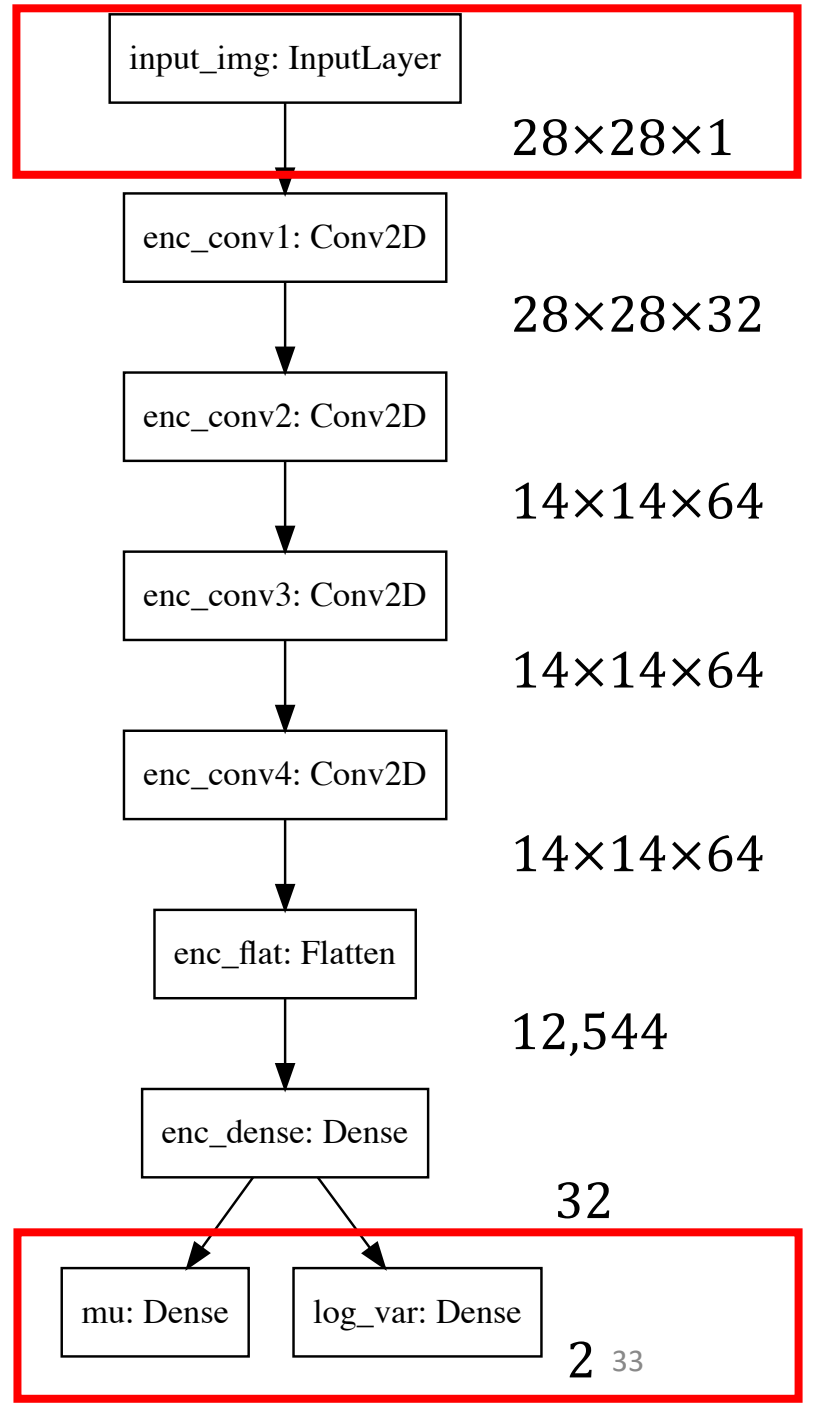
1. The Encoder Network

```
# define flatten and dense layers
enc_flat = Flatten(name='enc_flat')
enc_dense = Dense(32, activation='relu',
                  name='enc_dense')
enc_mu = Dense(shape_z, name='mu')
enc_log_var = Dense(shape_z, name='log_var')

enc_flat_out = enc_flat(enc_conv_out4)
enc_dense_out = enc_dense(enc_flat_out)
mu = enc_mu(enc_dense_out)
log_var = enc_log_var(enc_dense_out)
```

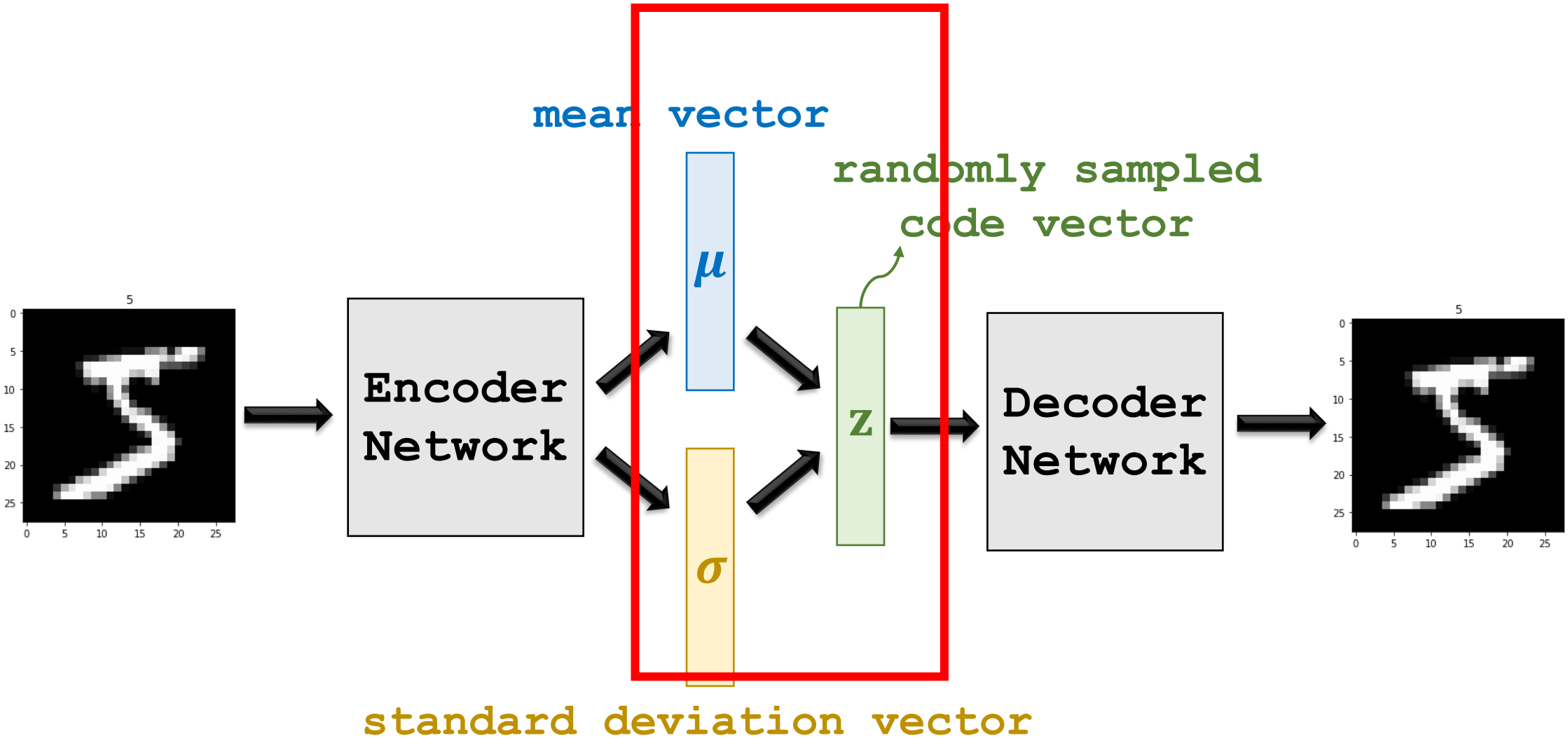


1. The Encoder Network

```
# define flatten and dense layers
enc_flat = Flatten(name='enc_flat')
enc_dense = Dense(32, activation='relu',
                  name='enc_dense')
enc_mu = Dense(shape_z, name='mu')
enc_log_var = Dense(shape_z, name='log_var')

enc_flat_out = enc_flat(enc_conv_out4)
enc_dense_out = enc_dense(enc_flat_out)
mu = enc_mu(enc_dense_out)
log_var = enc_log_var(enc_dense_out)

# model
encoder = models.Model(inputs=input_img,
                       outputs=[mu, log_var],
                       name='encoder')
```



2. The Sampling Network

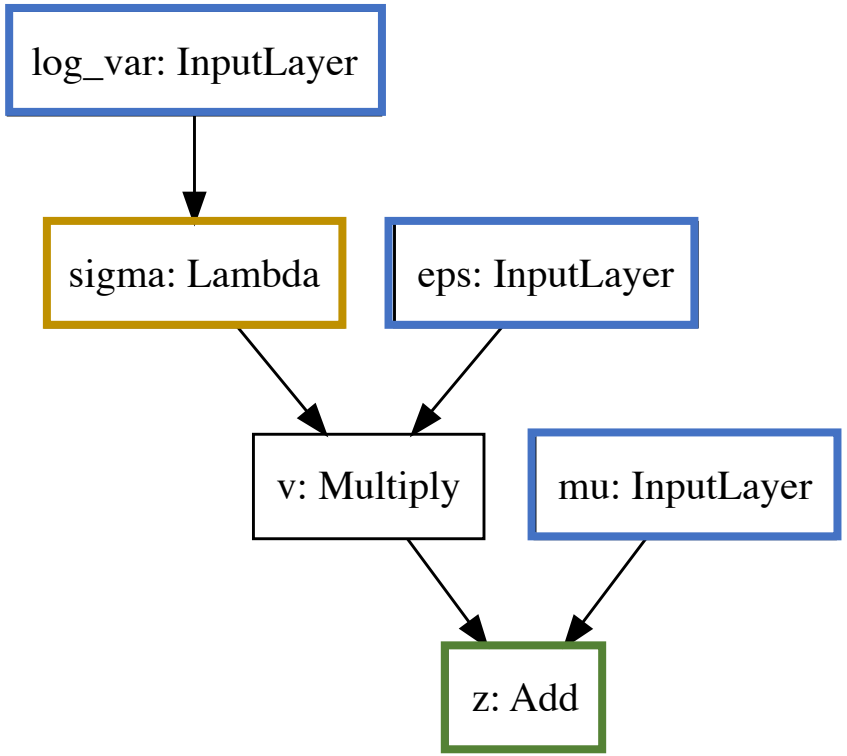


2. The Sampling Network

- The i-th element of $\mathbf{z}$ (denote z_i) is randomly sampled from $\mathcal{N}\left(\mu_i, \sigma_i^2\right)$.
- Equivalently,
  - Randomly sample scalar variable ϵ_i from $\mathcal{N}(0,1)$.
  - Let $v_i = \sigma_i \cdot \epsilon_i$. (Equivalently, v_i is sampled from $\mathcal{N}\left(0, \sigma_i^2\right)$.)
  - $z_i = \mu_i + v_i$.

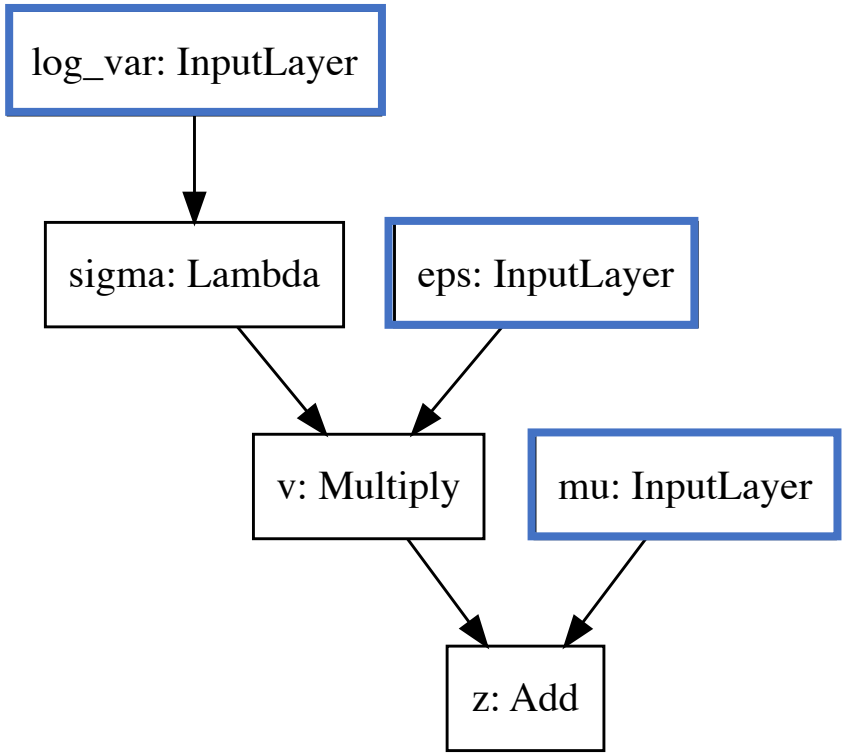
2. The Sampling Network

- Shape of $\mathbf{z}$: k ($= 2$ in our implementation).
- Input of the sampling network.
  - Log variance: $\boldsymbol{\eta}$ (k-dim vector).
  - Mean: $\boldsymbol{\mu}$ (k-dim vector).
  - $\boldsymbol{\epsilon}$: randomly sampled from $\mathcal{N}(\mathbf{0}, \mathbf{I}_k)$.
- The sampling network:
  - $\boldsymbol{\sigma} = e^{0.5\boldsymbol{\eta}}$ (k-dim vector).
  - $\mathbf{v} = \boldsymbol{\sigma} \circ \boldsymbol{\epsilon}$ (k-dim vector).
  - $\mathbf{z} = \boldsymbol{\mu} + \mathbf{v}$ (k-dim vector).
  - Output $\mathbf{z}$.



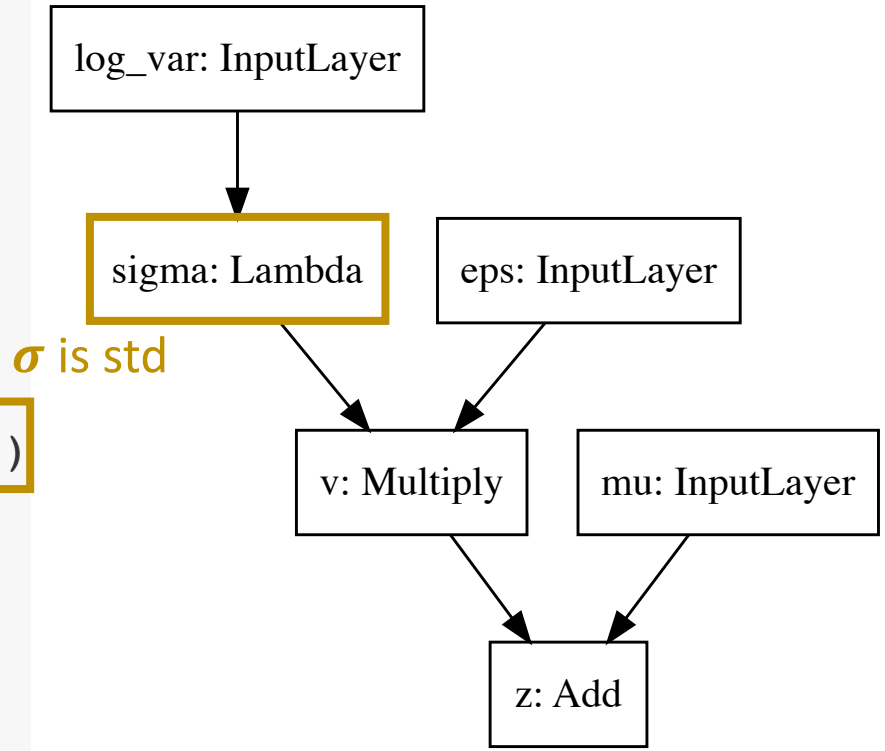
2. The Sampling Network

```
from keras.layers import Lambda, Multiply, Add
from keras import backend as K

# inputs
mu = Input(shape=(shape_z,), name='mu')
log_var = Input(shape=(shape_z,), name='log_var')
eps = Input(shape=(shape_z,), name='eps')

# layers
sigma = Lambda(lambda t: K.exp(.5*t), name='sigma')(log_var)
v = Multiply(name='v')([sigma, eps])
z = Add(name='z')([mu, v])

# model
sampling = models.Model(inputs=[mu, log_var, eps],
                        outputs=z,
                        name='sampling')
```



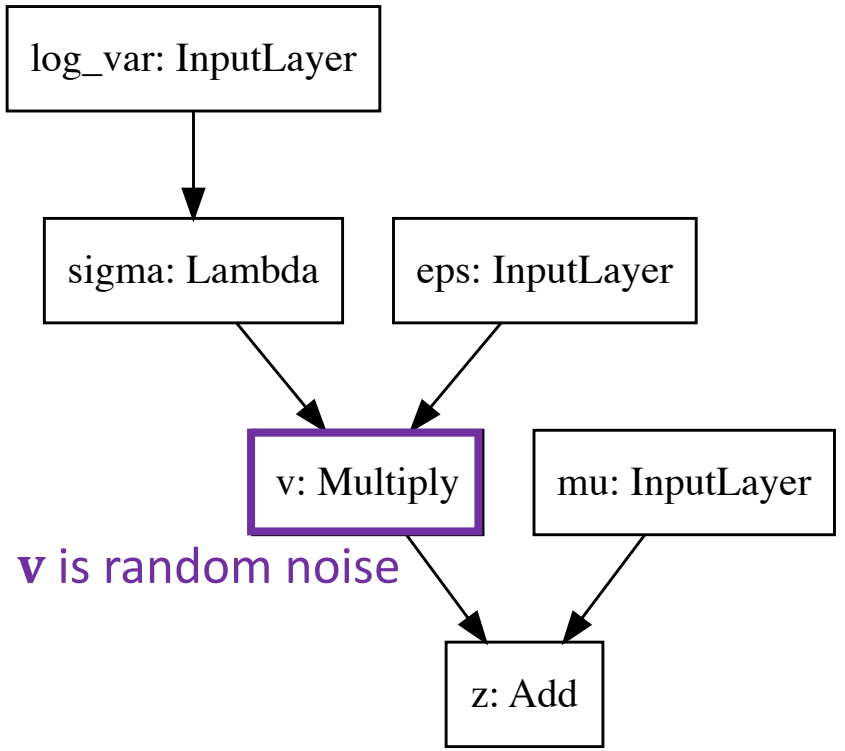
2. The Sampling Network

```
from keras.layers import Lambda, Multiply, Add
from keras import backend as K

# inputs
mu = Input(shape=(shape_z,), name='mu')
log_var = Input(shape=(shape_z,), name='log_var')
eps = Input(shape=(shape_z,), name='eps')

# layers
sigma = Lambda(lambda t: K.exp(.5*t), name='sigma')(log_var)
v = Multiply(name='v')([sigma, eps])
z = Add(name='z')([mu, v])

# model
sampling = models.Model(inputs=[mu, log_var, eps],
                        outputs=z,
                        name='sampling')
```



2. The Sampling Network

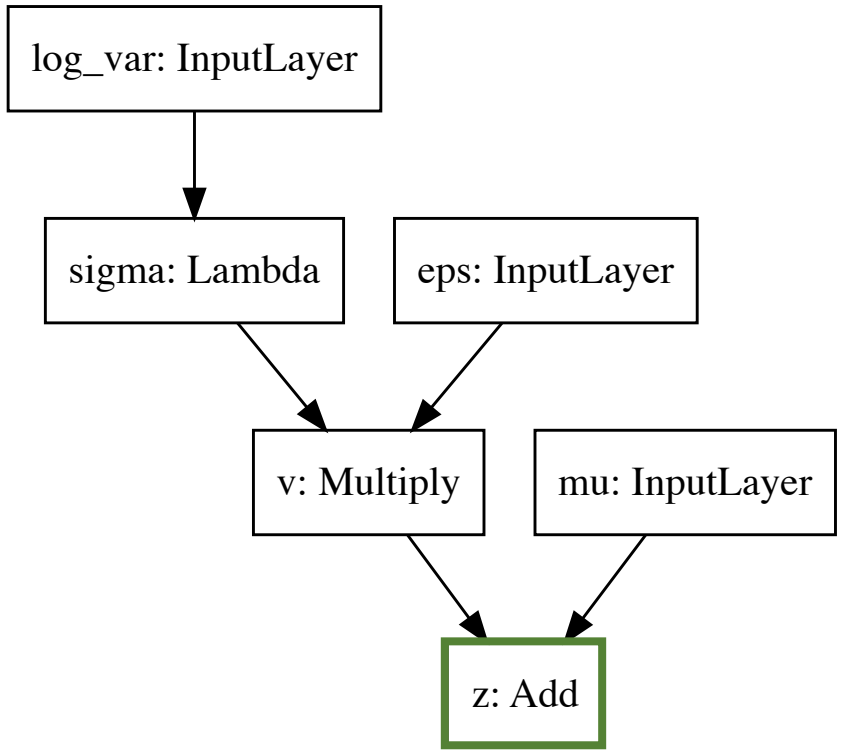
```
from keras.layers import Lambda, Multiply, Add
from keras import backend as K

# inputs
mu = Input(shape=(shape_z,), name='mu')
log_var = Input(shape=(shape_z,), name='log_var')
eps = Input(shape=(shape_z,), name='eps')

# layers
sigma = Lambda(lambda t: K.exp(.5*t), name='sigma')(log_var)
v = Multiply(name='v')([sigma, eps])
z = Add(name='z')([mu, v])

# model
sampling = models.Model(inputs=[mu, log_var, eps],
                        outputs=z,
                        name='sampling')
```



v is random noise

2. The Sampling Network

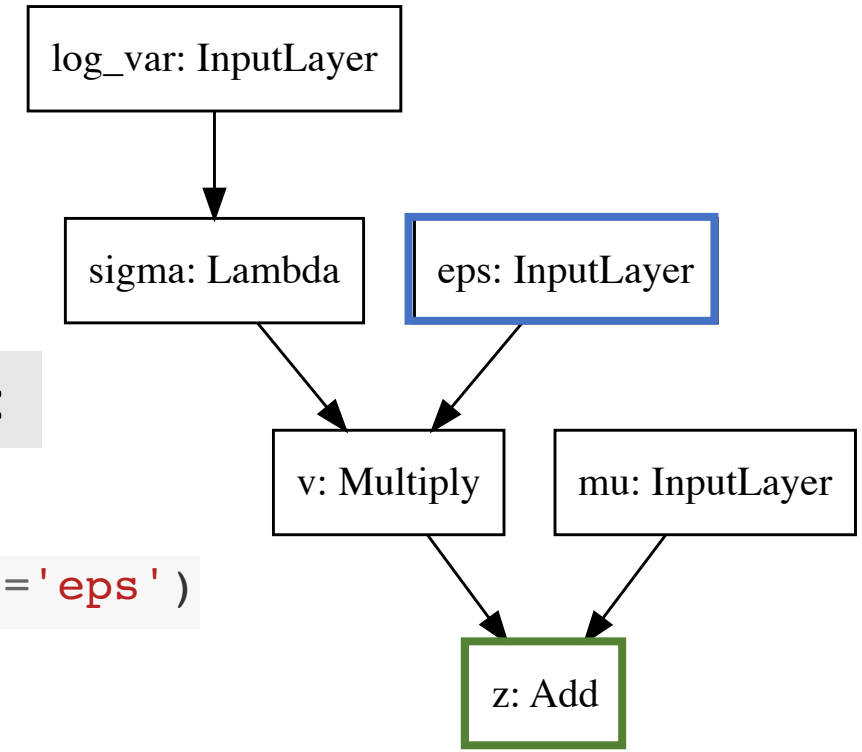
```
from keras.layers import Lambda, Multiply, Add
from keras import backend as K

# inputs
mu = Input(shape=(shape_z,), name='mu')
log_var = Input(shape=(shape_z,), name='log_var')
eps = Input(shape=(shape_z,), name='eps')

# layers
sigma = Lambda(lambda t: K.exp(.5*t), name='sigma')(log_var)
v = Multiply(name='v')([sigma, eps])
z = Add(name='z')([mu, v])

# model
sampling = models.Model(inputs=[mu, log_var, eps],
                        outputs=z,
                        name='sampling')
```



z is the sampled latent vector

2. The Sampling Network

Question: Where is the random sampling $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_k)$?

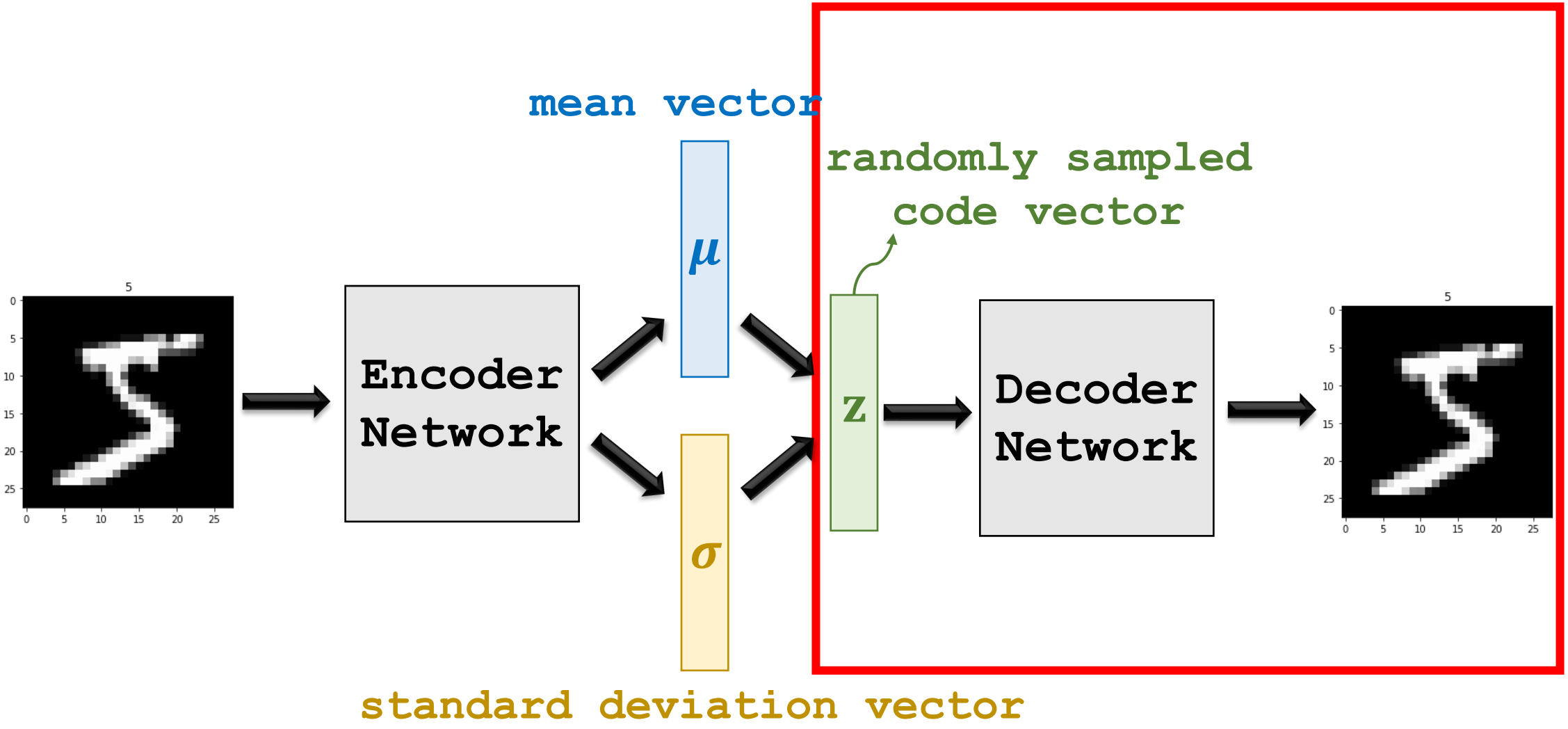


During training, set the input layer "eps" to the following code:

```
eps = Input(tensor=K.random_normal(shape=(n, shape_z)), name='eps')
```

z is the sampled latent vector

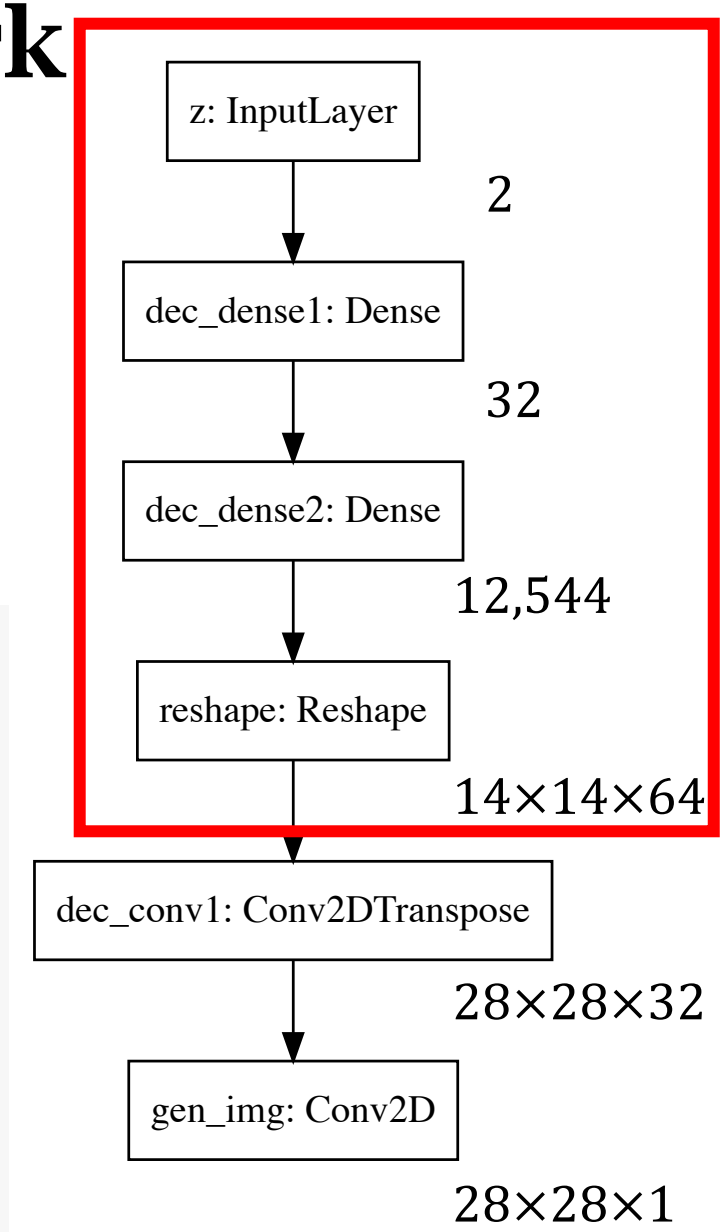
3. The Decoder Network



3. The Decoder Network

```
from keras import backend as K
import numpy

shape_before_flattening = K.int_shape(enc_conv_out4)[1:]
shape_after_flattening = numpy.prod(shape_before_flattening)
```

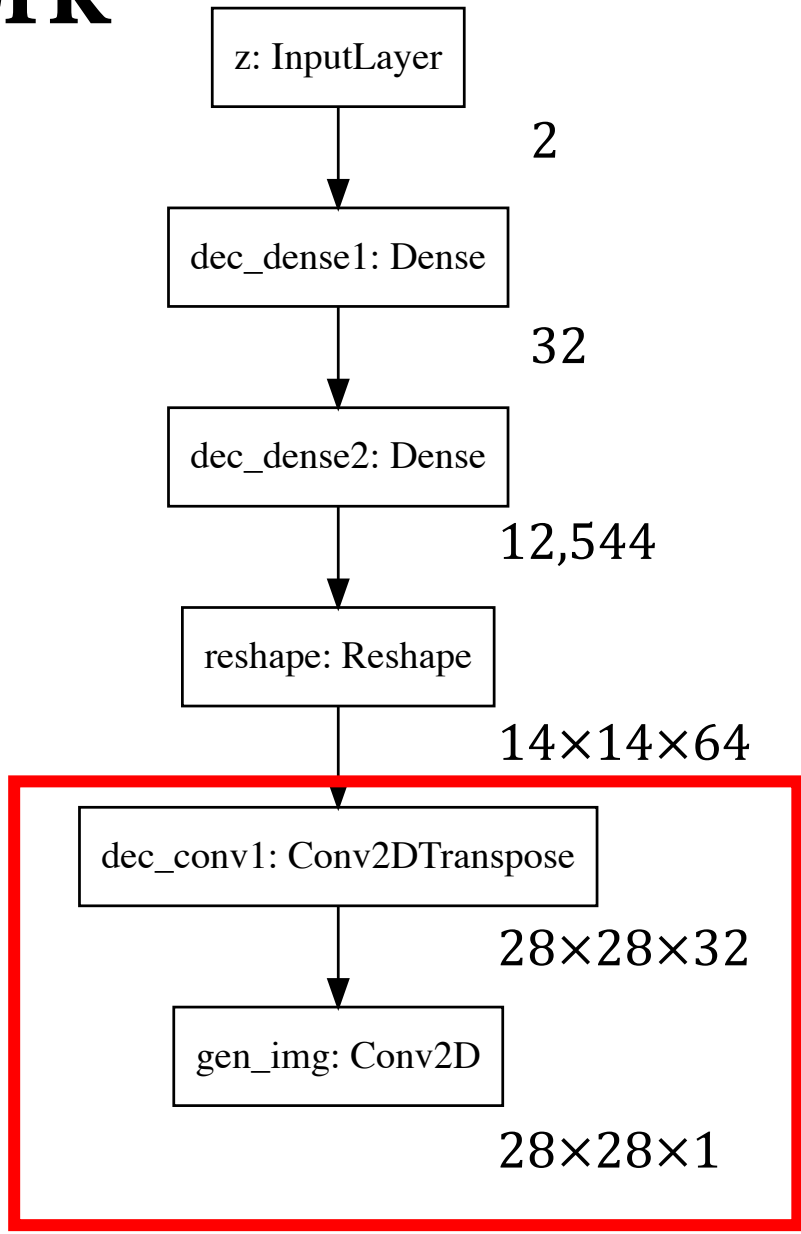
```
from keras.layers import Dense, Reshape, Conv2D, Conv2DTranspose

dec_dense1 = Dense(32, activation='relu', name='dec_dense1')
dec_dense2 = Dense(shape_after_flattening,
                   activation='relu', name='dec_dense2')
dec_reshape = Reshape(shape_before_flattening, name='reshape')

z = Input(shape=(shape_z,), name='z')
dec_dense_out1 = dec_dense1(z)
dec_dense_out2 = dec_dense2(dec_dense_out1)
dec_reshape_out = dec_reshape(dec_dense_out2)
```



3. The Decoder Network

```
dec_conv1 = Conv2DTranspose(32, 3, padding='same',
                            activation='relu',
                            strides=(2, 2),
                            name='dec_conv1')
dec_conv2 = Conv2D(1, 3, padding='same',
                   activation='relu',
                   name='gen_img')

dec_conv_out1 = dec_conv1(dec_reshape_out)
gen_img = dec_conv2(dec_conv_out1)
```

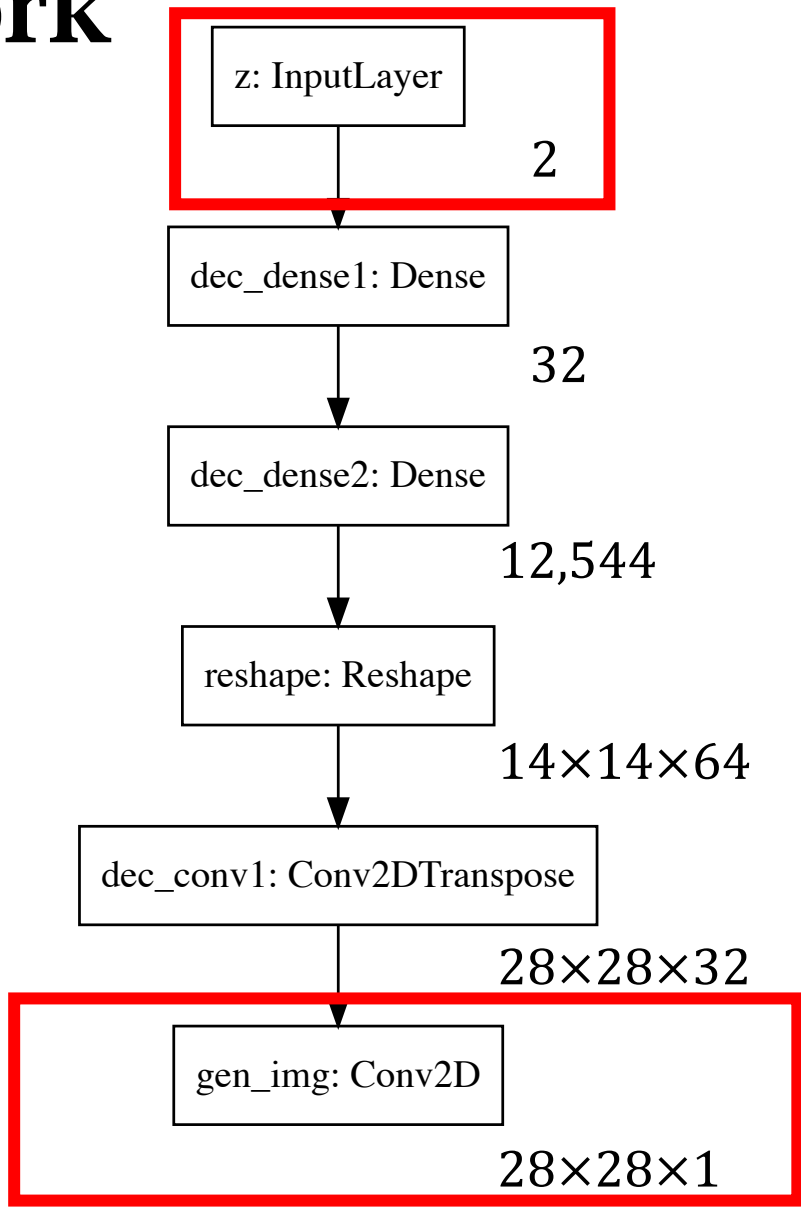
```
decoder = models.Model(inputs=z,
                       outputs=gen_img,
                       name='decoder')
```



3. The Decoder Network

```
dec_conv1 = Conv2DTranspose(32, 3, padding='same',
                            activation='relu',
                            strides=(2, 2),
                            name='dec_conv1')
dec_conv2 = Conv2D(1, 3, padding='same',
                   activation='relu',
                   name='gen_img')

dec_conv_out1 = dec_conv1(dec_reshape_out)
gen_img = dec_conv2(dec_conv_out1)
```

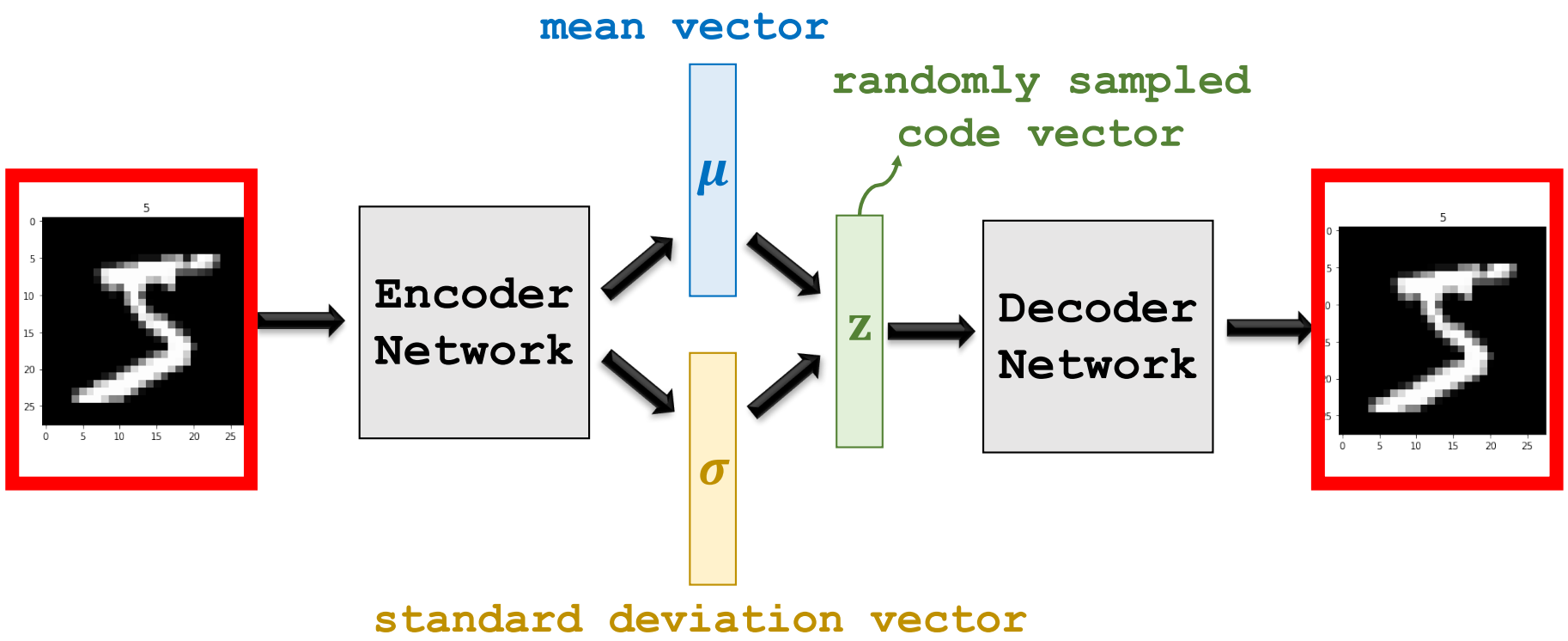
```
decoder = models.Model(inputs=z,
                       outputs=gen_img,
                       name='decoder')
```



Loss Functions

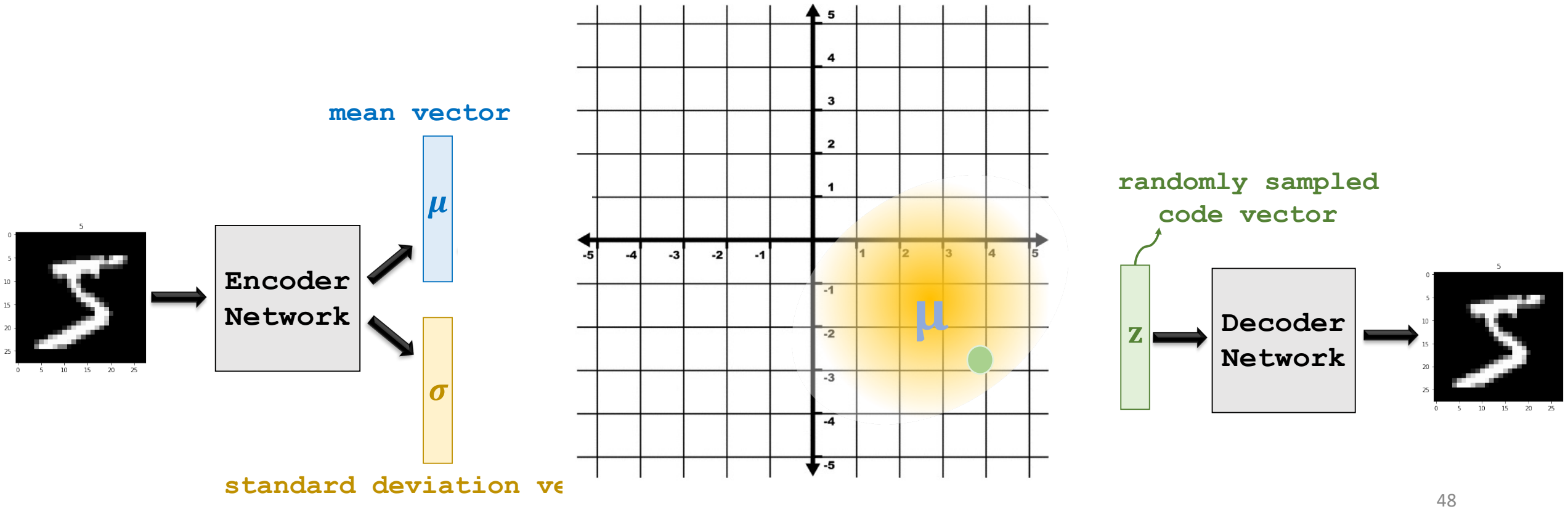
Generation Loss

- `GenLoss = dist(input_img, generated_img)`.
- E.g., the ℓ_2 distance, the cross-entropy, etc.



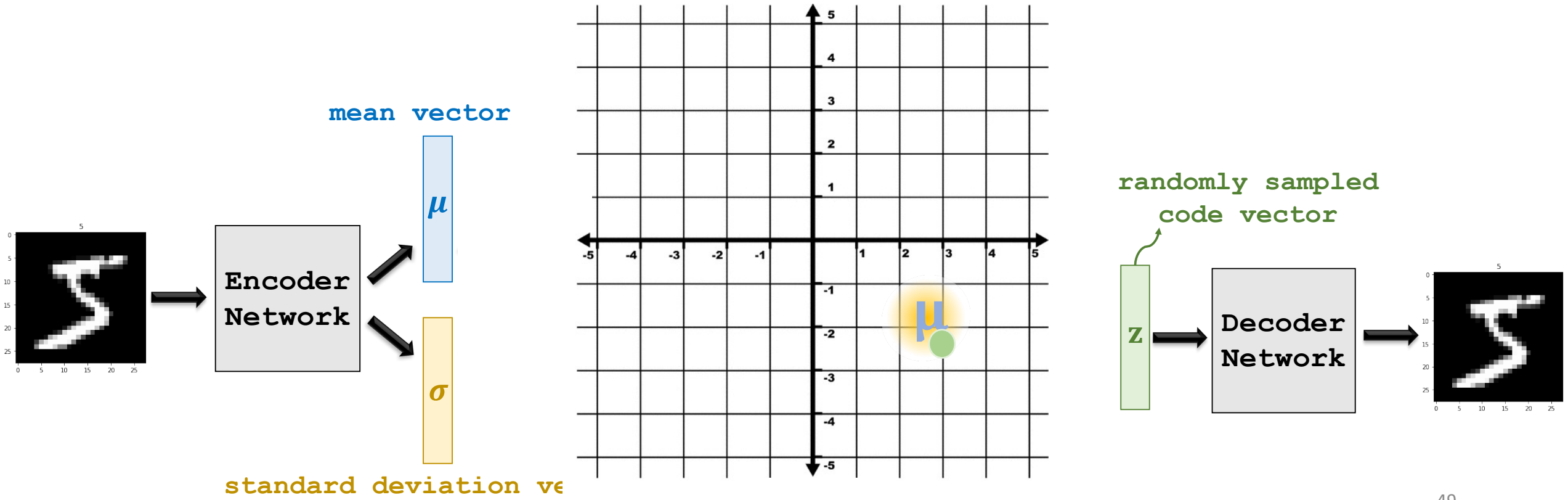
A problem with generation loss

Difficulty: The encoder network will learn an std vector $\sigma \to 0$.



A problem with generation loss

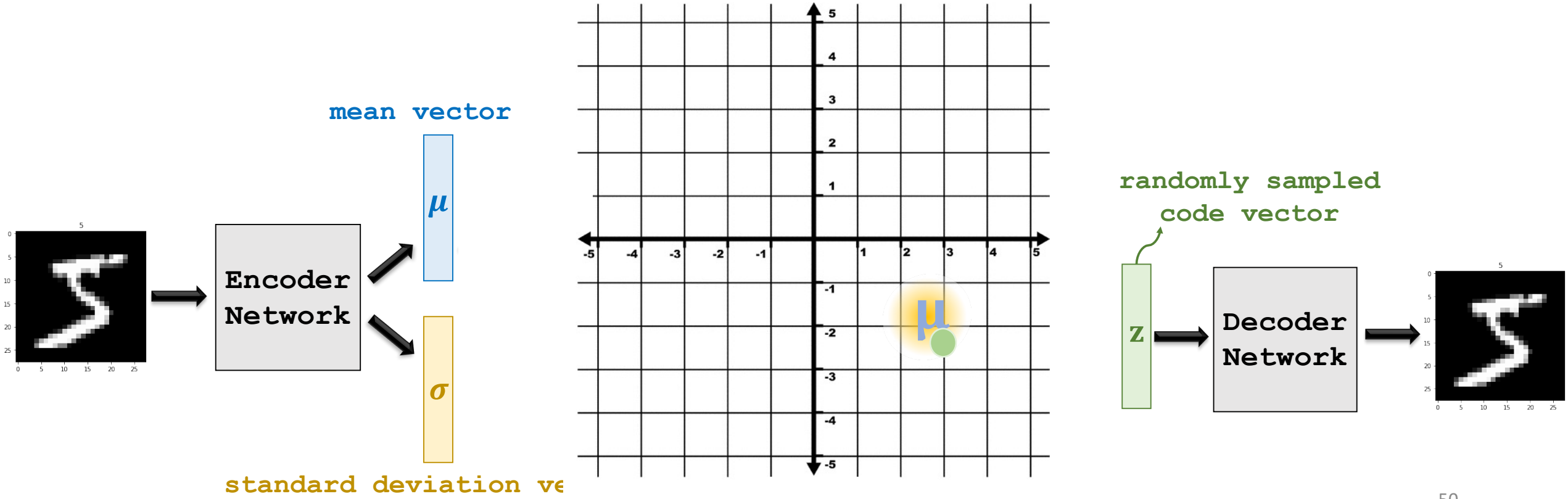
Difficulty: The encoder network will learn an std vector $\sigma \to 0$.



A problem with generation loss

Difficulty: The encoder network will learn an std vector $\sigma \to 0$. (Why?)

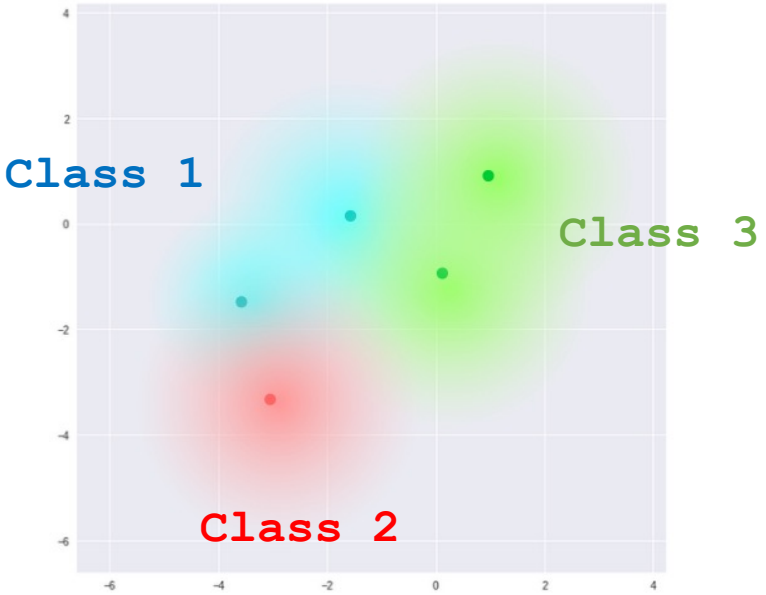
- Minimize `GenLoss` ➔ encourage z close to μ ➔ encourage small σ.
- VAE degrades to the standard AE (VAE with $\sigma = 0$ is exactly AE).



A problem with generation loss

Difficulty: By minimizing generation loss, VAE becomes standard AE.

Dots are code vectors ($\boldsymbol{\mu}$); shadows illustrate std ($\boldsymbol{\sigma}$).



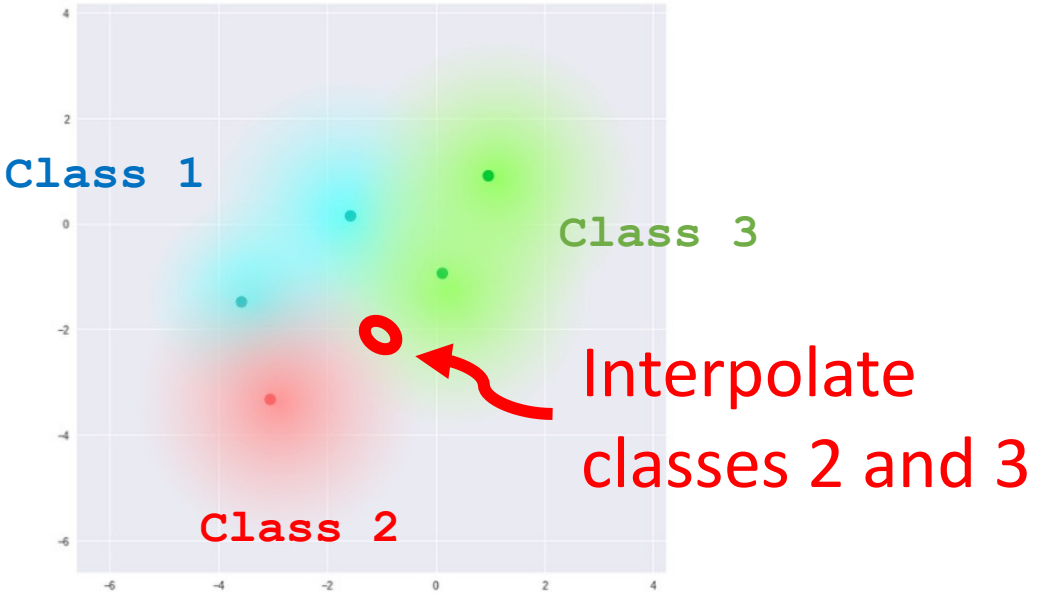
What we hope to learn.

What we actually learn by minimizing `GenLoss`.

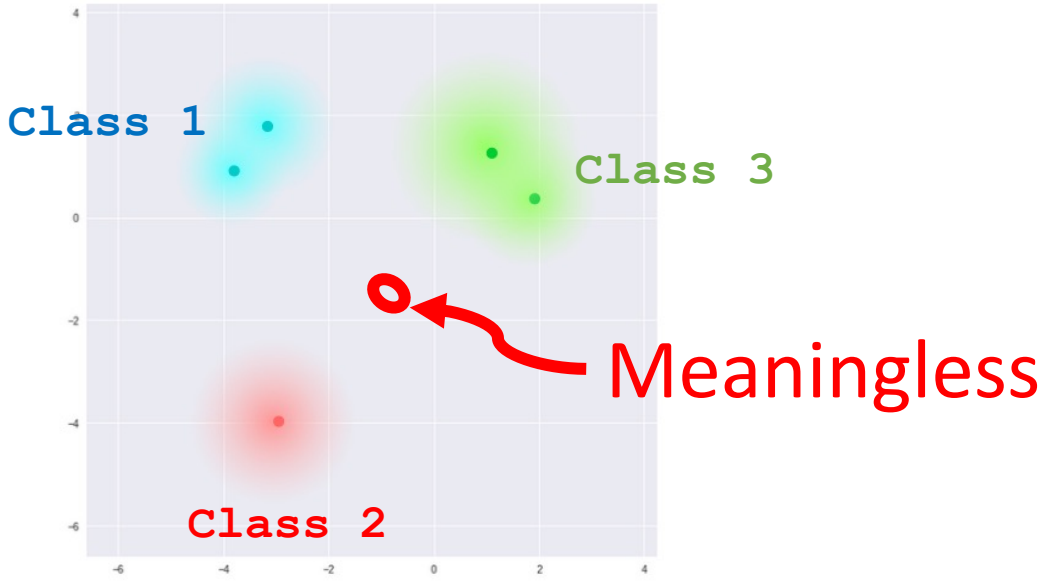
A problem with generation loss

Difficulty: By minimizing generation loss, VAE becomes standard AE.

Dots are code vectors ($\boldsymbol{\mu}$); shadows illustrate std ($\boldsymbol{\sigma}$).



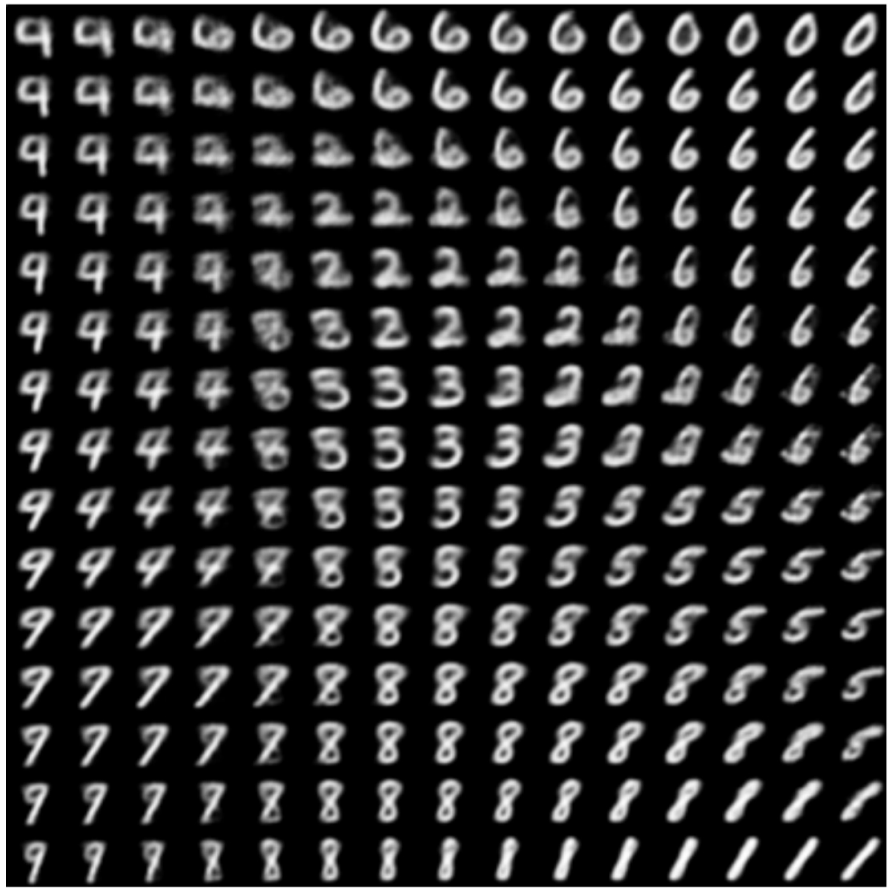
What we hope to learn.



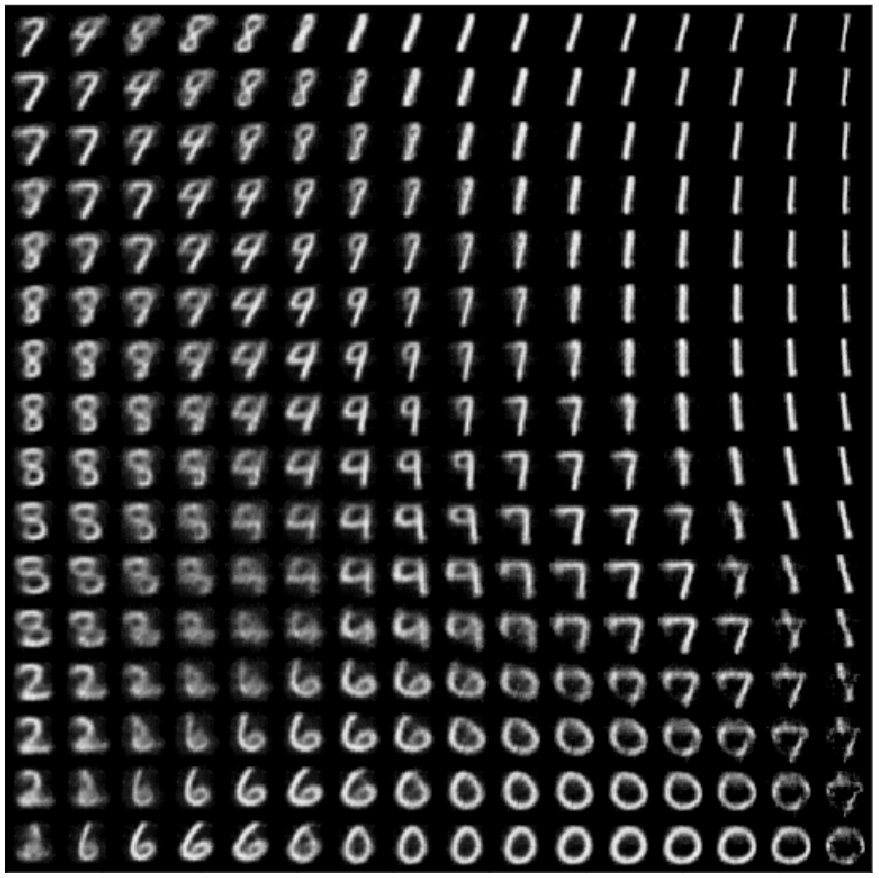
What we actually learn by minimizing `GenLoss`.

A problem with generation loss

Difficulty: By minimizing generation loss, VAE becomes standard AE.



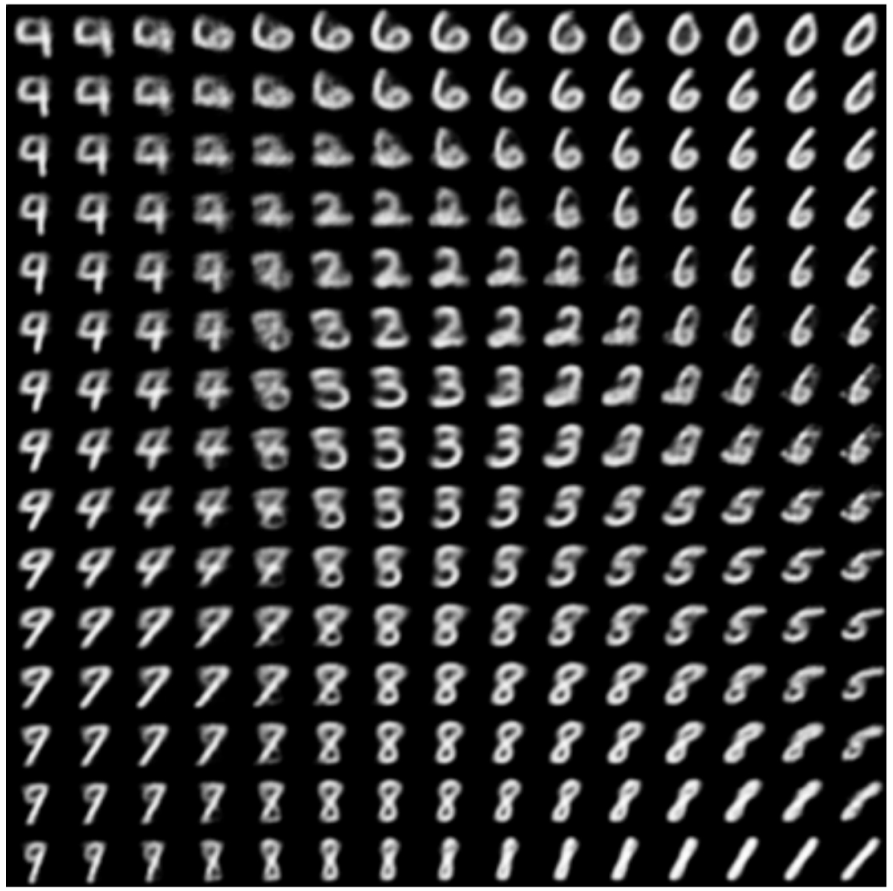
What we hope to learn.



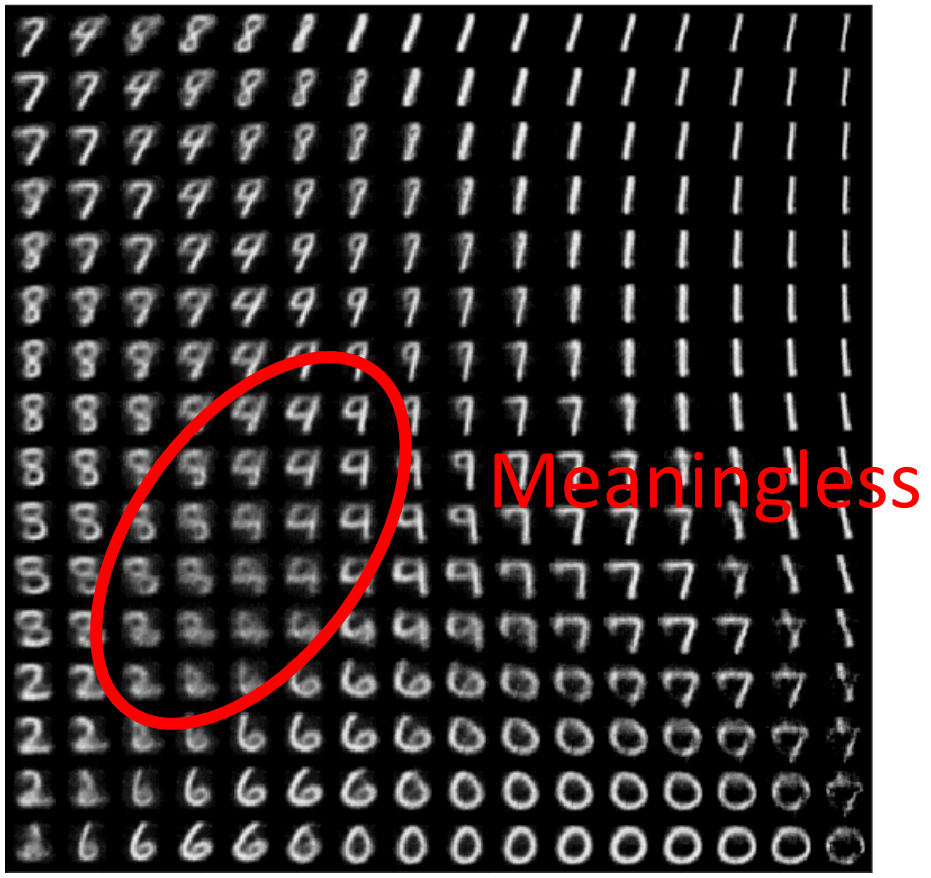
What we actually learn by minimizing `GenLoss`.

A problem with generation loss

Difficulty: By minimizing generation loss, VAE becomes standard AE.



What we hope to learn.

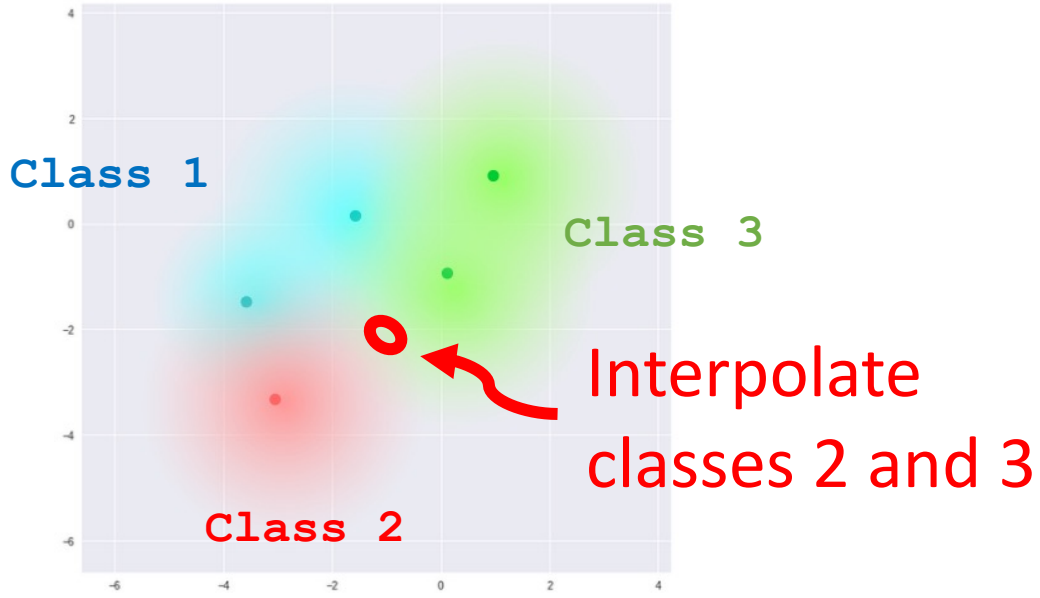


What we actually learn by minimizing `GenLoss`.

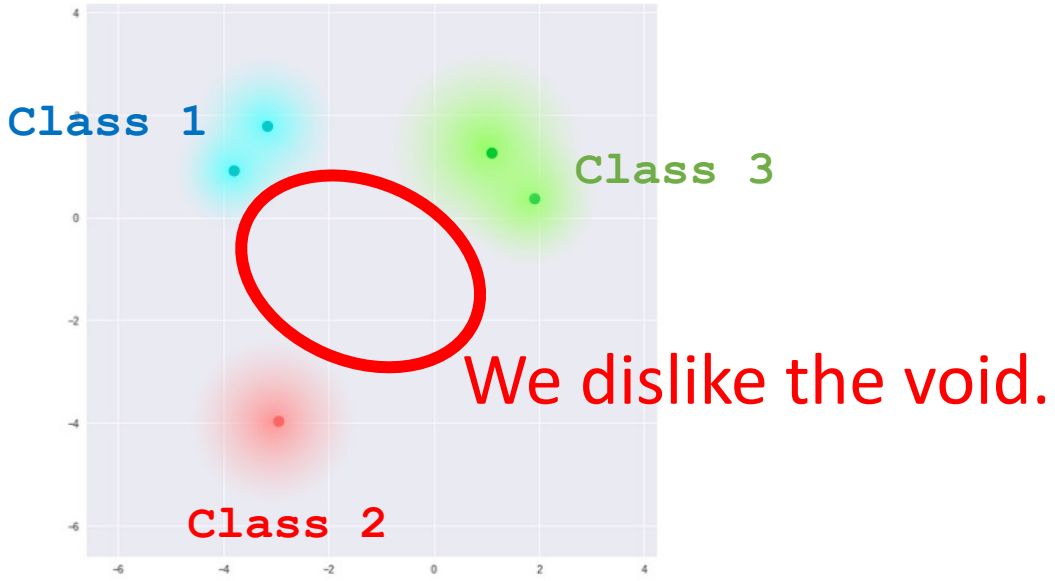
KL Loss

Idea: Encourage the distribution of $\mathbf{z}$ to be unit Gaussian.

- There will be a probability density everywhere around the origin.
- ➔ No more void.



What we hope to learn.

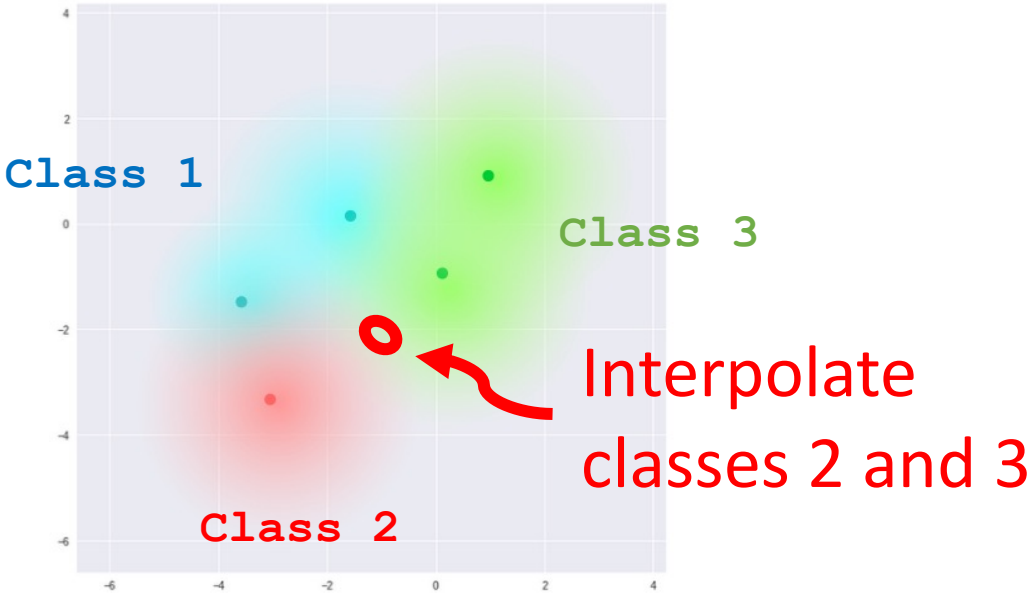


What we actually learn by minimizing `GenLoss`.

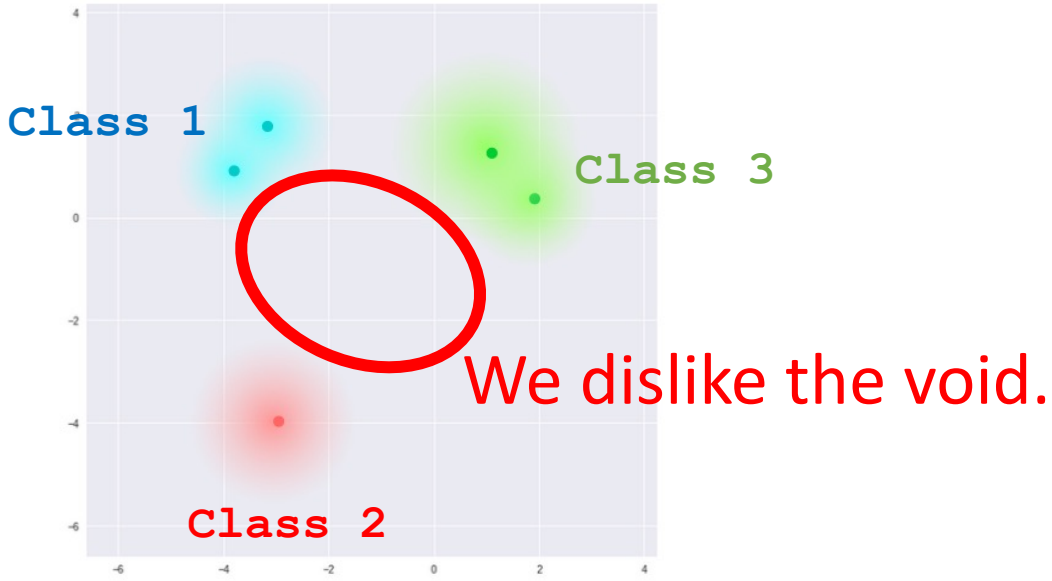
KL Loss

Idea: Encourage the distribution of $\mathbf{z}$ to be unit Gaussian.

- $\text{KLLoss} = D_{\text{KL}}\left(p(\mathbf{z}) \;||\; \mathcal{N}(0, \mathbf{I})\right)$



What we hope to learn.



What we actually learn by minimizing `GenLoss`.

KL Loss

Idea: Encourage the distribution of $\mathbf{z}$ to be unit Gaussian.

- $\text{KLLoss} = D_{\text{KL}}\left(p(\mathbf{z}) \;||\; \mathcal{N}(0, \mathbf{I})\right)$

- $\text{Loss} = \text{GenLoss} + \lambda \cdot \text{KLLoss}$

Generation Loss + KL Loss

- Generation loss:

$$\text{GenLoss} = \text{dist}(\text{input\_img}, \text{output\_img}).$$

```
l = keras.metrics.binary_crossentropy(input_img, output_img)
```

Generation Loss + KL Loss

- Generation loss:

$$\text{GenLoss} = \text{dist}(\text{input\_img}, \text{output\_img}).$$

```
l = keras.metrics.binary_crossentropy(input_img, output_img)
```

- KL loss:

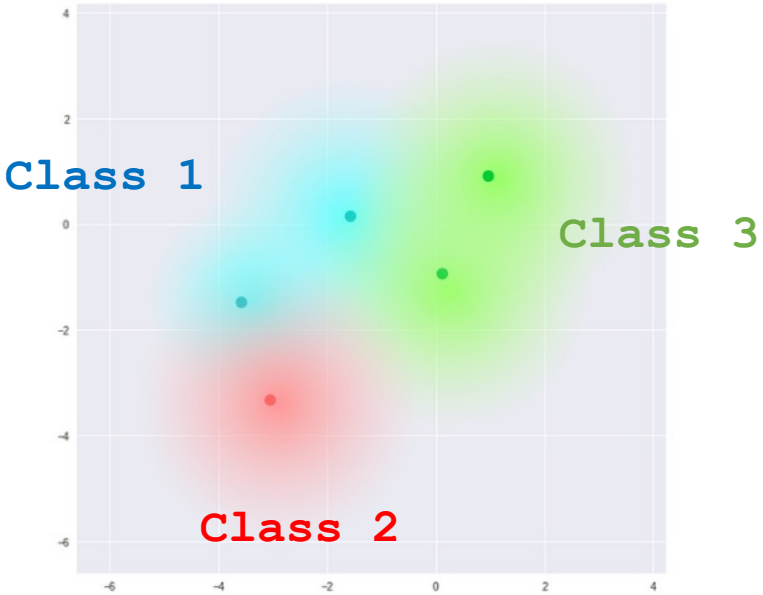
$$\text{KLLoss} = D_{\text{KL}}\big(p(\mathbf{z}) \;||\; \mathcal{N}(0, \mathbf{I})\big)$$

```
l = -0.5 * K.mean(1 + log_var - K.square(mu) - K.exp(log_var), axis=-1)
```

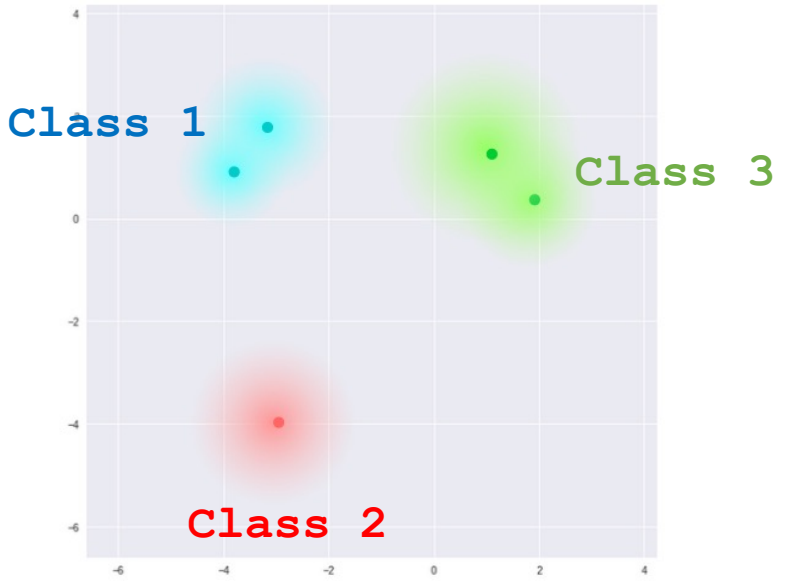
Another interpretation:

- Encourage the variance σ^2 to be big. (Avoid vanishing variance.)
- Pull the mean $\boldsymbol{\mu}$ towards the origin. (Avoid isolated clusters.)

Generation Loss + KL Loss



What we hope to learn.



What we actually learn by minimizing `GenLoss`.

Another interpretation:

- Encourage the variance σ^2 to be big. (Avoid vanishing variance.)
- Pull the mean $\boldsymbol{\mu}$ towards the origin. (Avoid isolated clusters.)

Generation Loss + KL Loss

- Generation loss:

$$\text{GenLoss} = \text{dist}(\text{input\_img}, \text{output\_img}).$$

```
l = keras.metrics.binary_crossentropy(input_img, output_img)
```

- KL loss:

$$\text{KLLoss} = D_{\text{KL}}\left(p(\mathbf{z}) \;||\; \mathcal{N}(0, \mathbf{I})\right)$$

```
l = -0.5 * K.mean(1 + log_var - K.square(mu) - K.exp(log_var), axis=-1)
```

- $\text{Loss} = \text{GenLoss} + 10^{-3} \cdot \text{KLLoss}$

Take `input_img` and `output_img` as inputs

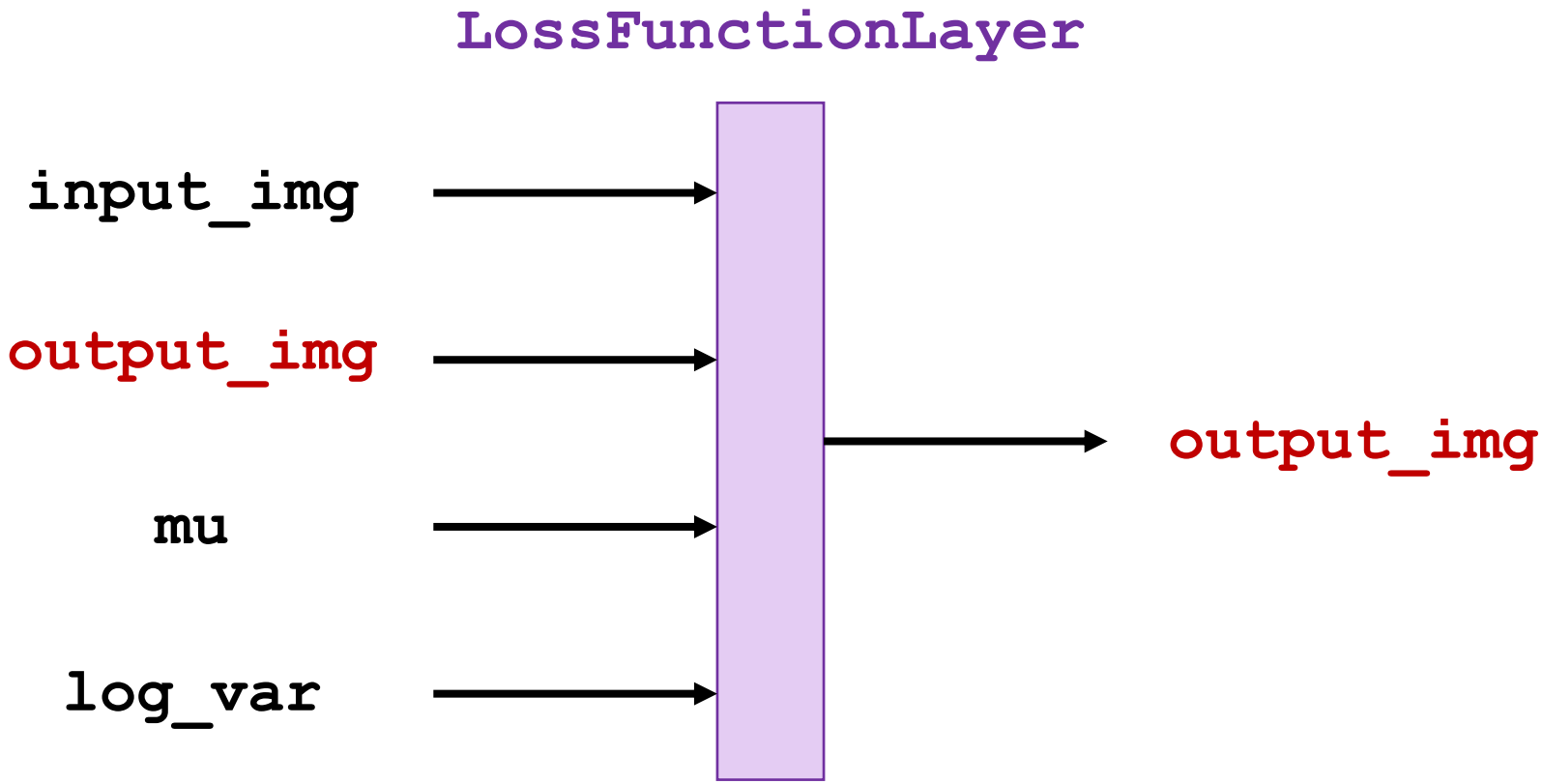
tuning hyper-parameter

Take `log_var` and `mu` as inputs

Implement loss function

```
import keras
from keras import backend as K

class LossFunctionLayer(keras.layers.Layer):
    param = 1E-3

    def kl_loss(self, mu, log_var):
        l = -0.5 * K.mean(1 + log_var - K.square(mu) - K.exp(log_var), axis=-1)
        return self.param * K.mean(l)

    def gen_loss(self, input_img, output_img):
        l = keras.metrics.binary_crossentropy(input_img, output_img)
        return K.mean(l)

    def call(self, inputs):
        input_img, output_img, mu, log_var = inputs
        loss1 = self.gen_loss(input_img, output_img)
        loss2 = self.kl_loss(mu, log_var)
        self.add_loss(loss1+loss2, inputs=inputs)
        return output_img
```

Implement loss function



Also compute the loss function.

- **input_img** and **output_img** for **GenerationLoss**.
- **mu** and **log_var** for **KLtLoss**.

Training

Connect the modules

We have defined 4 modules:

- **Encoder:** `[input_img]` → `[mu, log_var]`
- **Sampling:** `[mu, log_var, eps]` → `[z]`
- **Decoder:** `[z]` → `[output_img]`
- **LossFunctionLayer:** `[input_img, output_img, mu, log_var]` → `[output_img]`

```
input_img = Input(shape=(28,28,1), name='input_img')
n = K.shape(input_img)[0]
eps = Input(tensor=K.random_normal(shape=(n, shape_z)), name='eps')
```

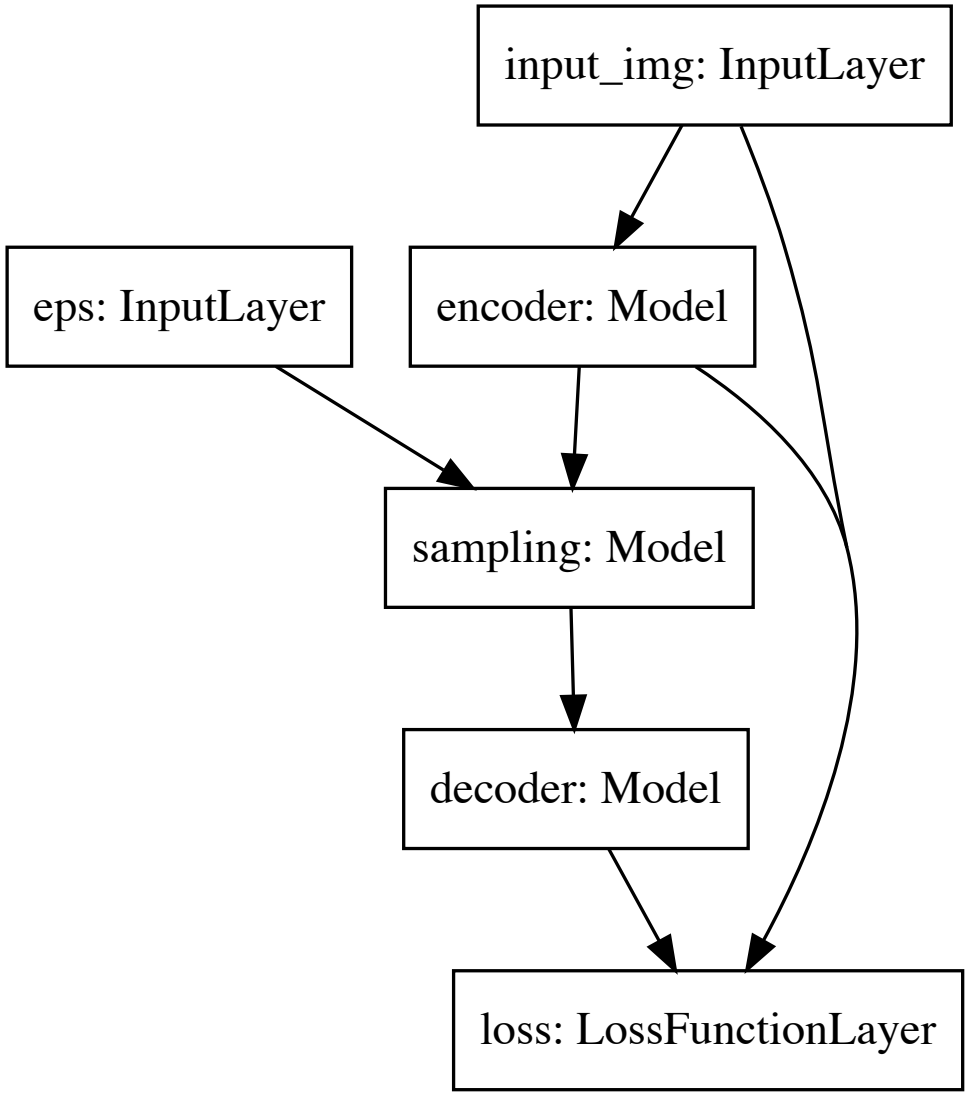
Connect the modules

We have defined 4 modules:

- **Encoder:** `[input_img]` → `[mu, log_var]`
- **Sampling:** `[mu, log_var, eps]` → `[z]`
- **Decoder:** `[z]` → `[output_img]`
- **LossFunctionLayer:** `[input_img, output_img, mu, log_var]` → `[output_img]`

```
input_img = Input(shape=(28,28,1), name='input_img')
n = K.shape(input_img)[0]
eps = Input(tensor=K.random_normal(shape=(n, shape_z)), name='eps')

mu, log_var = encoder(input_img)
z = sampling([mu, log_var, eps])
output_img = decoder(z)
output_img = LossFunctionLayer(name='loss')([input_img, output_img, mu, log_var])
model = models.Model(inputs=[input_img, eps], outputs=output_img)
```

Connect the modules



Train the model on MNIST dataset

```
history = model.fit(
    x_train,
    None,
    shuffle=True,
    epochs=50,
    batch_size=128,
    validation_data=(x_test, None)
)
```

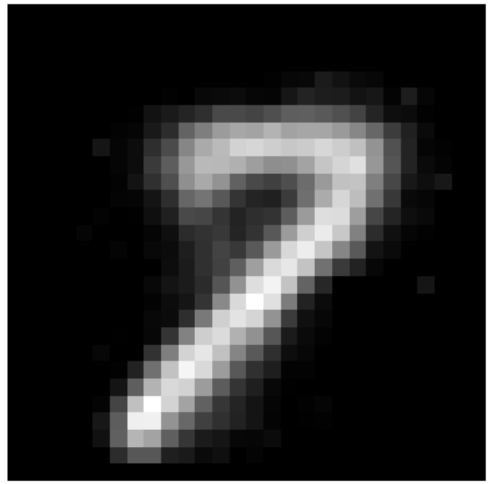
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/50
60000/60000 [==============================] - 231s 4ms/step - loss: 0.6717 - val_loss: 0.2194
Epoch 2/50
60000/60000 [==============================] - 251s 4ms/step - loss: 0.2139 - val_loss: 0.2083
Epoch 3/50
60000/60000 [==============================] - 299s 5ms/step - loss: 0.2066 - val_loss: 0.2096
    ⋮
Epoch 49/50
60000/60000 [==============================] - 240s 4ms/step - loss: 0.1839 - val_loss: 0.1859
Epoch 50/50
60000/60000 [==============================] - 233s 4ms/step - loss: 0.1837 - val_loss: 0.1858
```

Visualize code vectors

arbitrary 2-dim vector

map the 2-dim vector to a 28×28 image

```
z_sample = np.array([0.1, 0.2]).reshape((1, 2))
x_decoded = decoder.predict(z_sample)[0].reshape((28, 28))

fig = plt.figure(figsize=(6, 6))
plt.imshow(x_decoded, cmap='gray')
plt.axis('off')
plt.show()
```

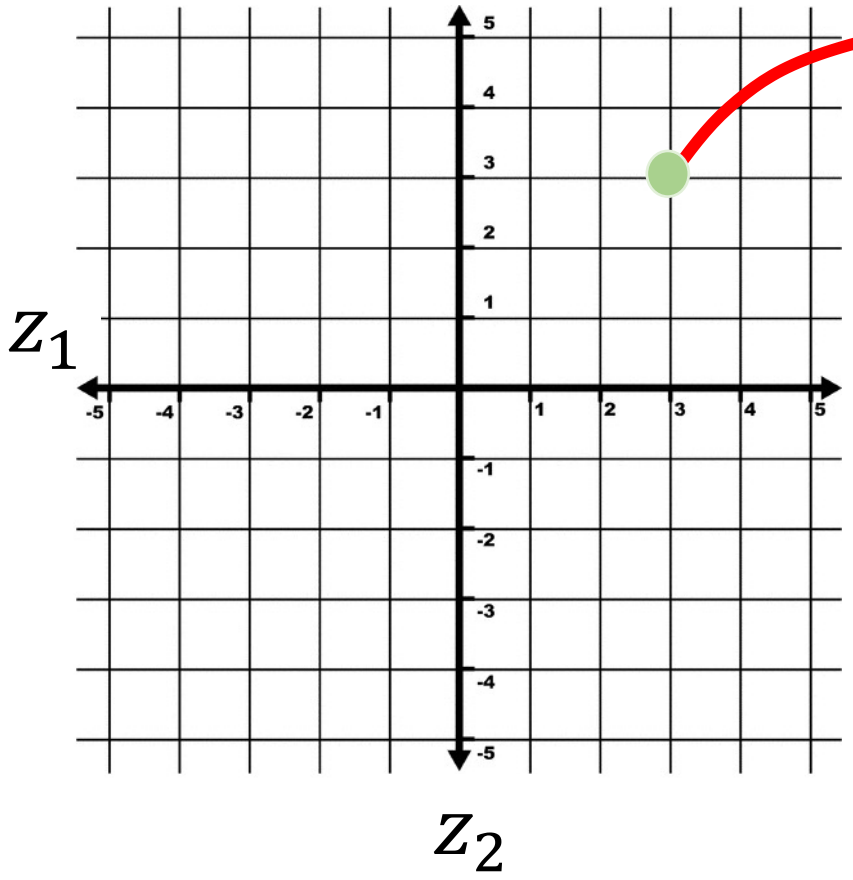


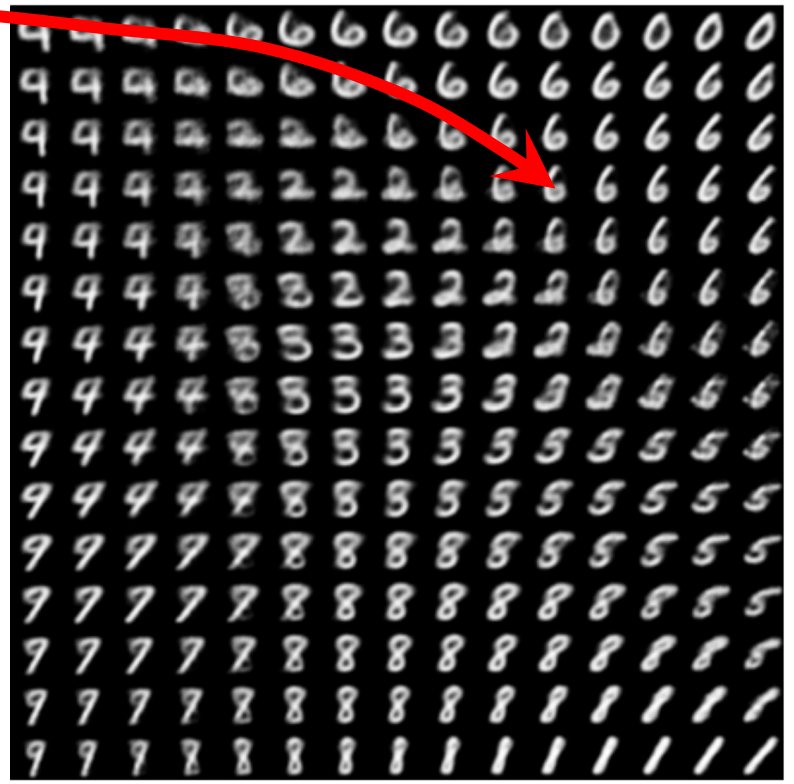
Visualize code vectors

Code vectors

Images

Decoder





Summary

Variational Autoencoder (VAE)

- VAE = AE + probability tricks.
- The decoder network behaves like a continuous function.

Variational Autoencoder (VAE)

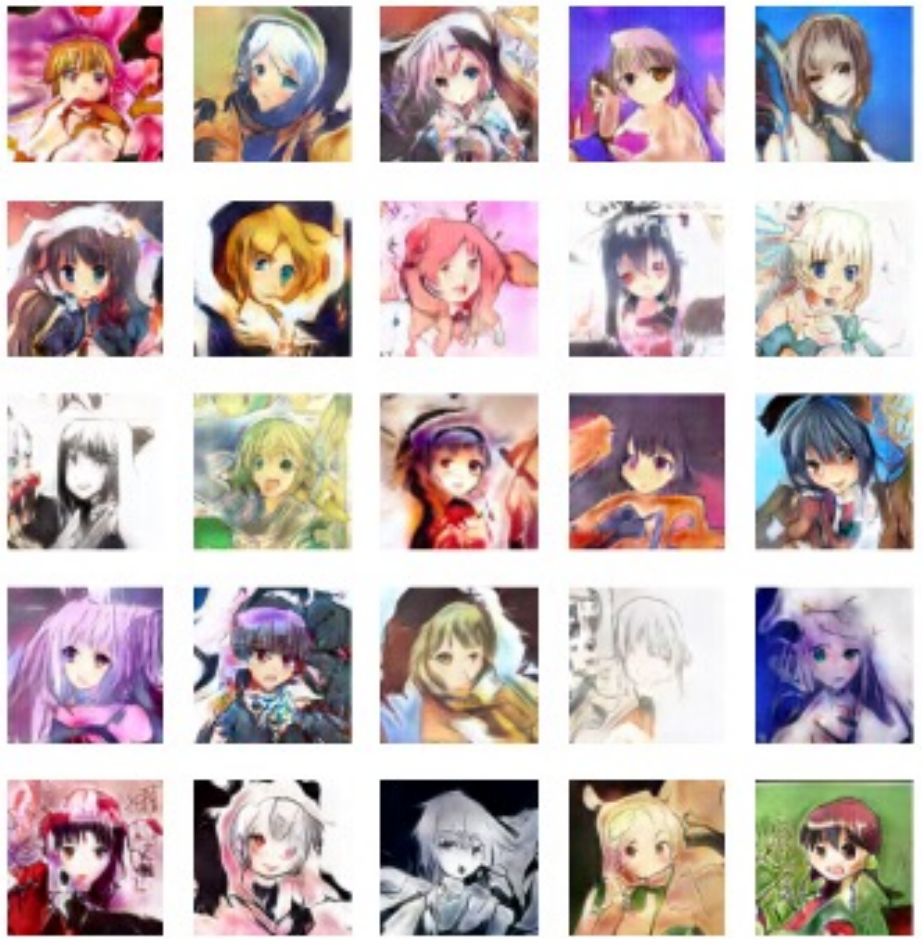
- VAE = AE + probability tricks.
- The decoder network behaves like a continuous function.
- Loss = Generation Loss + $\lambda \cdot$ KL Loss.
- Application: edit images via editing code vectors.
  - Average faces.
  - Add smile.

Generative Adversarial Networks (GANs)

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.

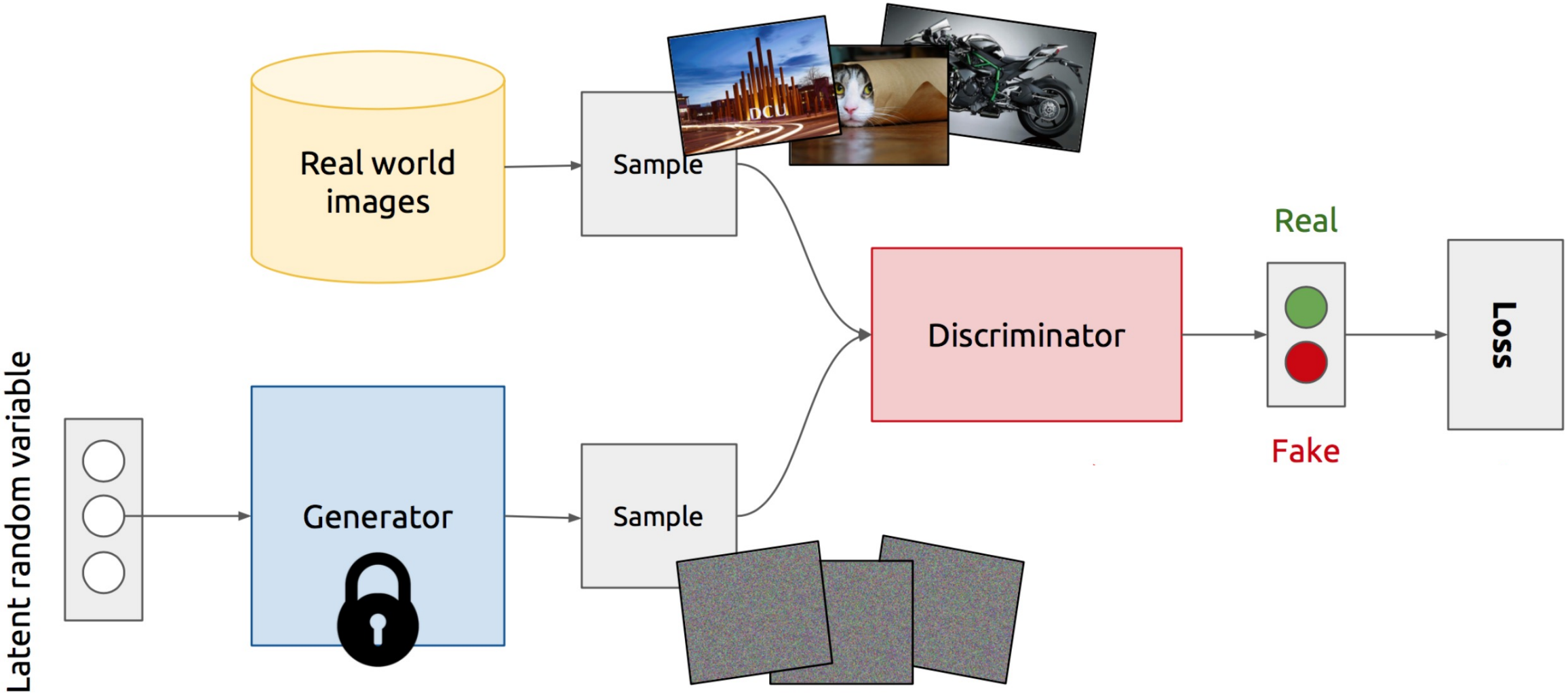
Which is real?

GAN: Main Idea

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

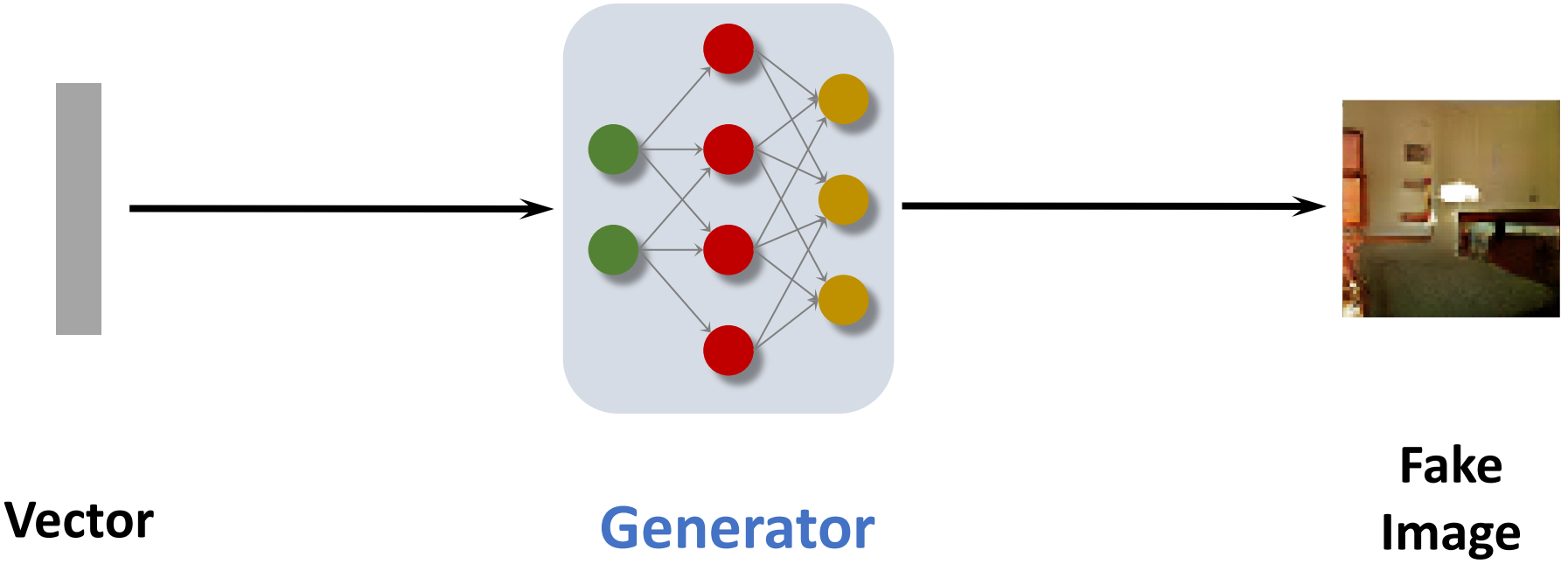
- Train them against each other.
  - Forger mixes his fake with authentic paintings and ask art dealer to assess them.
  - Art dealer gives feedback about what makes a Picasso look like a Picasso.
  - Forger improves his competence at imitating the style of Picasso.
  - Art dealer improves his competence 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

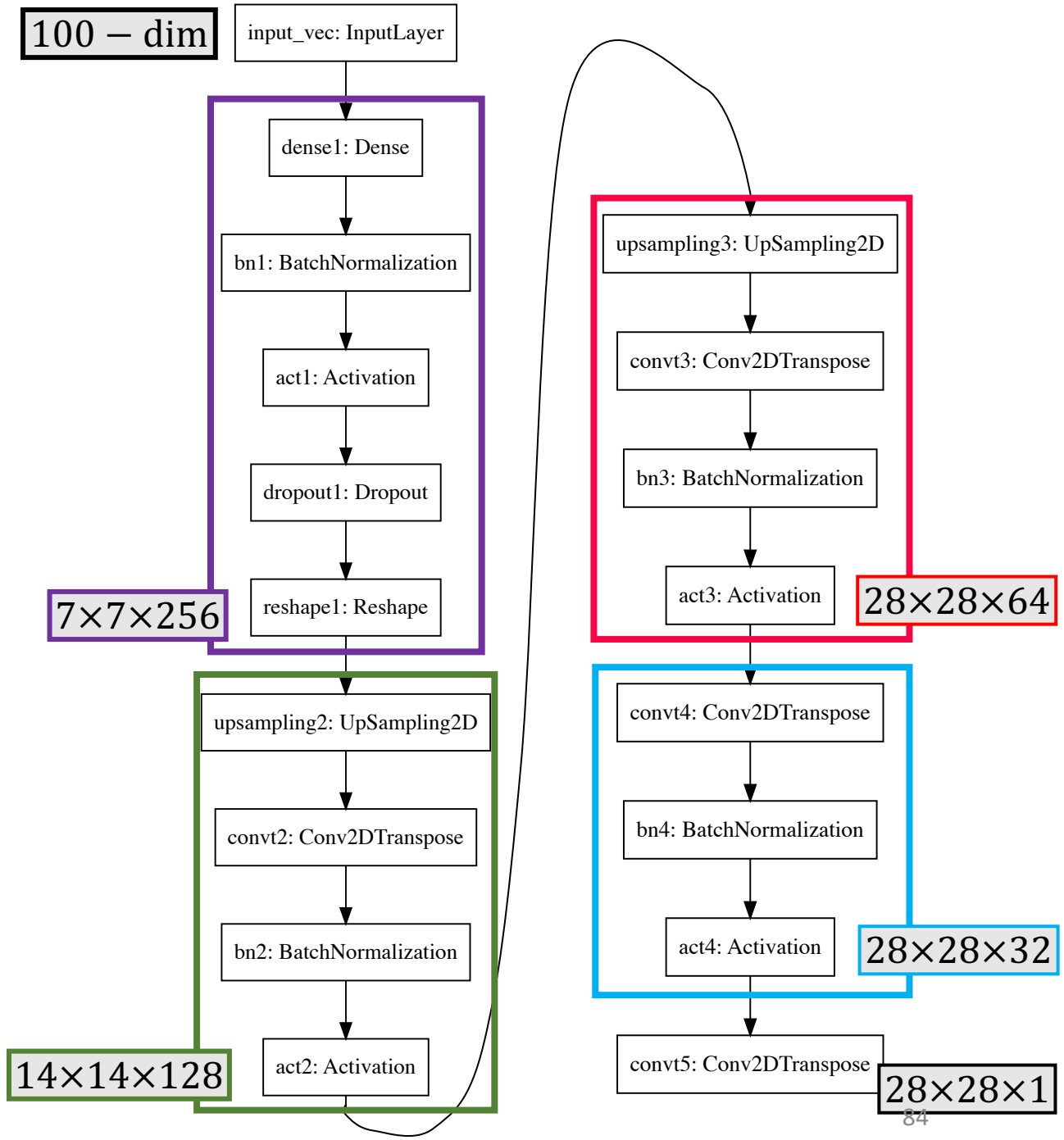
Block 4:
From 28×28×64 to 28×28×32

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 $28\times28\times1$ to $14\times14\times64$

Block 2:
from $14\times14\times64$ to $7\times7\times128$

Block 3:
From $7\times7\times128$ to $4\times4\times256$

Block 4:
From $4\times4\times256$ to $4\times4\times512$

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

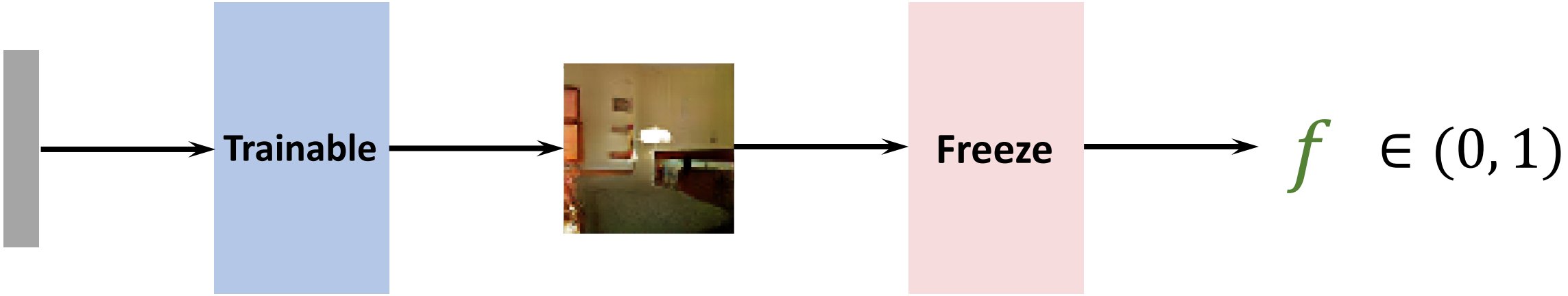
Update the Discriminator

```
# 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

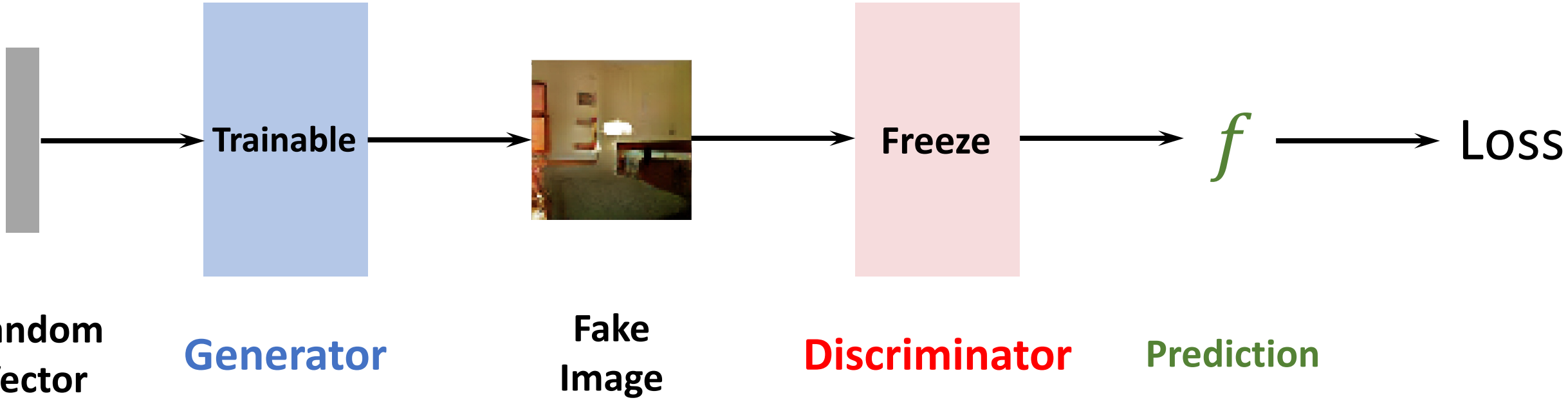
Connect the generator and discriminator (freeze discriminator's parameters).



Update the Generator

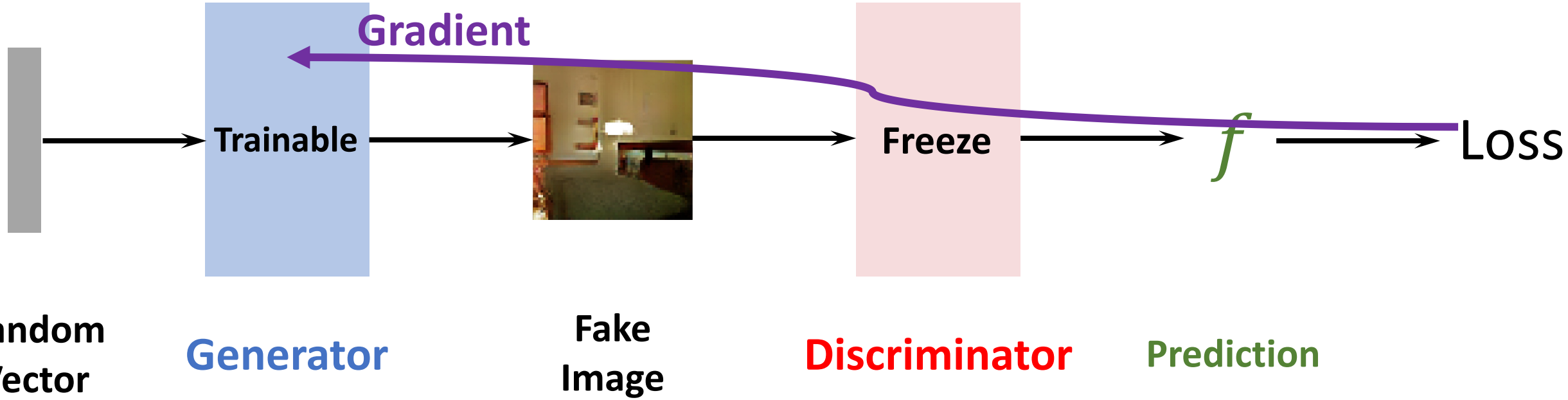
Connect the generator and discriminator (freeze discriminator's parameters).

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



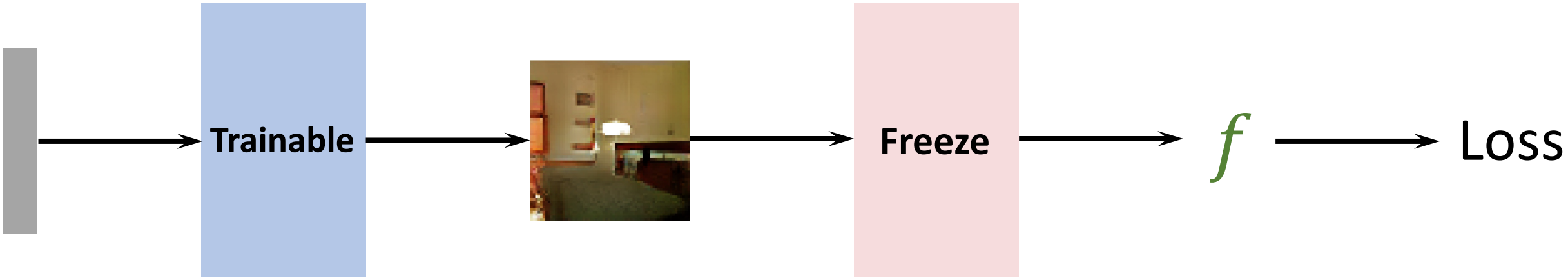
Update the Generator

- $\mathbf{W_G}$: 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

Question: What will happen if we do the followings? (We don't actually do.)

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

Random Vector

Generator (updated)

Fake Image (new)

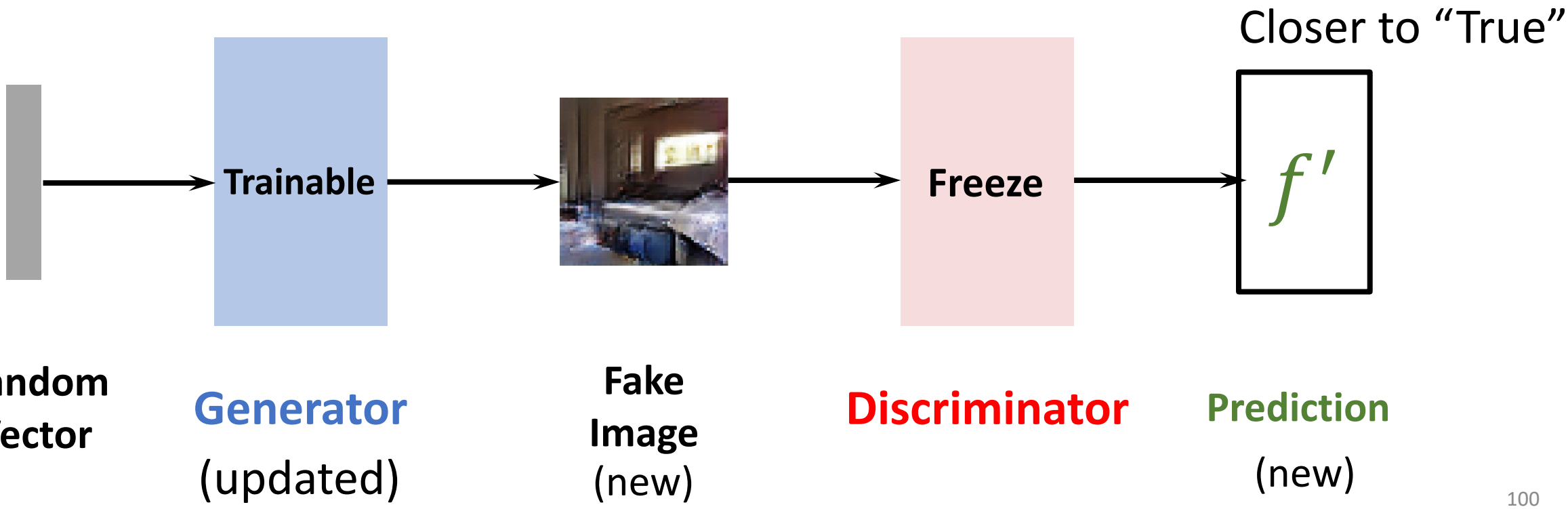
Discriminator

Prediction (new)

Update the Generator

Question: What will happen if we do the followings? (We don't actually do.)

- Use the same **vector**.
- Generate a 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')
fake_img = generator(gan_input)
f = discriminator(fake_img)
gan = Model(inputs=gan_input, outputs=f, 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.

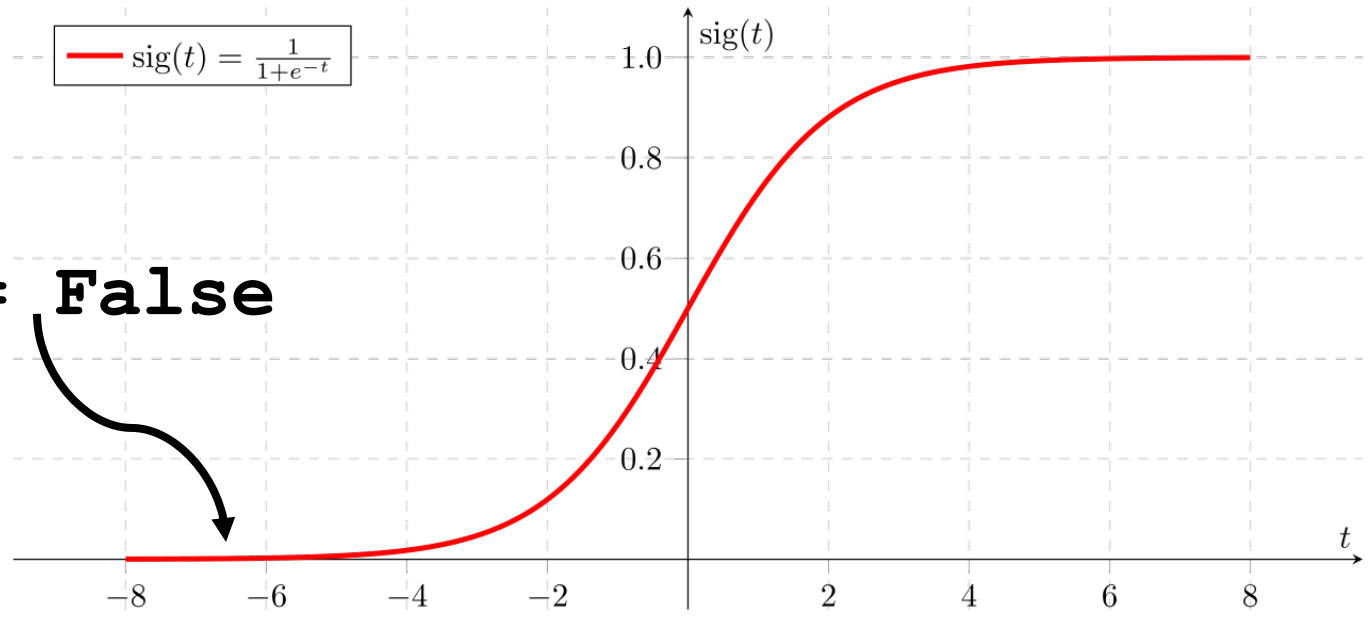
What if the art dealer is 100% correct at judging Picasso painting?

- 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 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 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 100%,
  - ➔ vanishing gradient,
  - ➔ the generator is dead.

- If the discriminator improves too slowly,
  - ➔ 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

Thank you!