

Variational Autoencoder for Image Generation

Shusen Wang

Preliminary: Distance between Two Probability Distributions

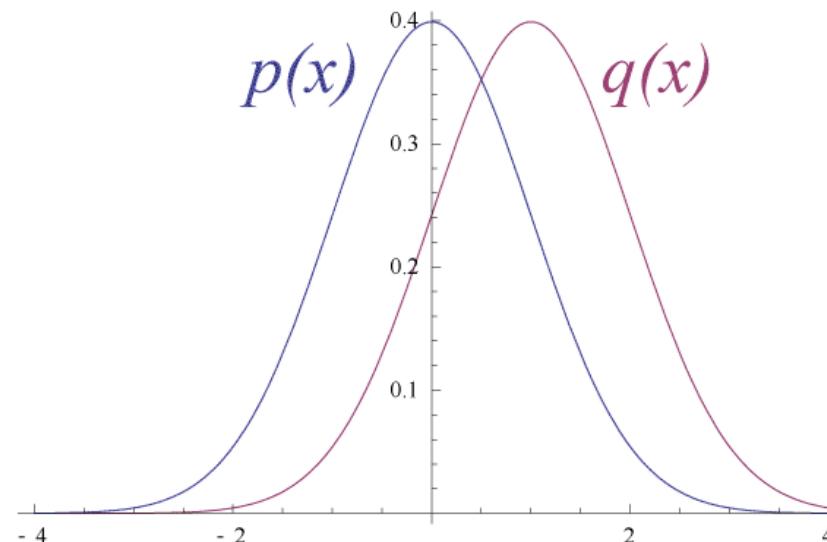
Distance between Two Vectors

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

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 $D_{\text{KL}}(p||q)$.

- For discrete probability distributions, the KL divergence is

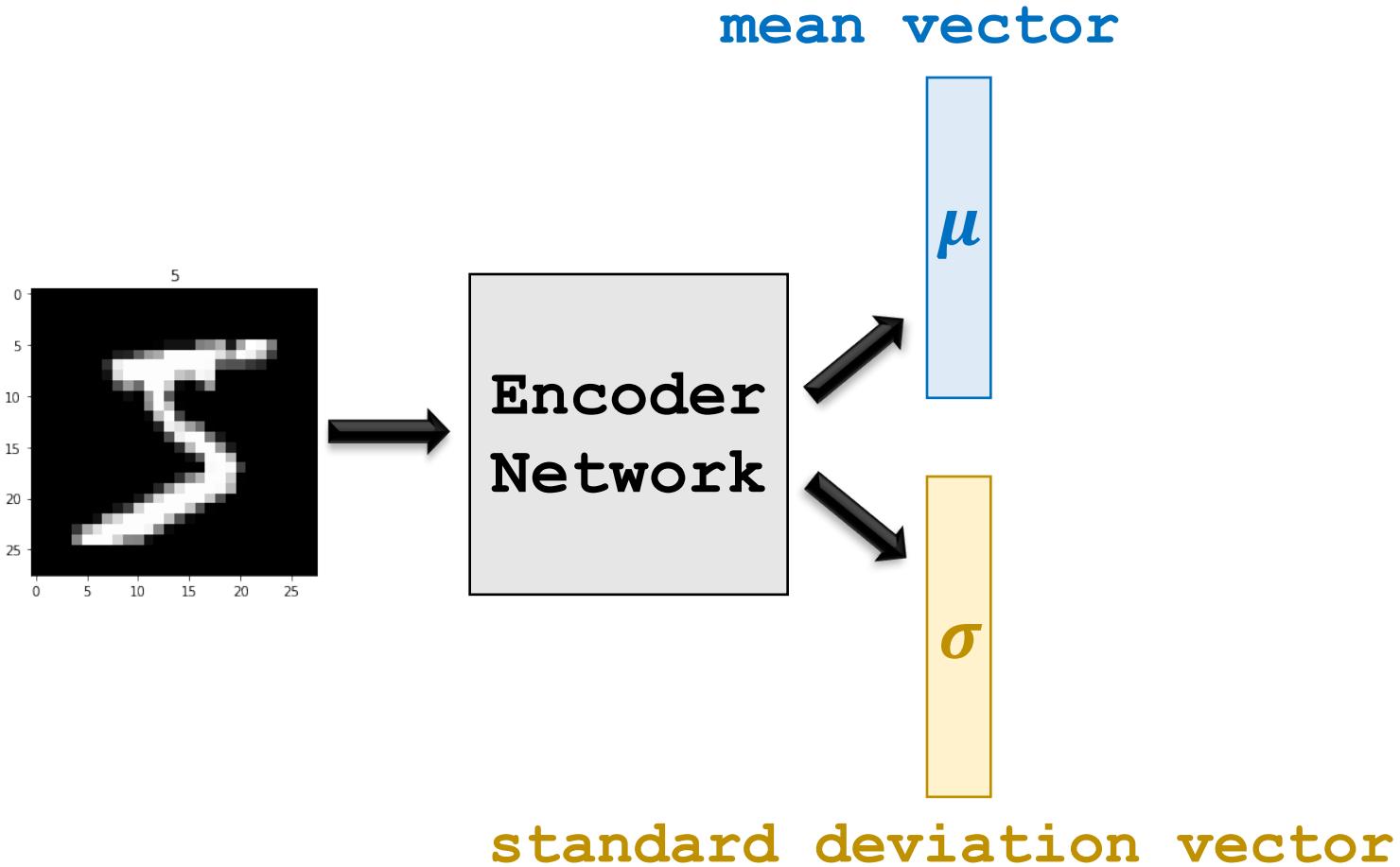
$$D_{\text{KL}}(p||q) = -\sum_i p(i) \log \frac{q(i)}{p(i)}.$$

- For continuous probability distributions, the KL divergence is

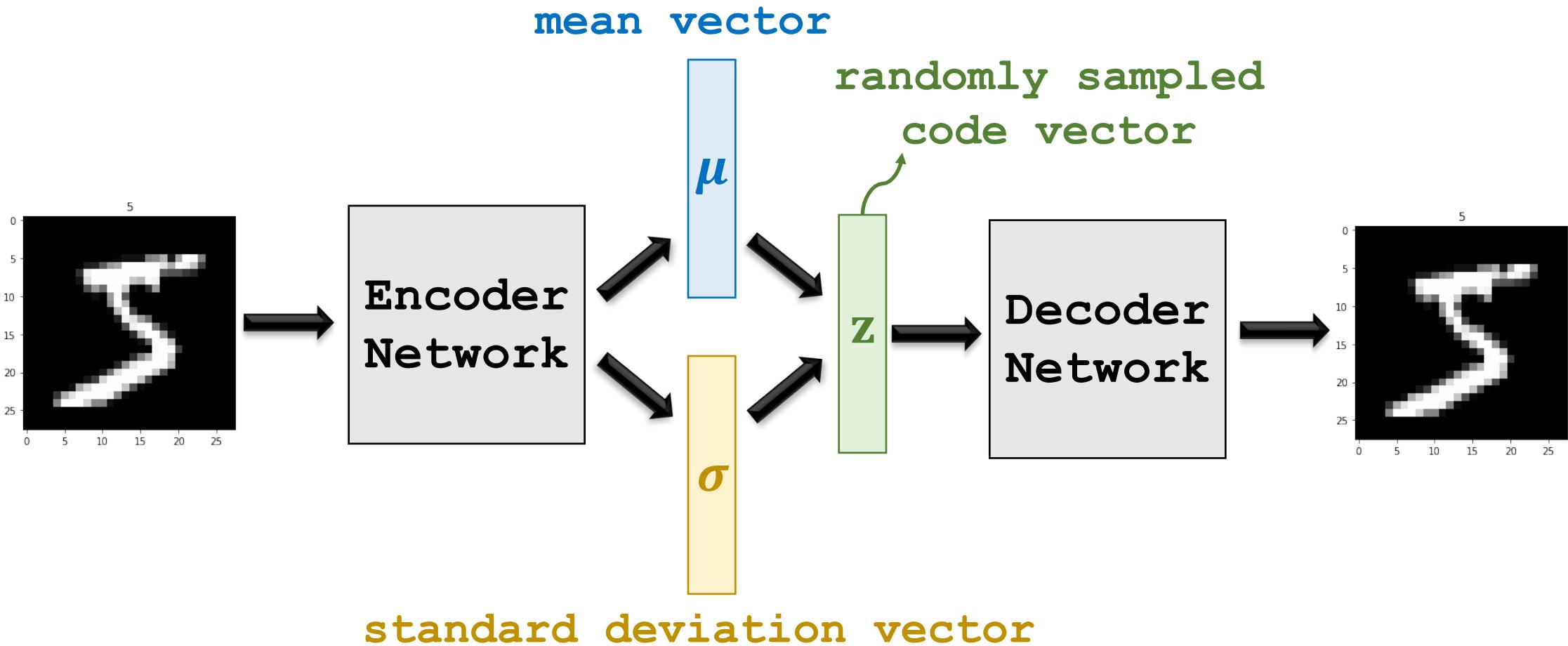
$$D_{\text{KL}}(p||q) = -\int_{-\infty}^{+\infty} p(x) \log \frac{q(x)}{p(x)} dx.$$

Variational Autoencoder

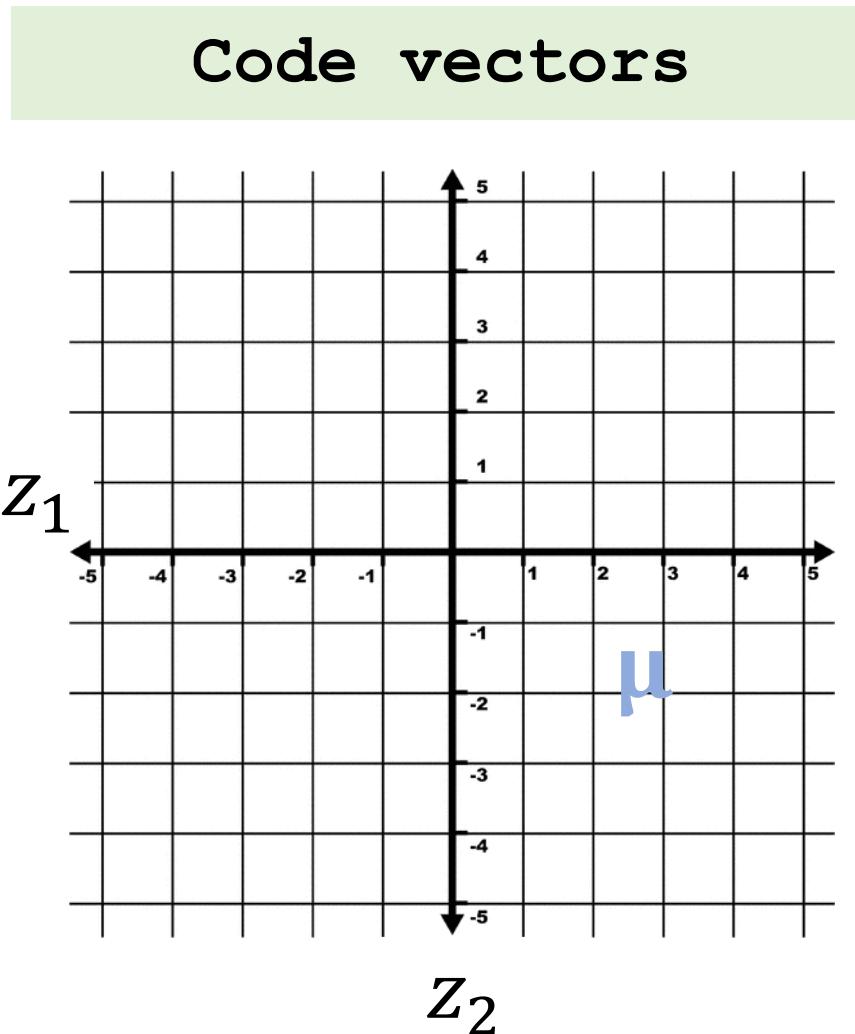
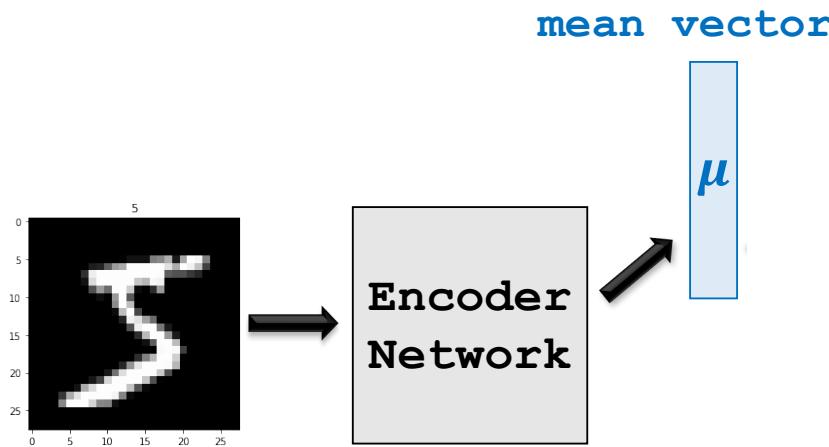
Variational Autoencoder (VAE)



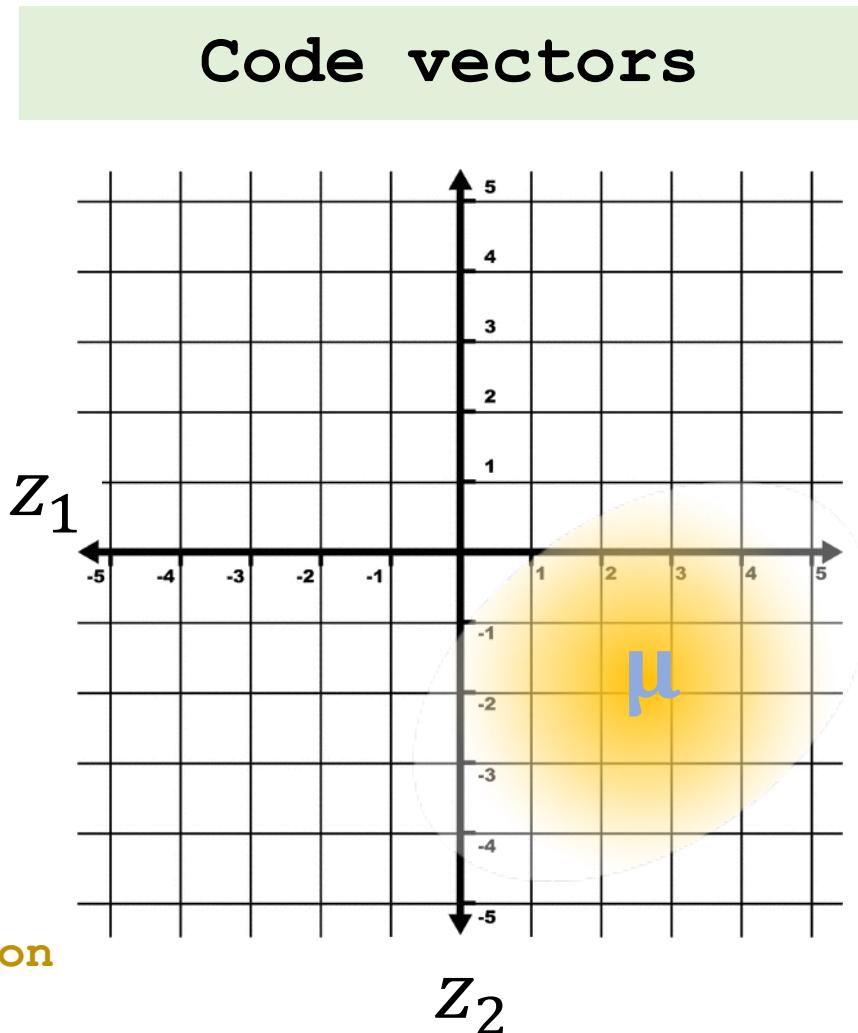
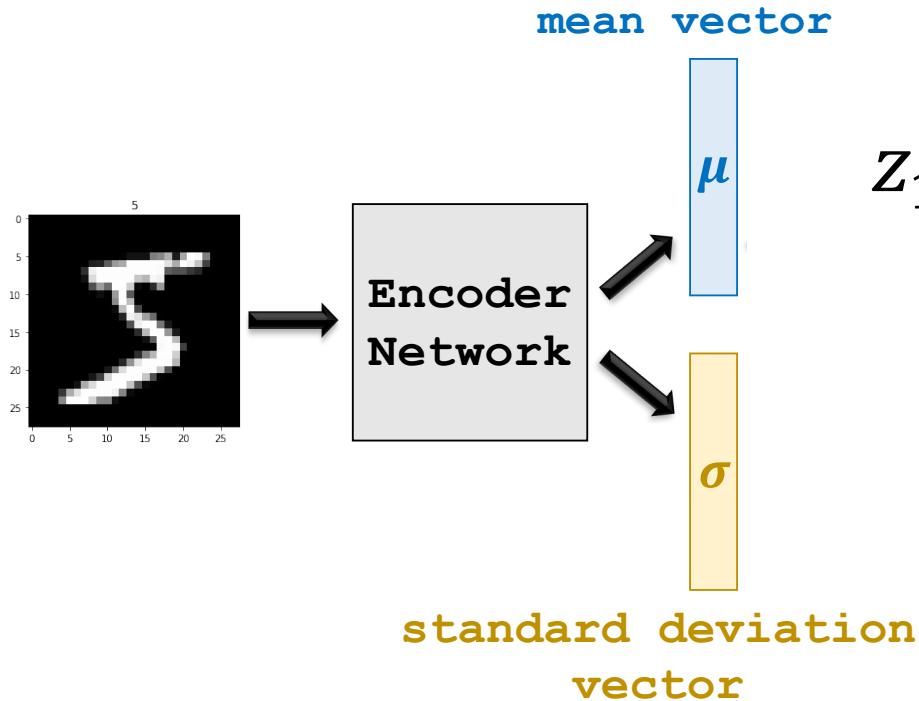
Variational Autoencoder (VAE)



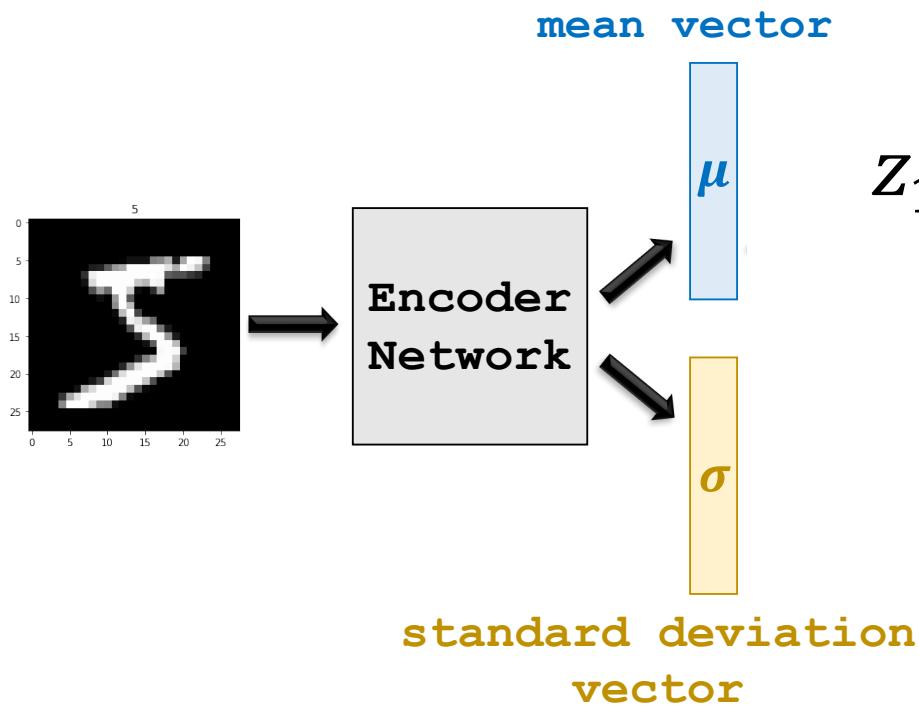
VAE Sampling



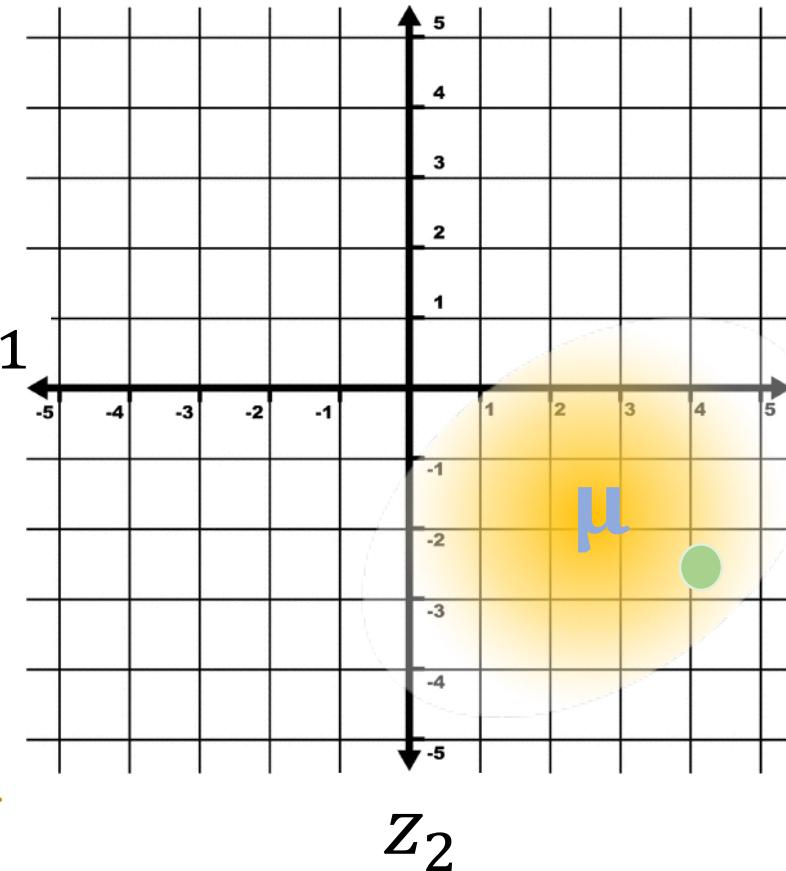
VAE Sampling



VAE Sampling



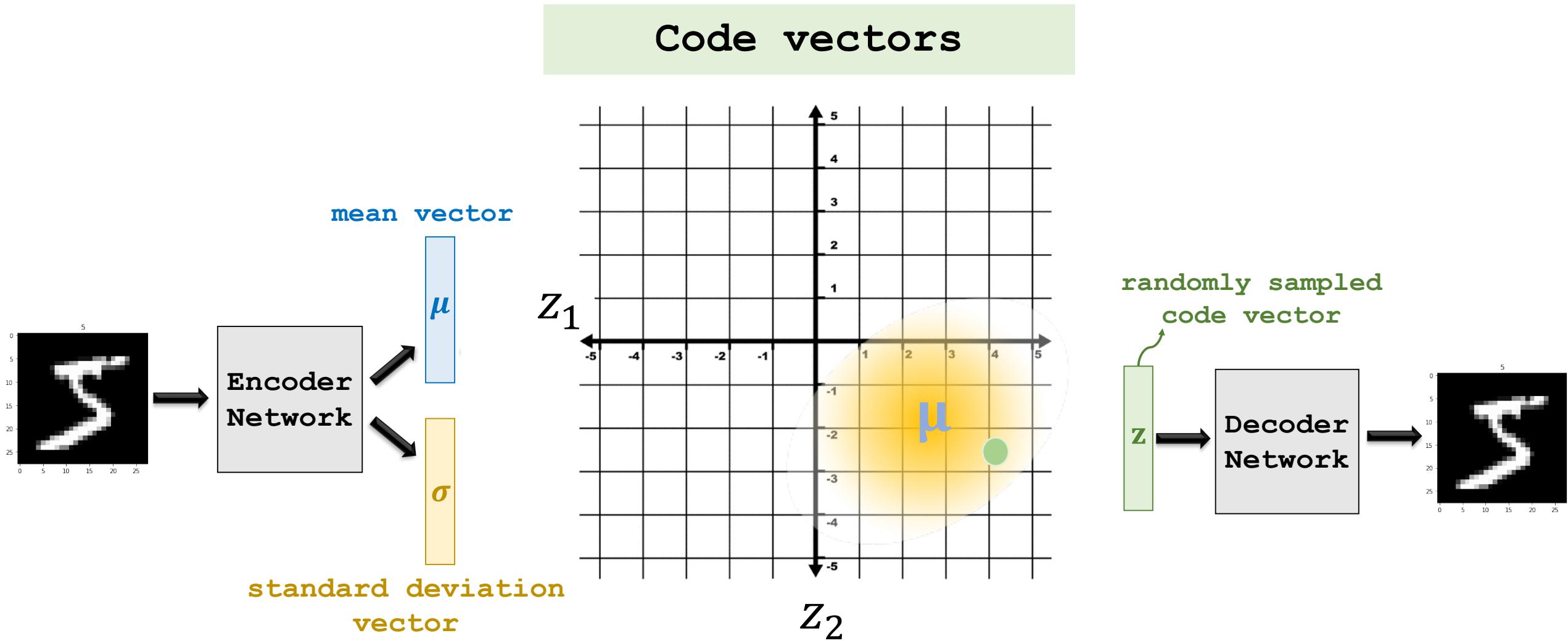
Code vectors



randomly sampled
code vector



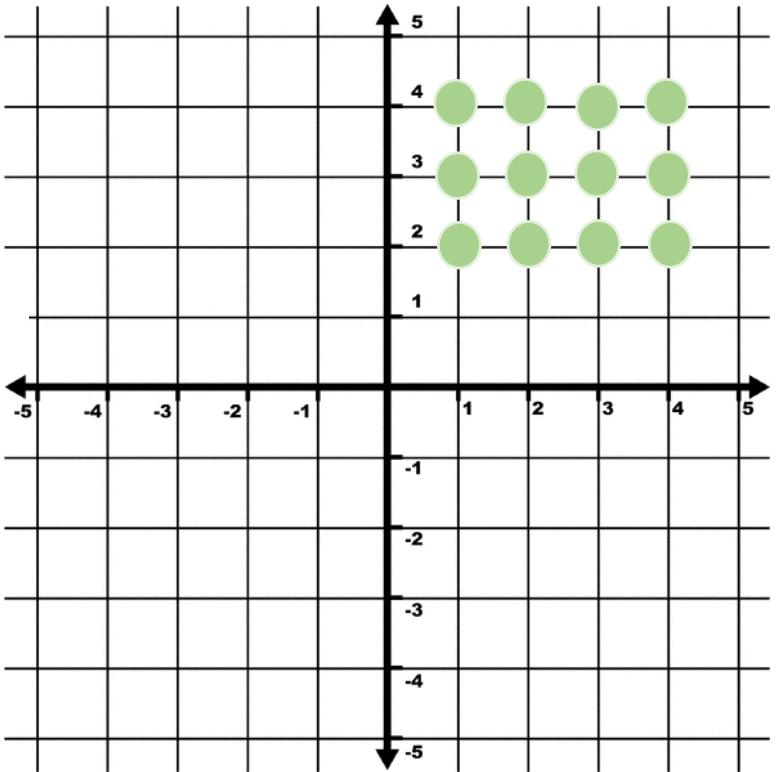
VAE Sampling



Visualize Code Vectors

Visualize code vectors.

Code vectors

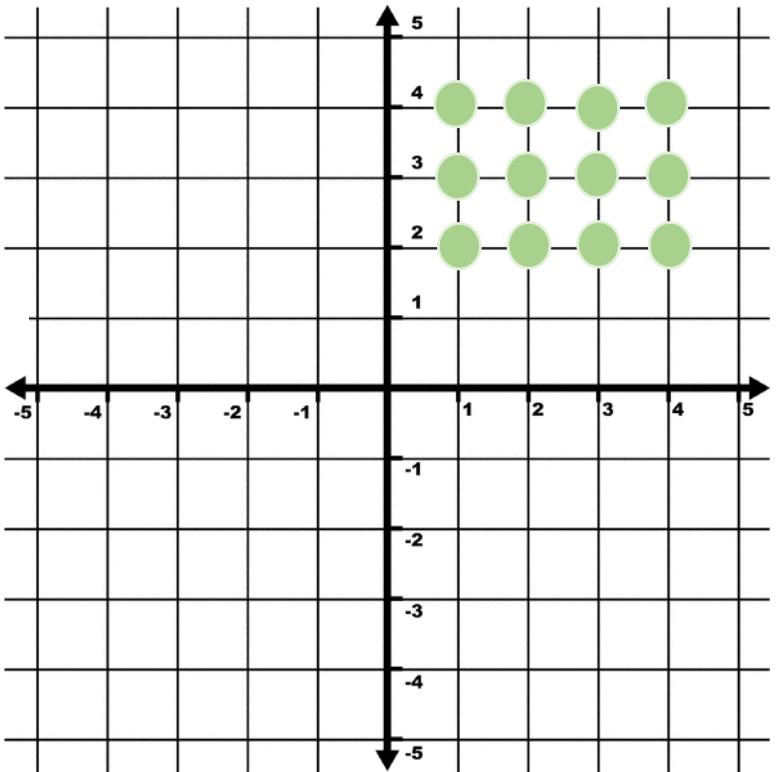


- Get a set of code vectors from the grid:

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

Visualize code vectors.

Code vectors



- Get a set of code vectors from the grid:

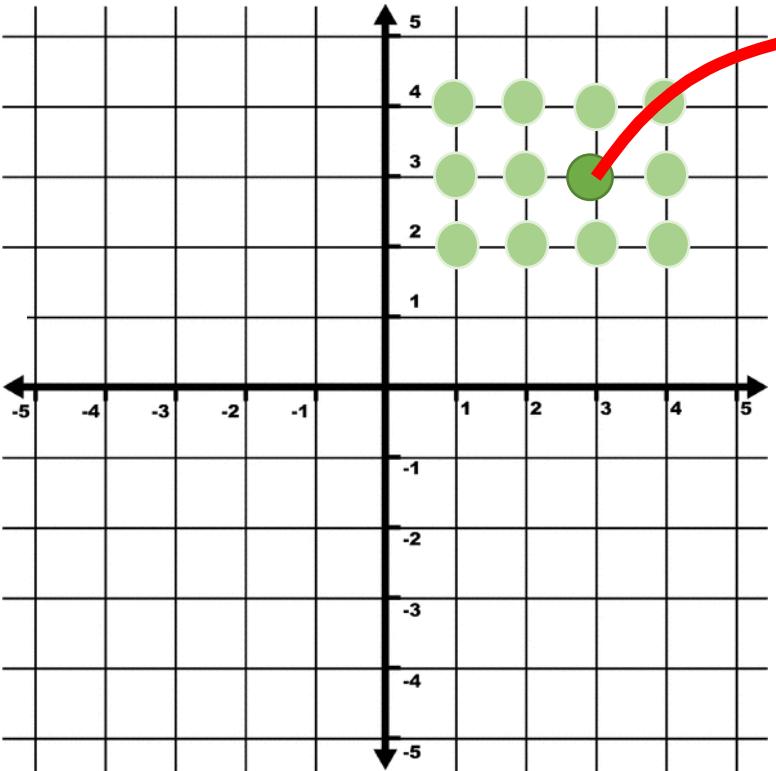
$$\mathbf{z}_1, \mathbf{z}_2, \mathbf{z}_3, \mathbf{z}_4, \mathbf{z}_5, \dots$$

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

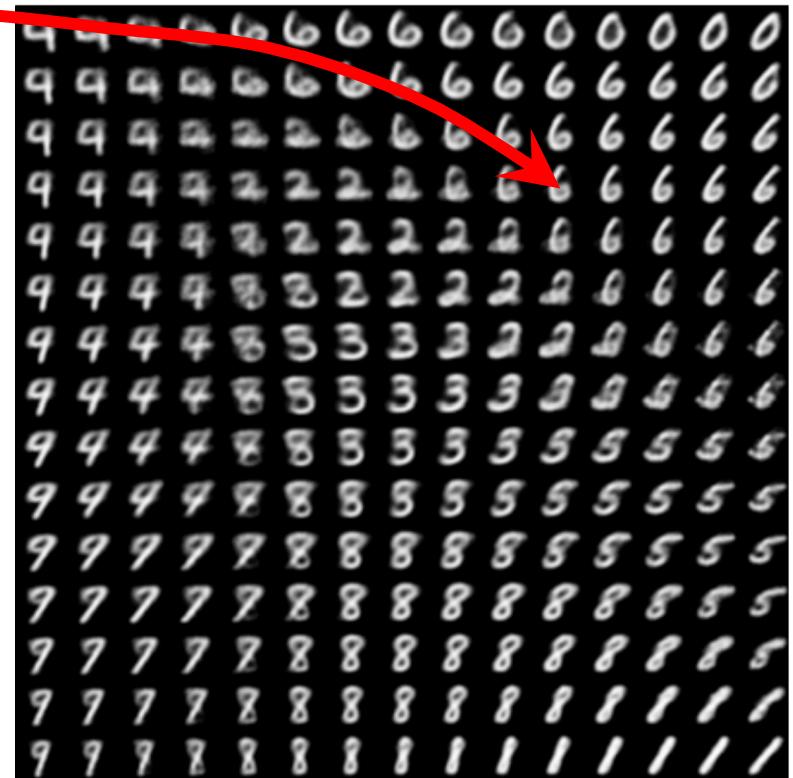
$$\text{image}_i = \text{Decoder}(\mathbf{z}_i).$$

Visualize code vectors.

Code vectors



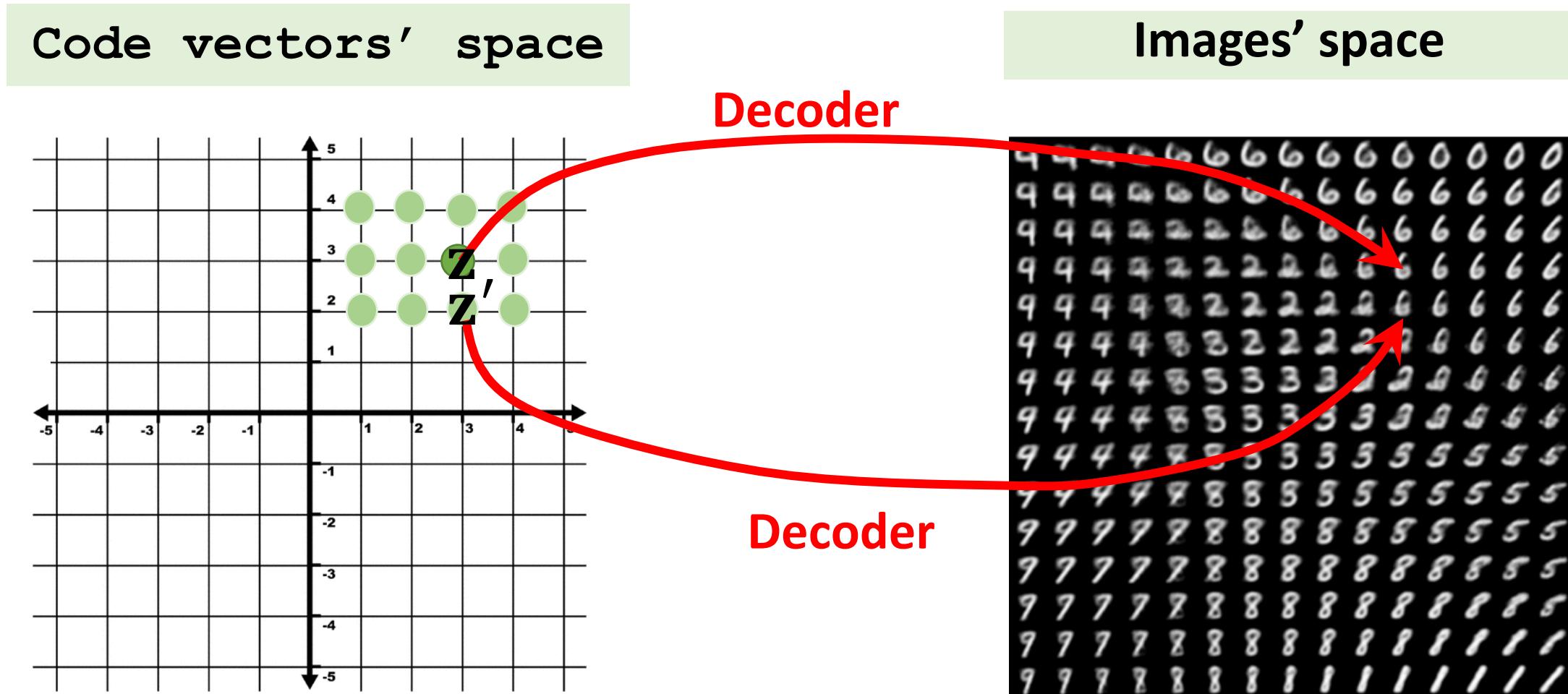
Images



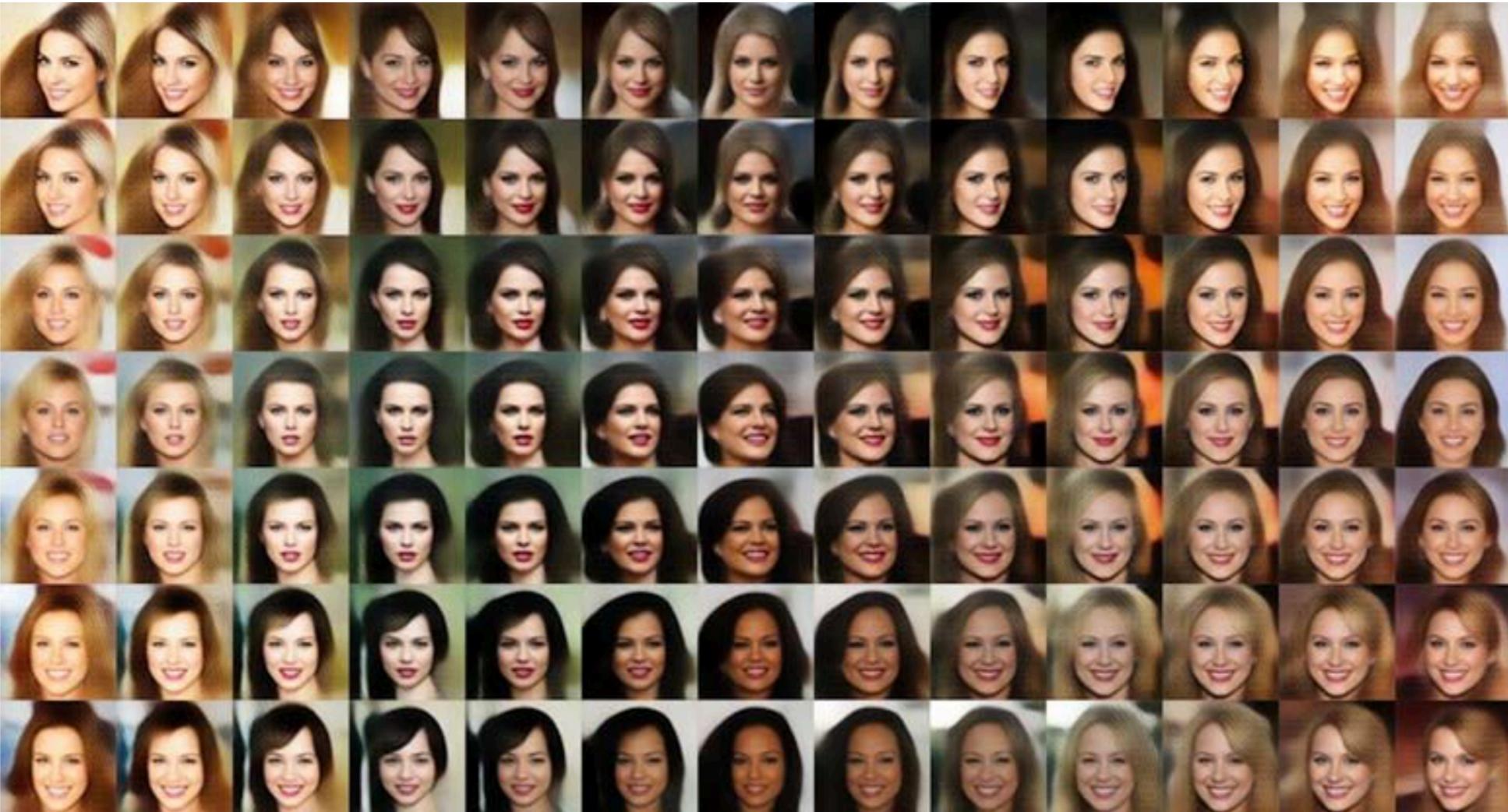
The decoder is (almost) continuous function.

- Function f is continuous if a small change in \mathbf{z} results in a small change in $f(\mathbf{z})$.
- The decoder network is trained to be (almost) continuous.
- If the code vectors \mathbf{z} and \mathbf{z}' are close, then the images
 $\text{Decoder}(\mathbf{z})$ and $\text{Decoder}(\mathbf{z}')$
are also close.

The decoder is (almost) continuous function.

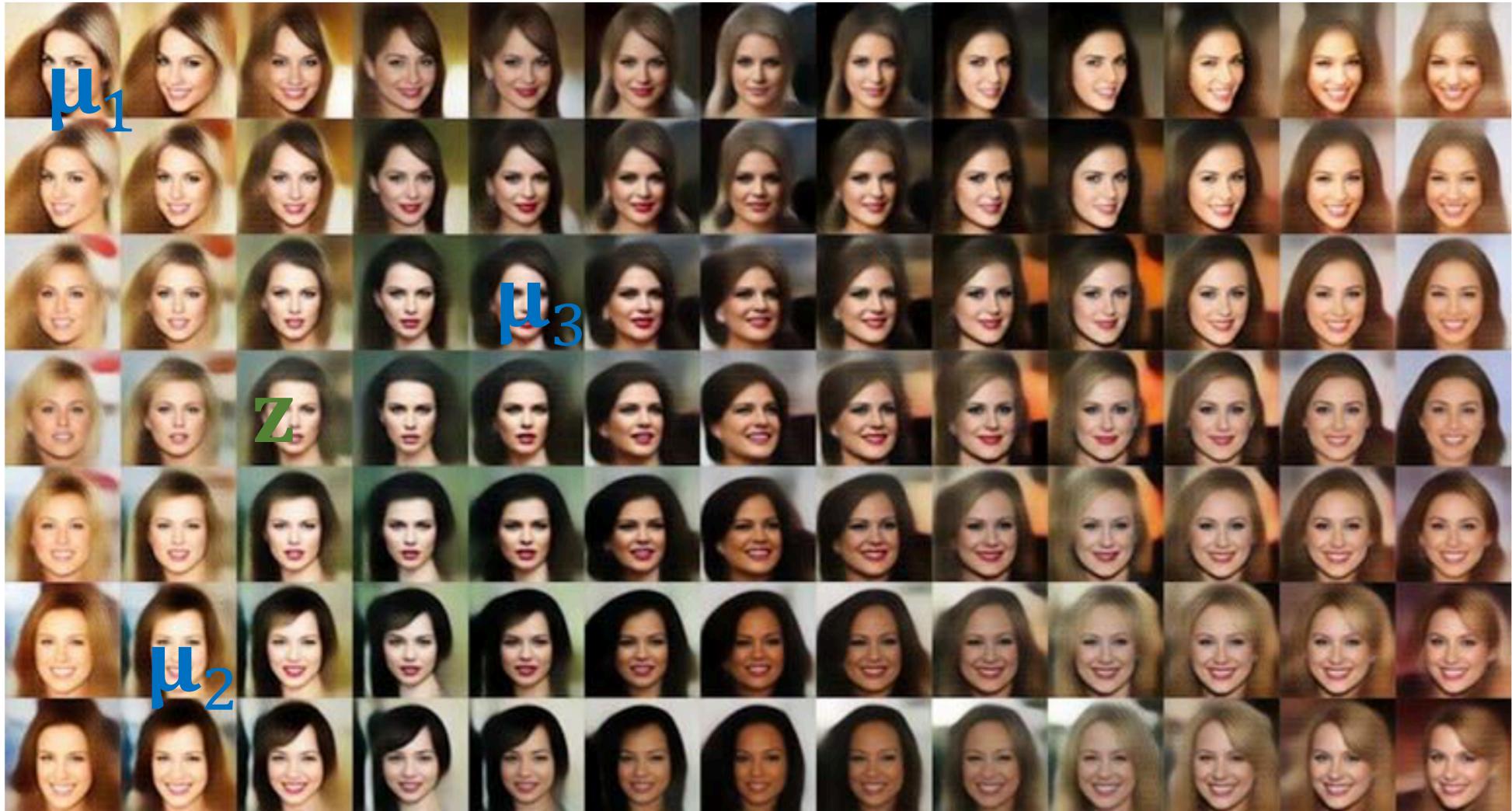


The decoder is (almost) continuous function.



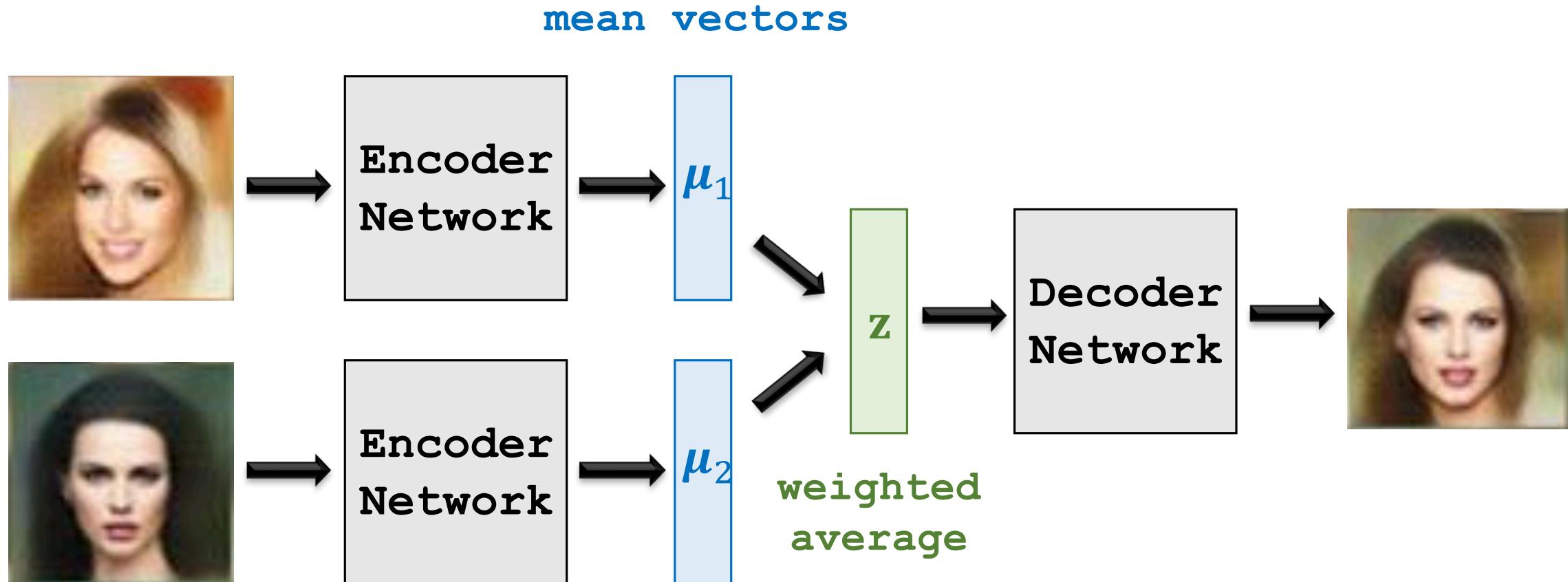
Visualization of the generated faces (by Tom White)

Averaging images via averaging code vectors

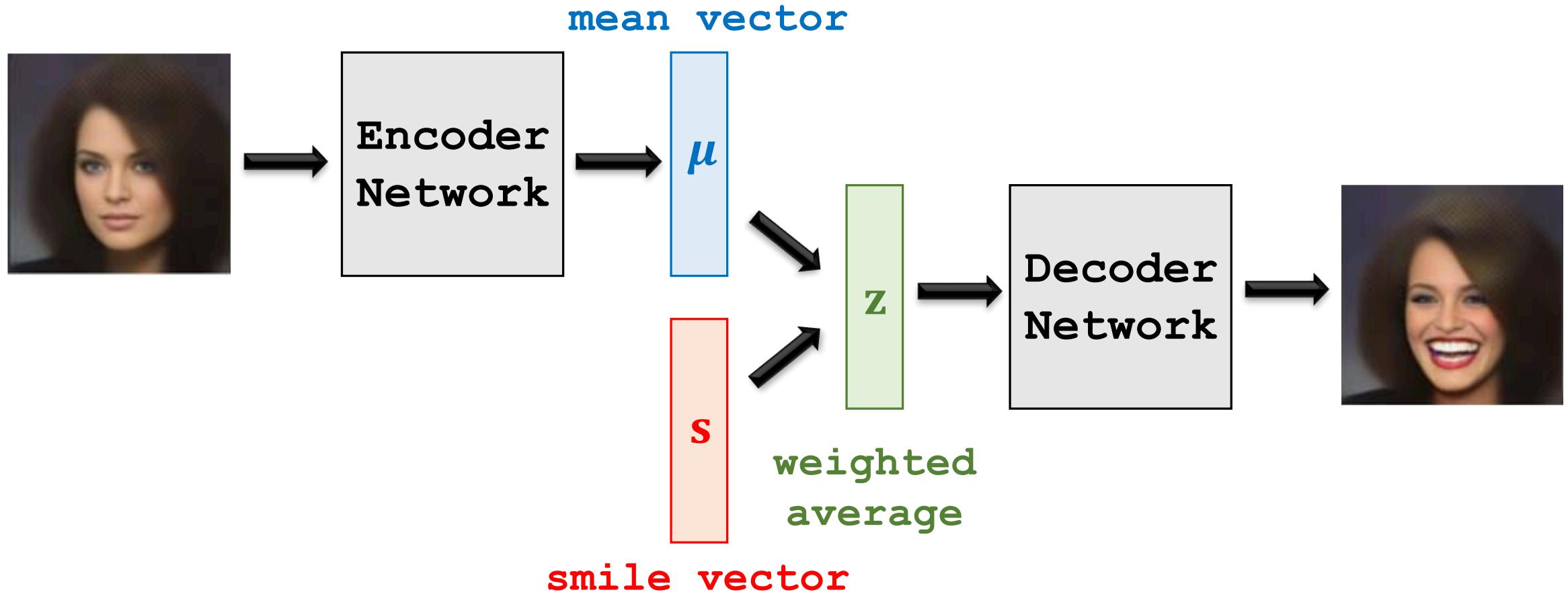


Visualization of the generated faces (by Tom White)

Averaging images via averaging code vectors



Editing images via editing code vectors



Editing images via editing code vectors

μ

$\mu + \mathbf{s}$

$\mu + 2\mathbf{s}$

$\mu + 3\mathbf{s}$

$\mu + 4\mathbf{s}$



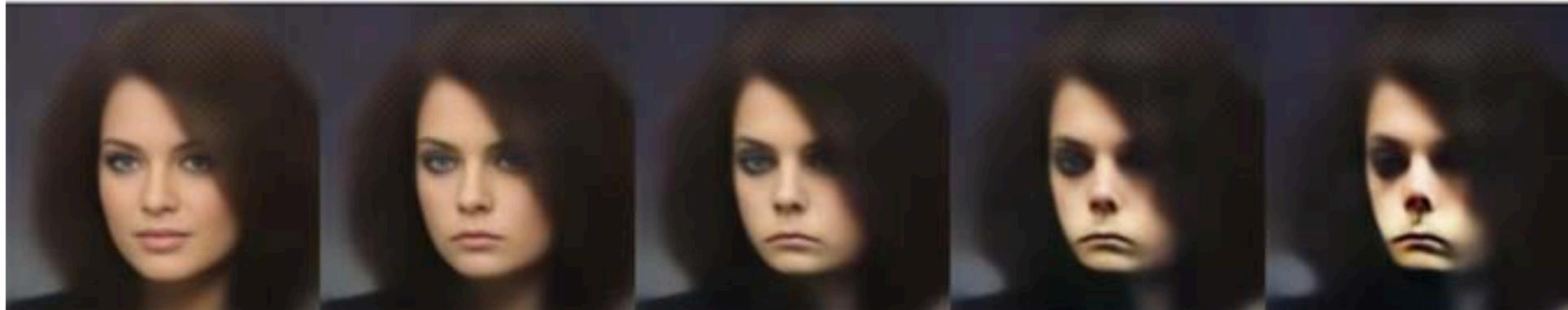
μ

$\mu - \mathbf{s}$

$\mu - 2\mathbf{s}$

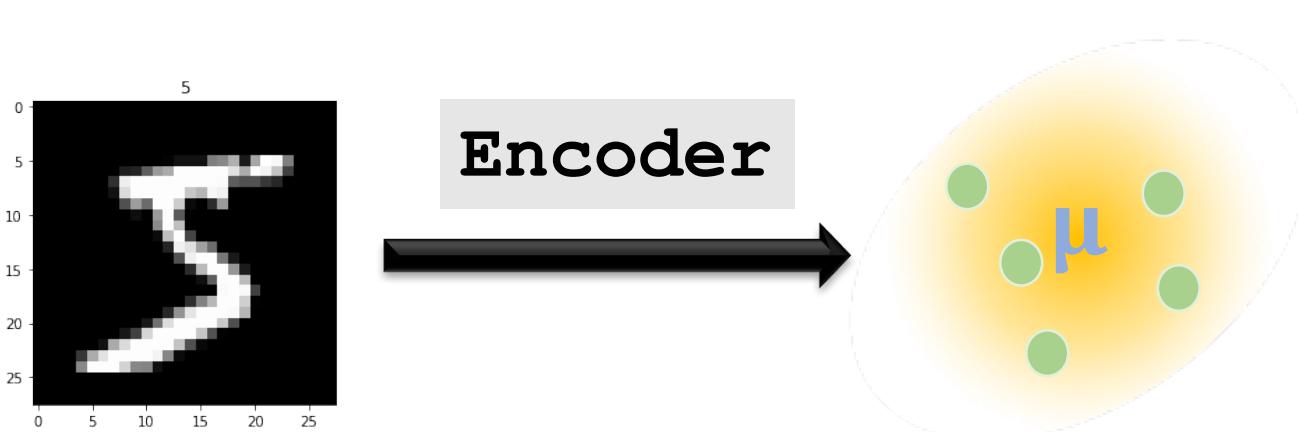
$\mu - 3\mathbf{s}$

$\mu - 4\mathbf{s}$



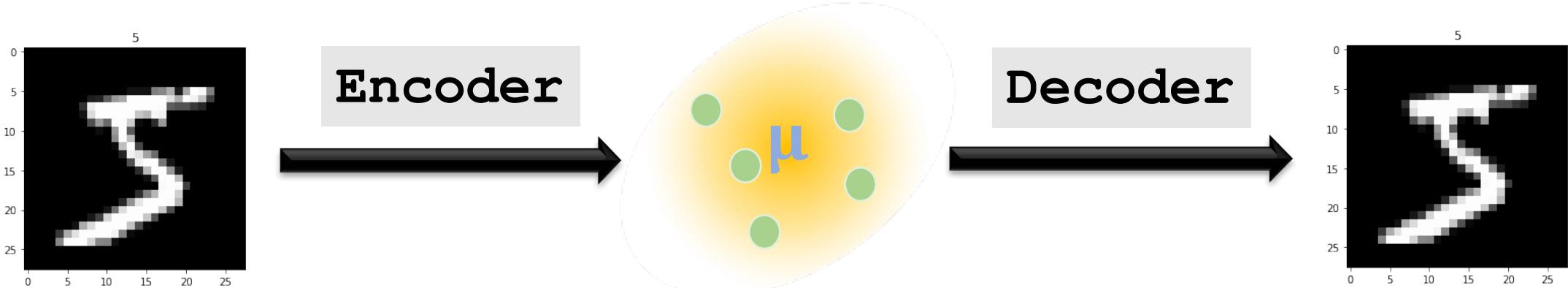
What makes the decoder continuous?

- Given μ and σ , generate $\mathbf{z}_i \sim \mathcal{N}(\mu, \text{diag}(\sigma^2))$, for $i = 1, 2, 3, \dots$.



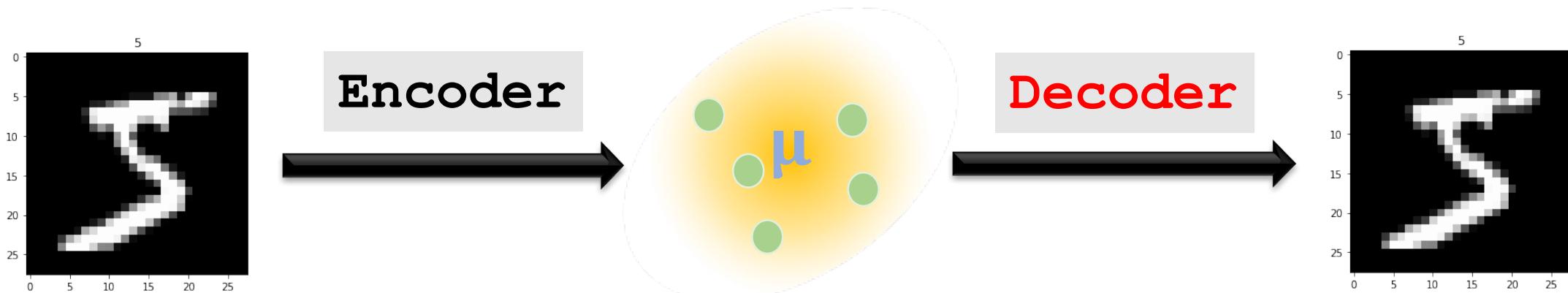
What makes the decoder continuous?

- Given μ and σ , generate $\mathbf{z}_i \sim \mathcal{N}(\mu, \text{diag}(\sigma^2))$, for $i = 1, 2, 3, \dots$.
- All the code vectors $\mathbf{z}_1, \mathbf{z}_2, \mathbf{z}_3, \dots$ have the same target (which is the input image.)



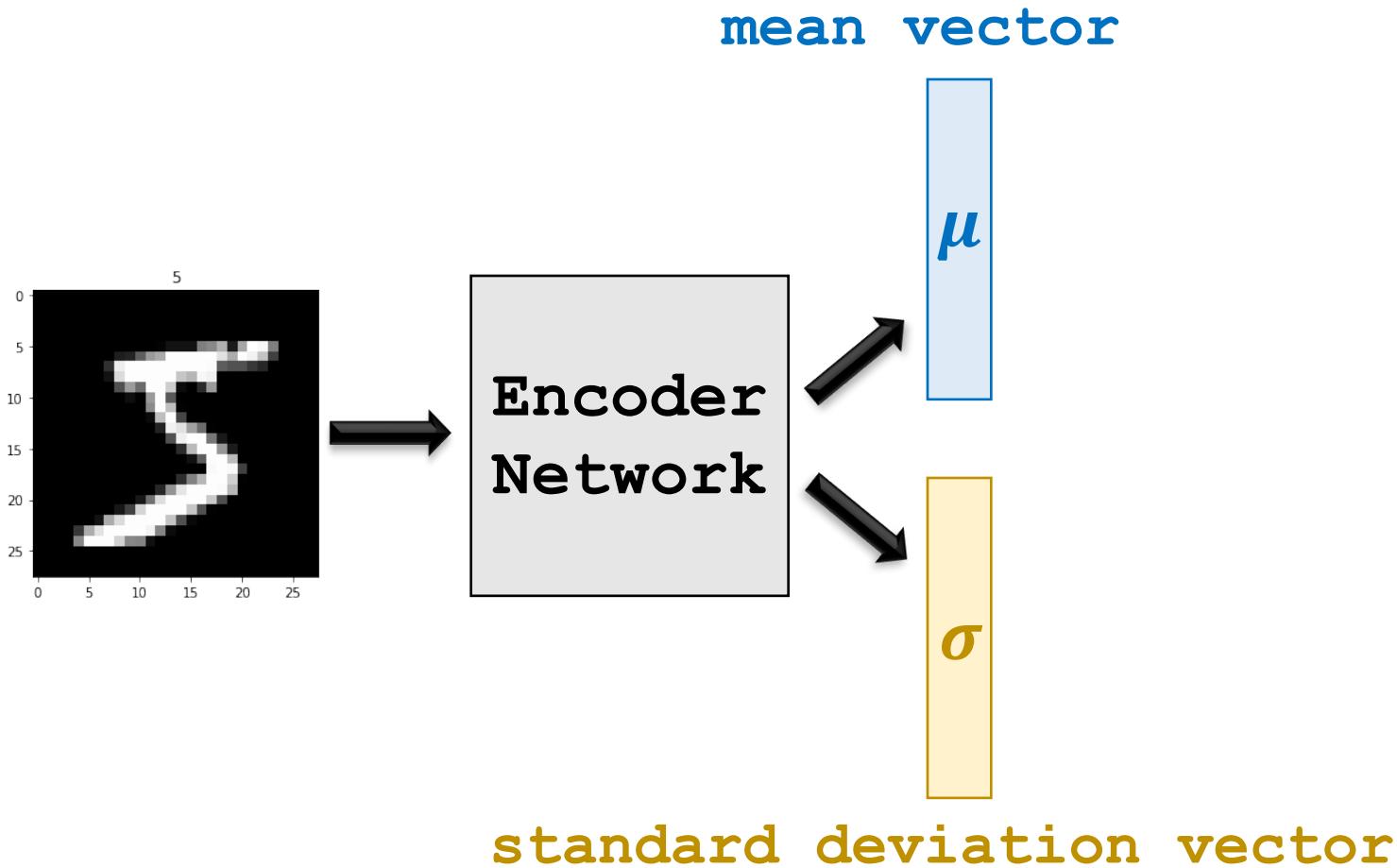
What makes the decoder continuous?

- Given μ and σ , generate $\mathbf{z}_i \sim \mathcal{N}(\mu, \text{diag}(\sigma^2))$, for $i = 1, 2, 3, \dots$.
- All the code vectors $\mathbf{z}_1, \mathbf{z}_2, \mathbf{z}_3, \dots$ have the same target (which is the input image.)
- Thus, $\text{Decoder}(\mathbf{z}_1)$, $\text{Decoder}(\mathbf{z}_2)$, $\text{Decoder}(\mathbf{z}_3)$, \dots must be similar.
- Inputs (code vectors) are similar; outputs (generated images) are similar.
- → The decoder is trained to be (almost) a continuous function.



Build the Networks

1. The Encoder Network



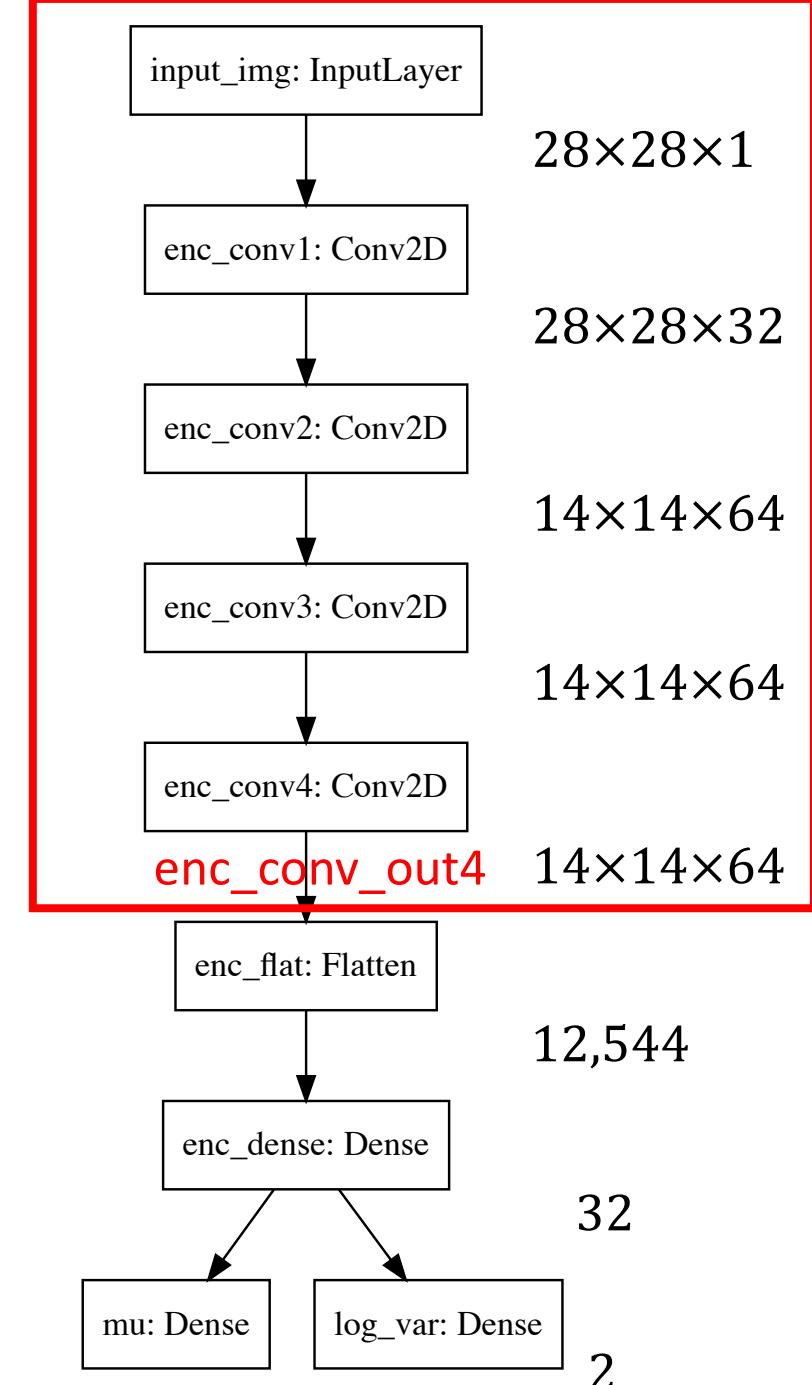
1. The Encoder Network

```
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')

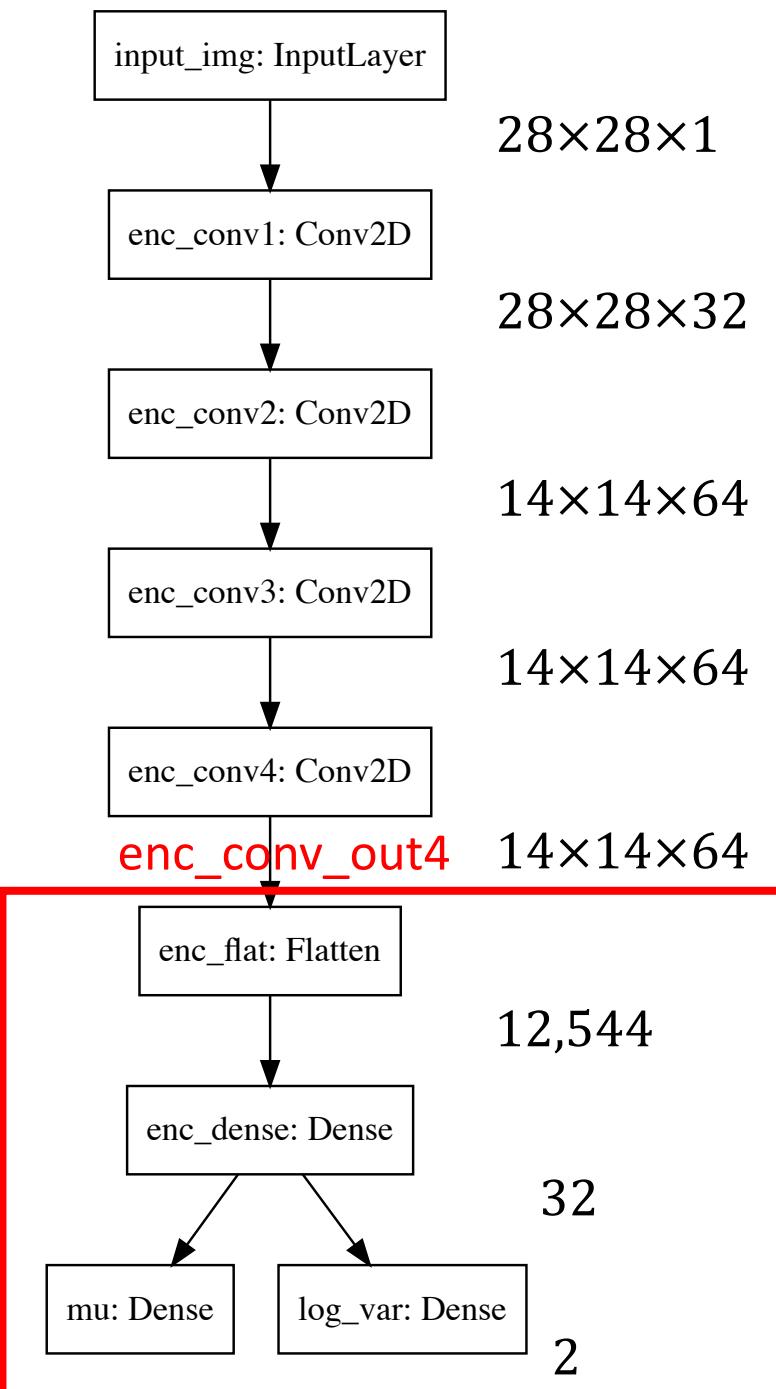
input_img = Input(shape=(28, 28, 1), name='input_img')
enc_conv_out1 = enc_conv1(input_img)
enc_conv_out2 = enc_conv2(enc_conv_out1)
enc_conv_out3 = enc_conv3(enc_conv_out2)
enc_conv_out4 = enc_conv4(enc_conv_out2)
```



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)
```

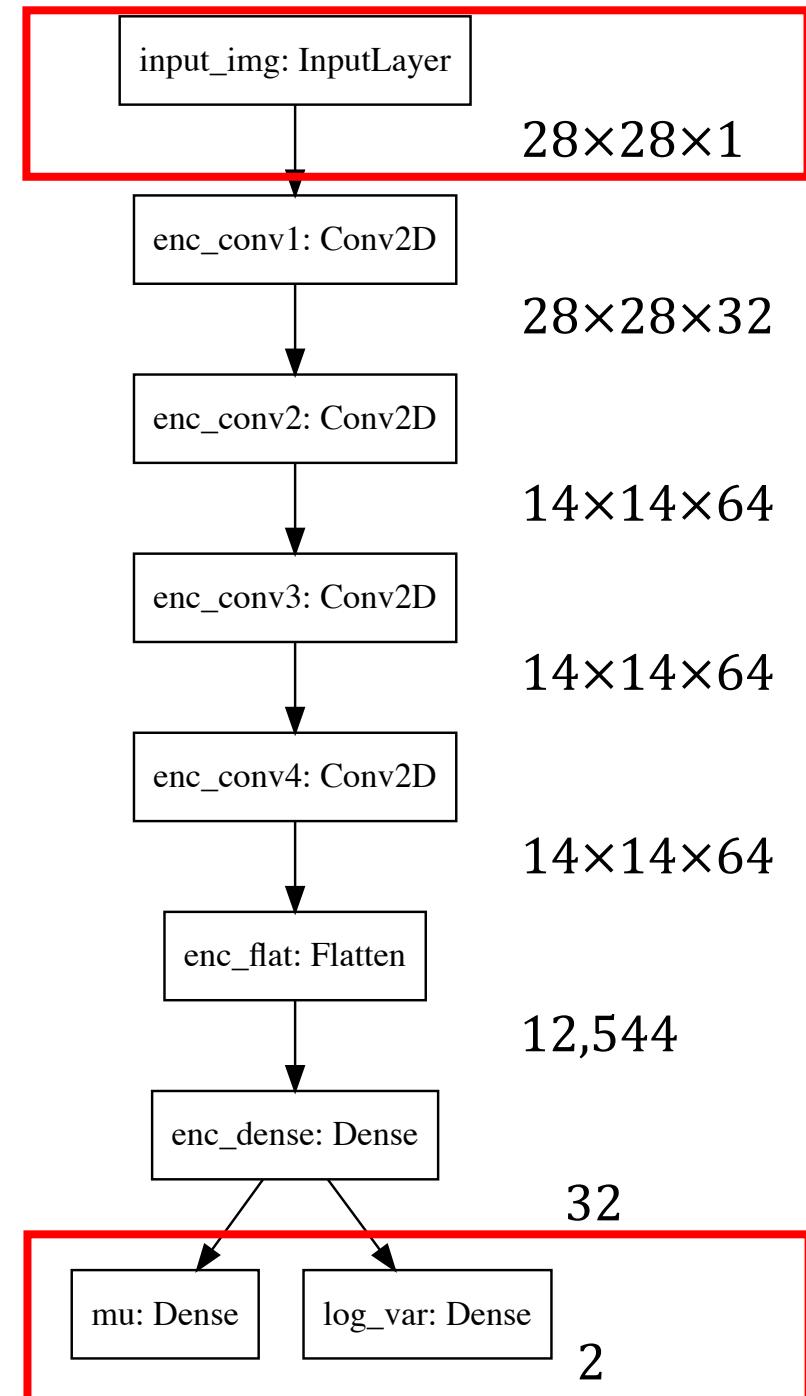


1. The Encoder Network

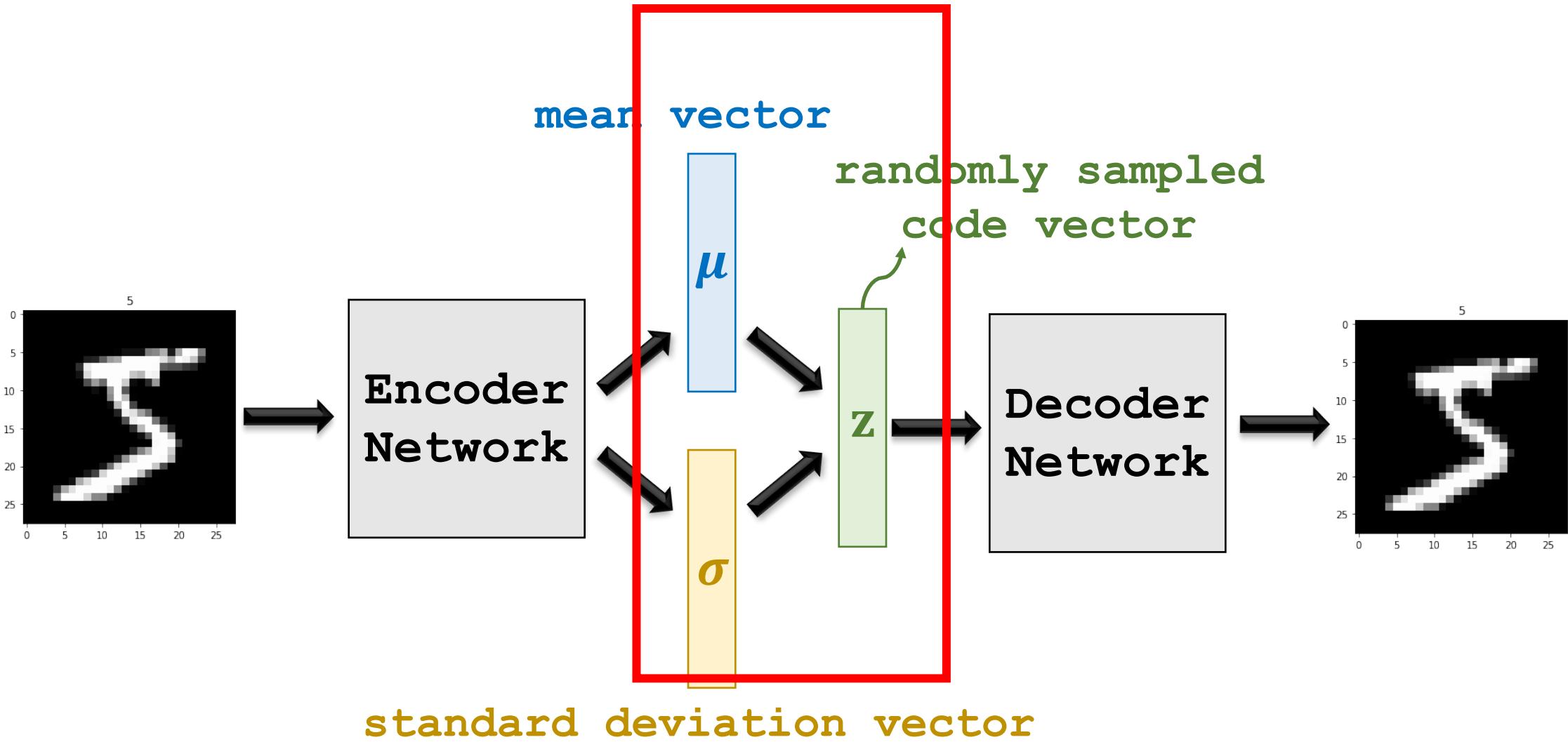
```
# define flatten and dense layers
enc_flat = Flatten(name='enc_flat')
enc_dense = Dense(32, activation='relu',
                  name='enc_dense')
enc_mu = Dense(shape_z, name='mu')
enc_log_var = Dense(shape_z, name='log_var')

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

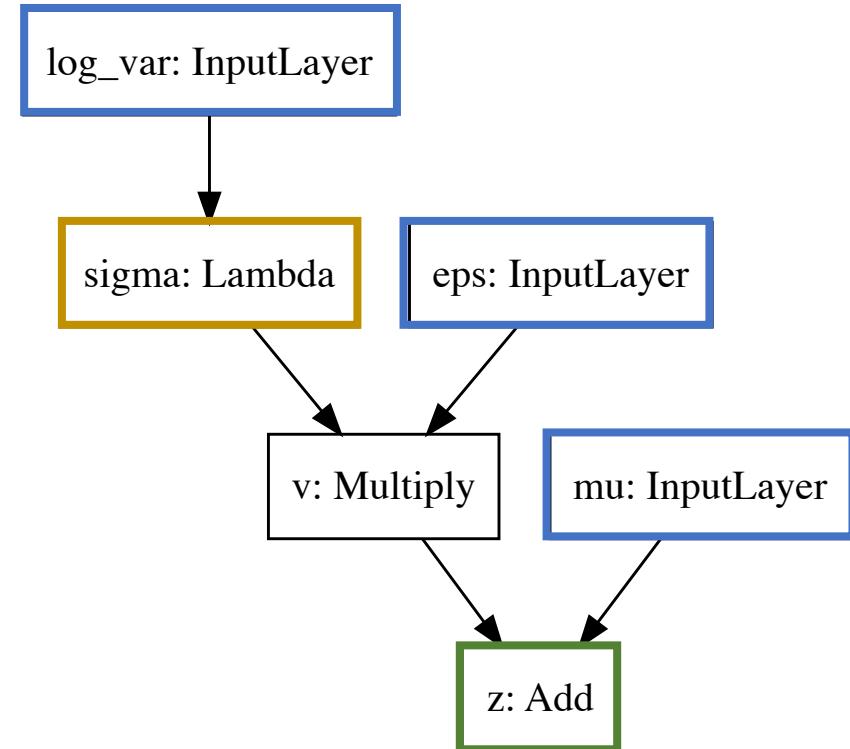


2. The Sampling Network

- Given μ and σ , randomly sample \mathbf{z} from $\mathcal{N}(\mu, \text{diag}(\sigma^2))$.
- The i -th entry of \mathbf{z} (denote z_i) is randomly sampled from $\mathcal{N}(\mu_i, \sigma_i^2)$.
- Equivalently,
 - Randomly sample scalar variable ϵ_i from $\mathcal{N}(0, 1)$.
 - Let $v_i = \sigma_i \cdot \epsilon_i$.
 - $z_i = \mu_i + v_i$.

2. The Sampling Network

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



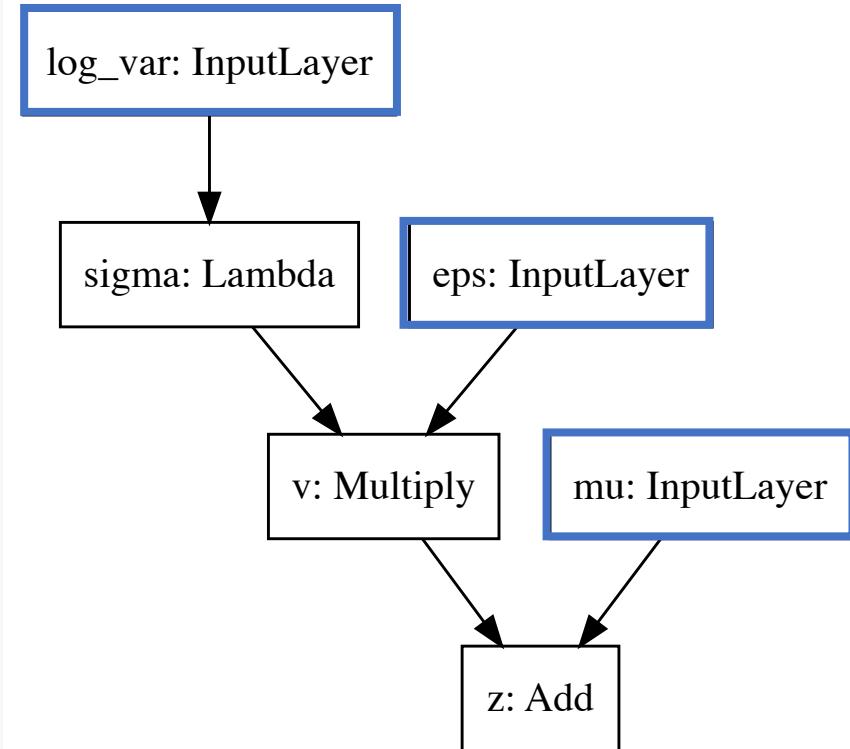
2. The Sampling Network

```
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')
```



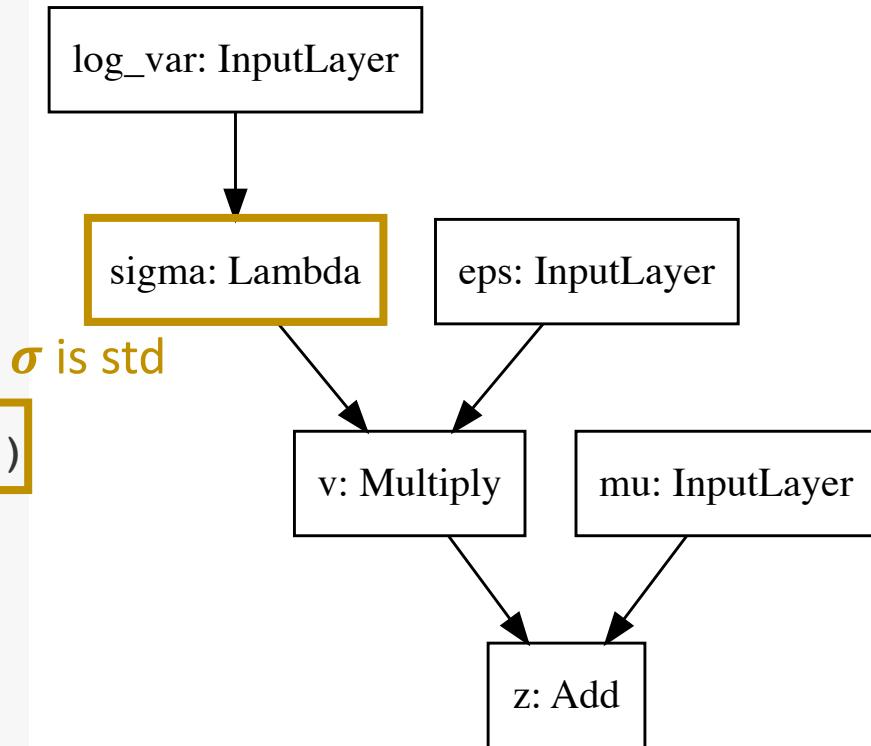
2. The Sampling Network

```
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')
```



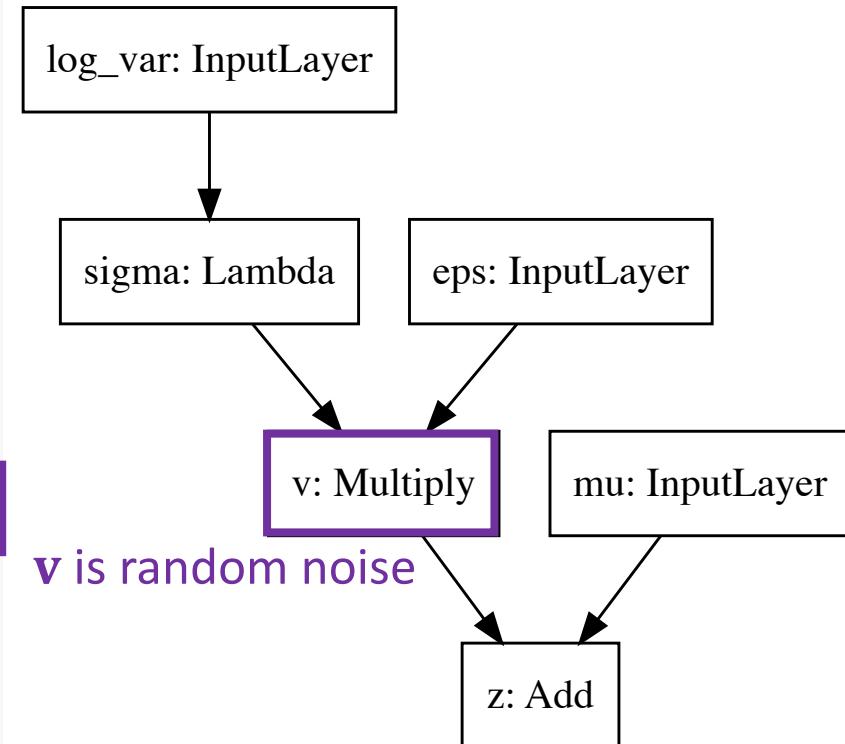
2. The Sampling Network

```
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')
```



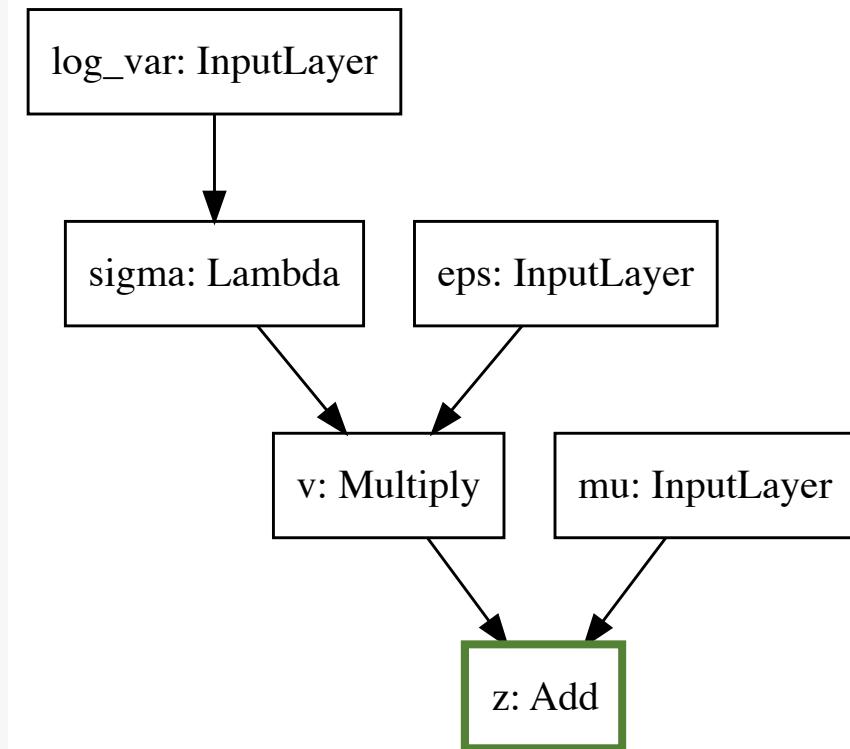
2. The Sampling Network

```
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')
```



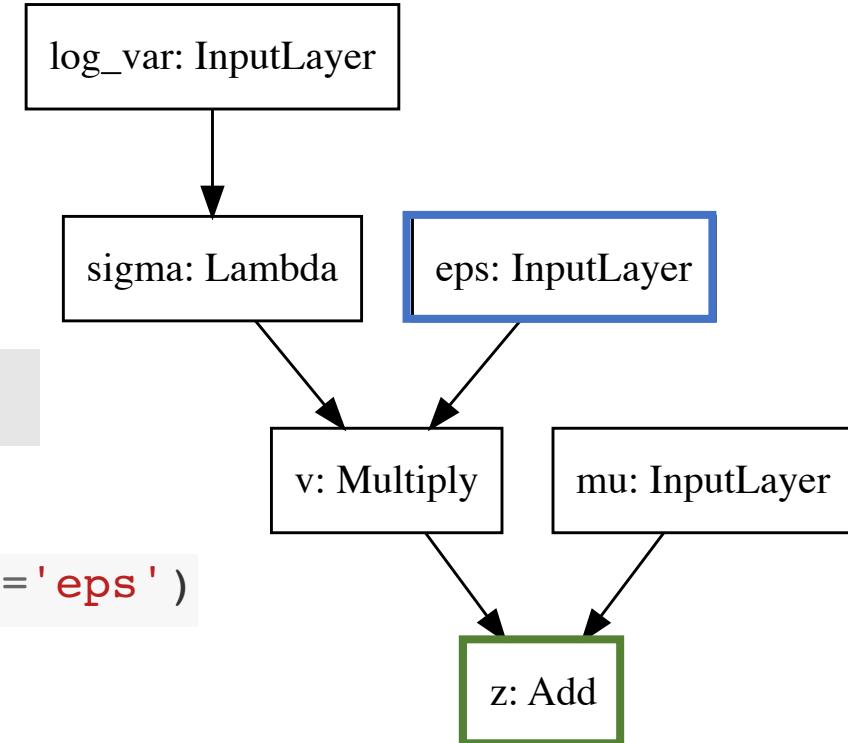
`z` is the sampled latent vector

2. The Sampling Network

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

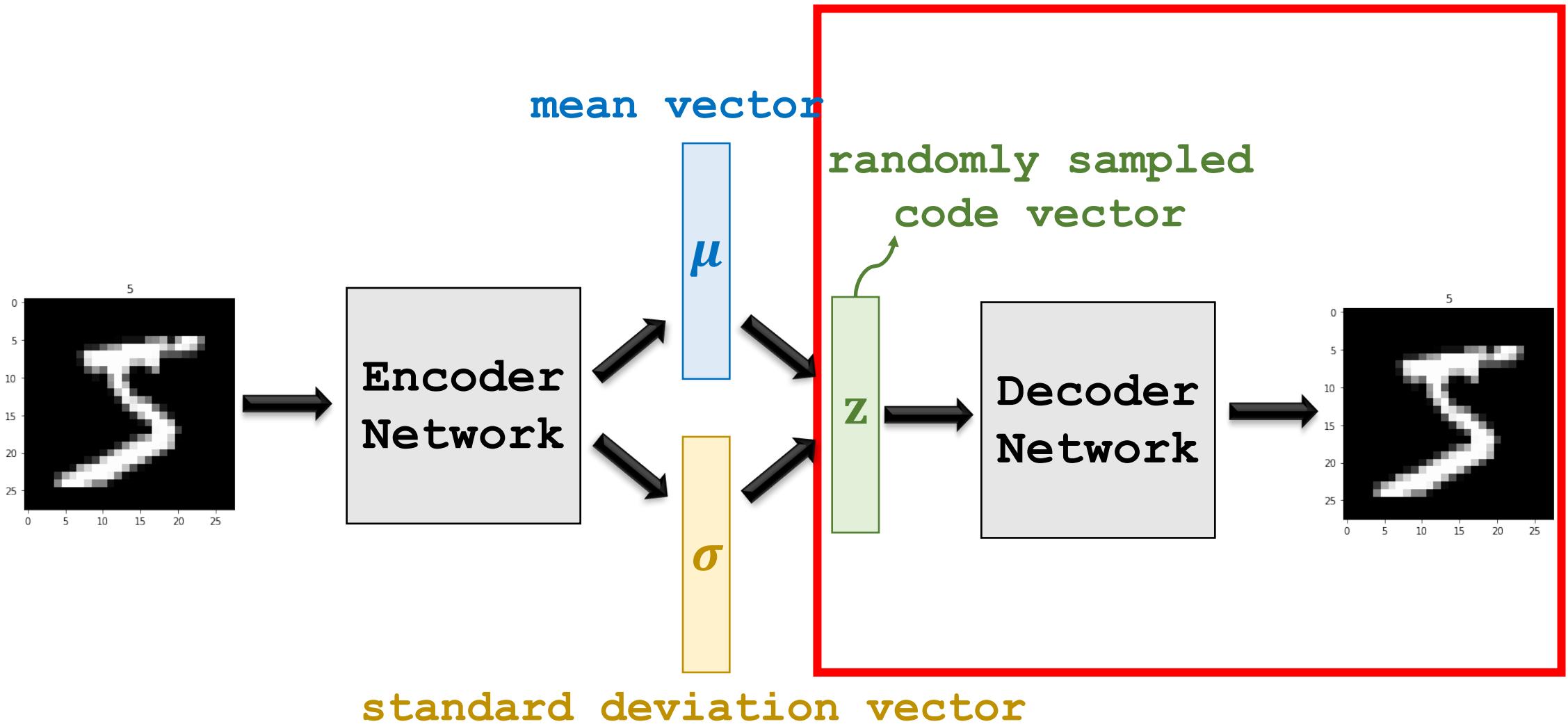
During training, set the input layer “eps” to the following code:

```
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

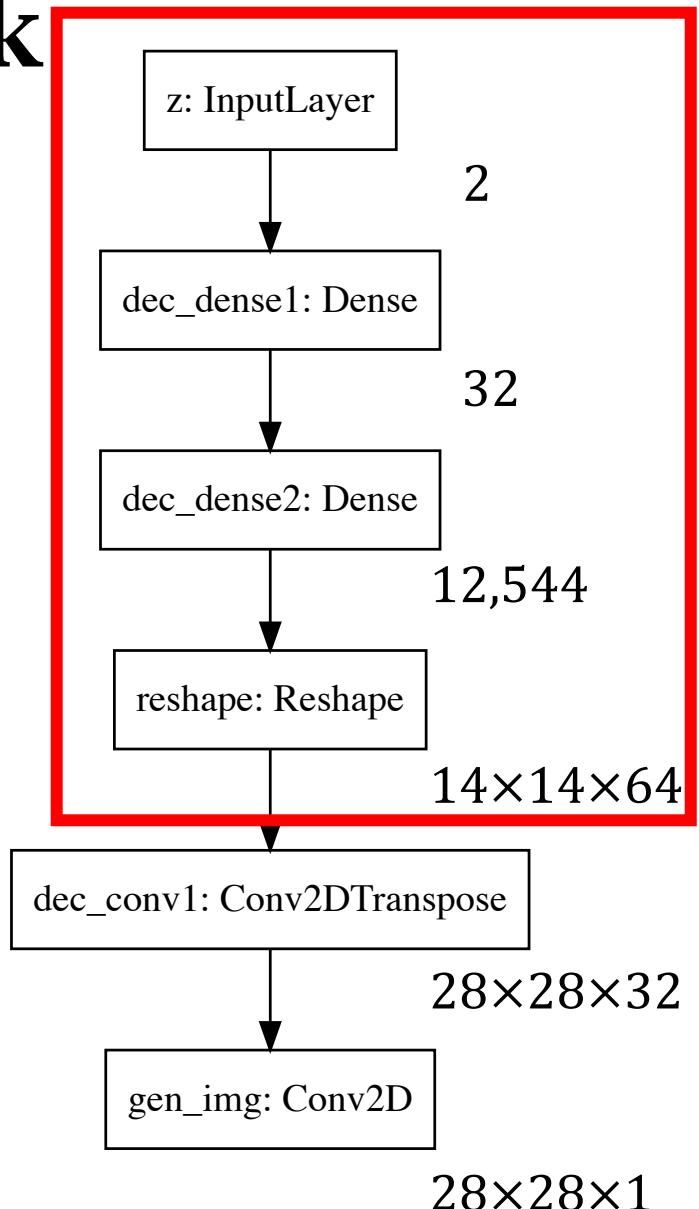
```
from keras import backend as K
import numpy

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)
```

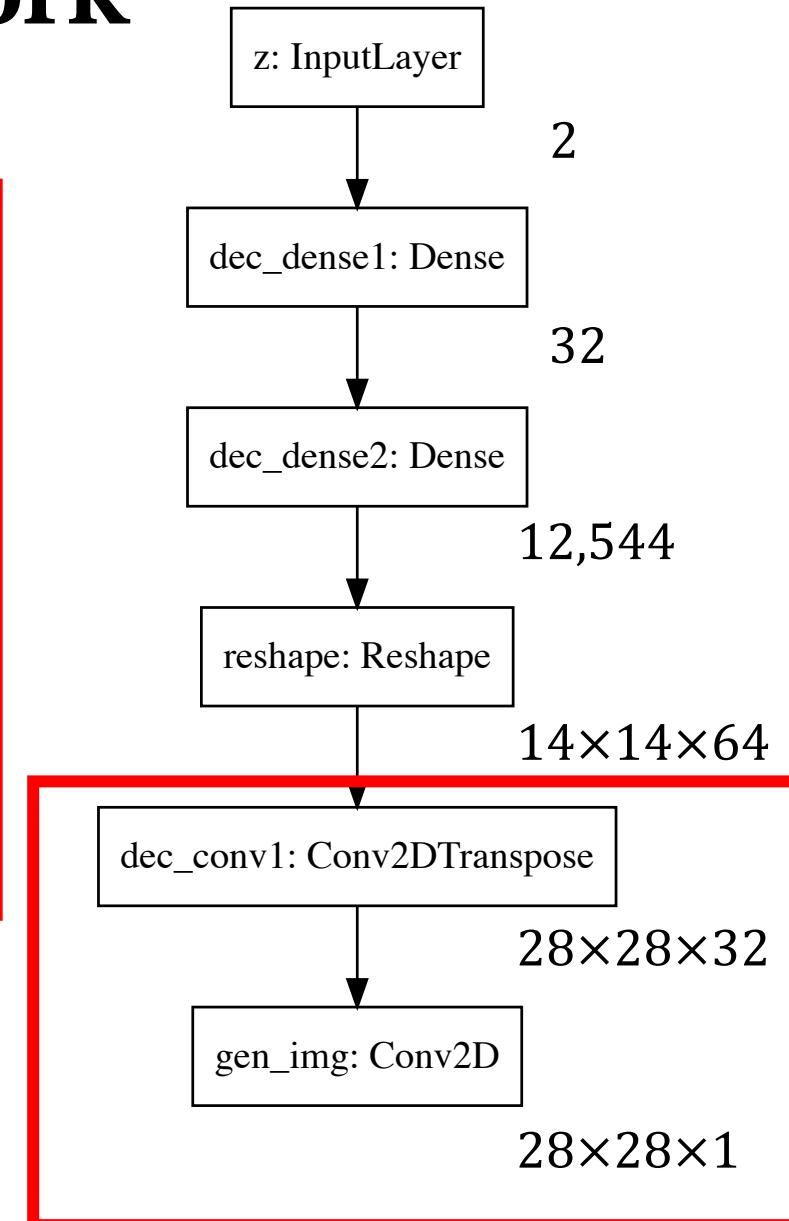


3. The Decoder Network

```
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')
```

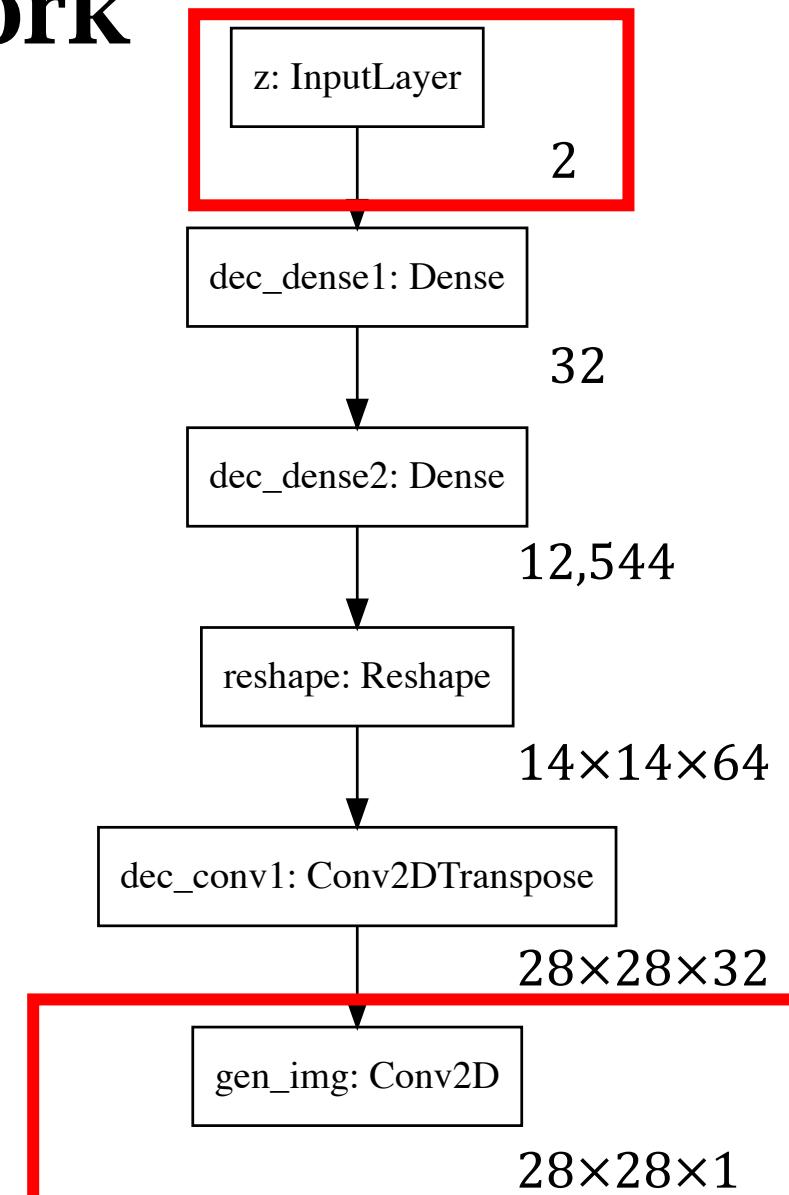


3. The Decoder Network

```
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)

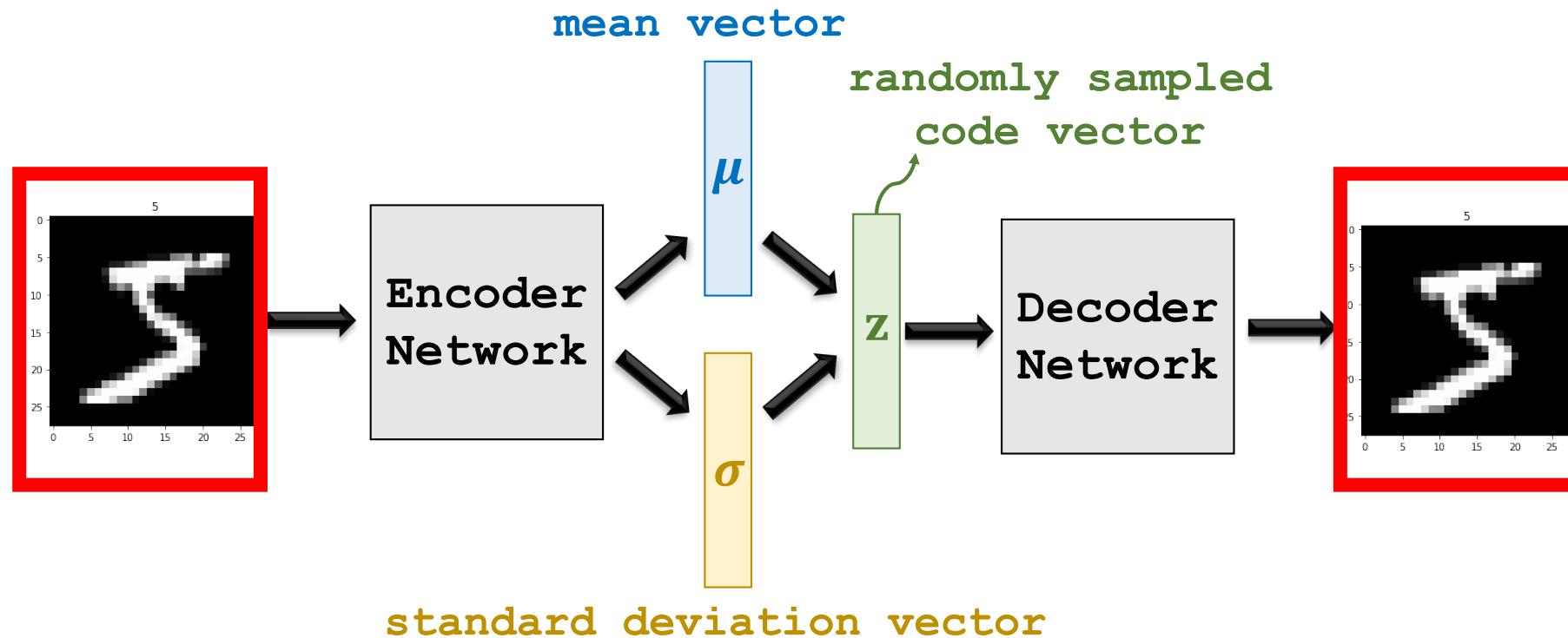
decoder = models.Model(inputs=z,
                      outputs=gen_img,
                      name='decoder')
```



Loss Functions

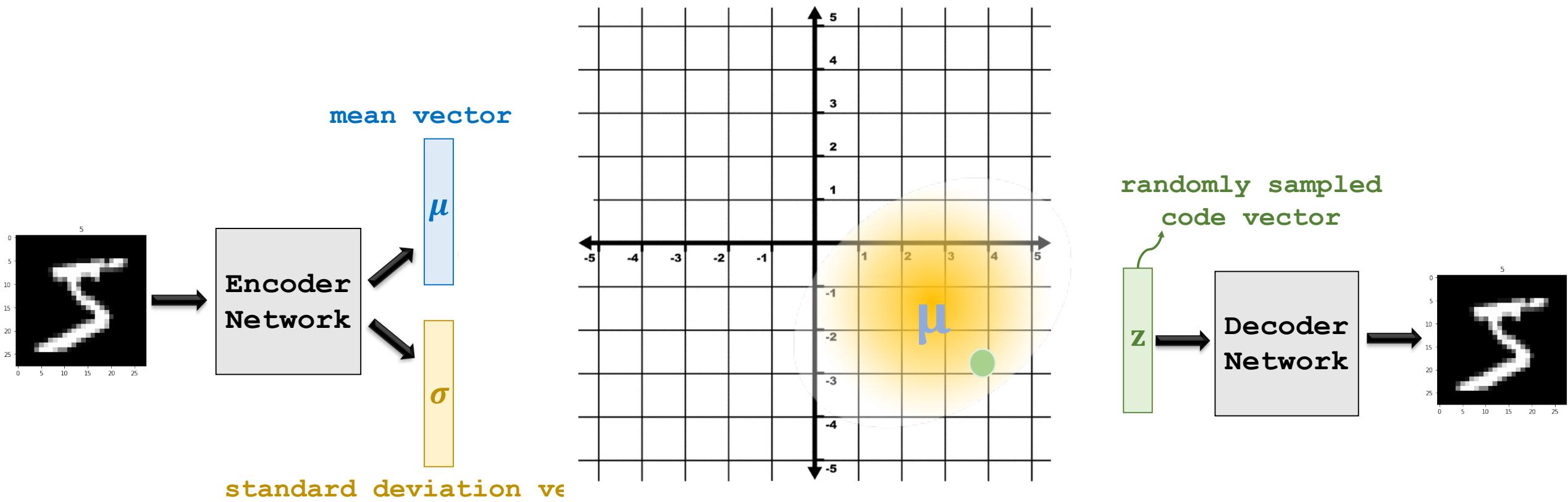
Generation Loss

- $\text{GenLoss} = \text{dist}(\text{input_img}, \text{generated_img})$.
- E.g., the ℓ_2 distance, the cross-entropy, etc.



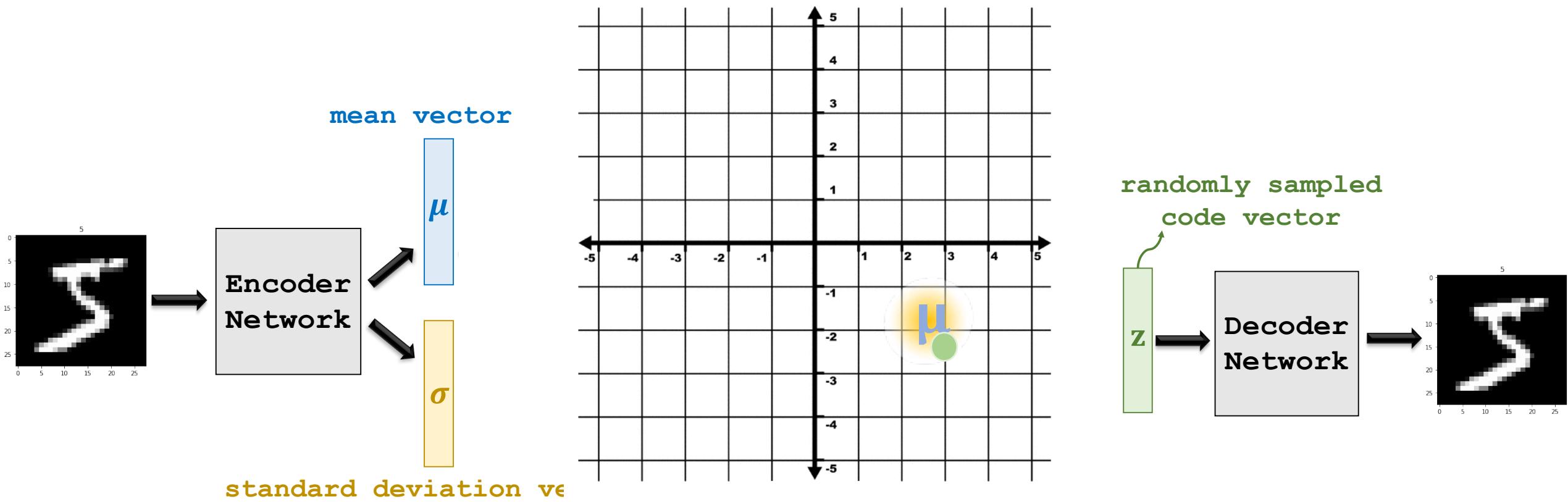
A problem with generation loss

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



A problem with generation loss

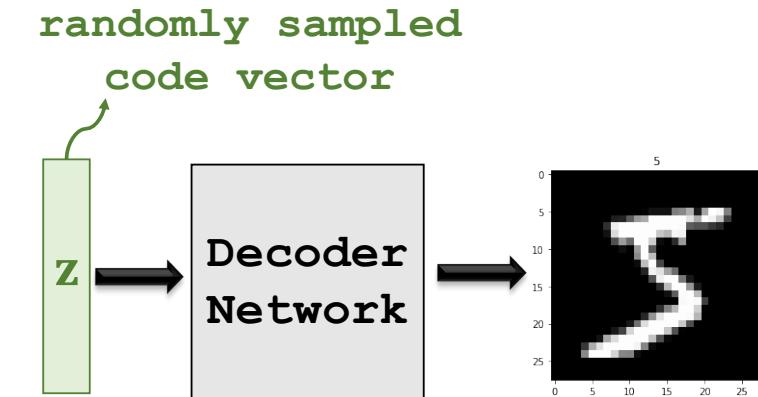
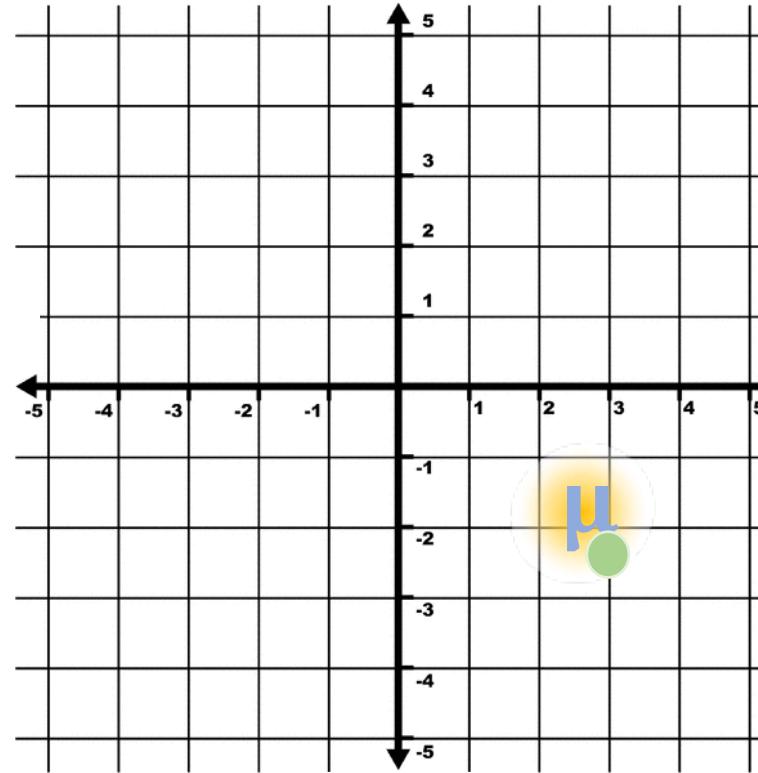
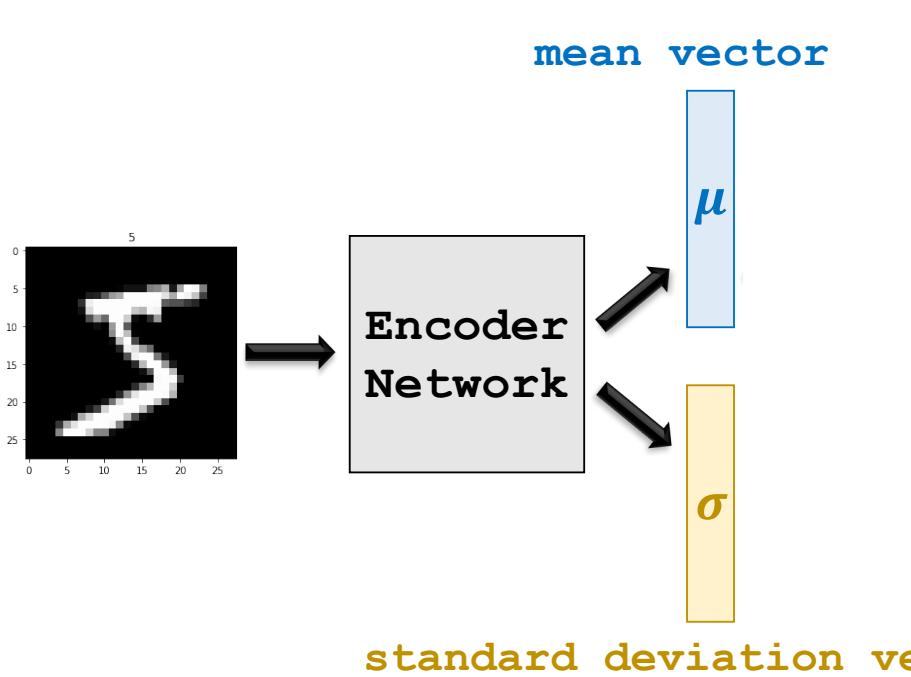
Difficulty: The encoder network will learn an std vector $\sigma \rightarrow 0$.



A problem with generation loss

Difficulty: The encoder network will learn an std vector $\sigma \rightarrow \mathbf{0}$. (Why?)

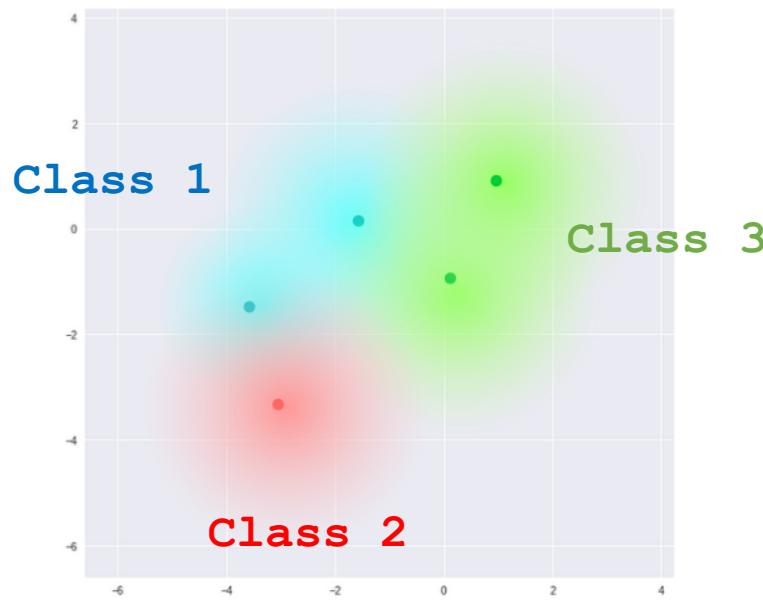
- Minimize GenLoss \rightarrow encourage \mathbf{z} close to μ \rightarrow encourage small σ .
- VAE degrades to the standard AE (VAE with $\sigma = \mathbf{0}$ is exactly AE).



A problem with generation loss

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

Dots are code vectors (μ); shadows illustrate std (σ).



What we hope to learn.

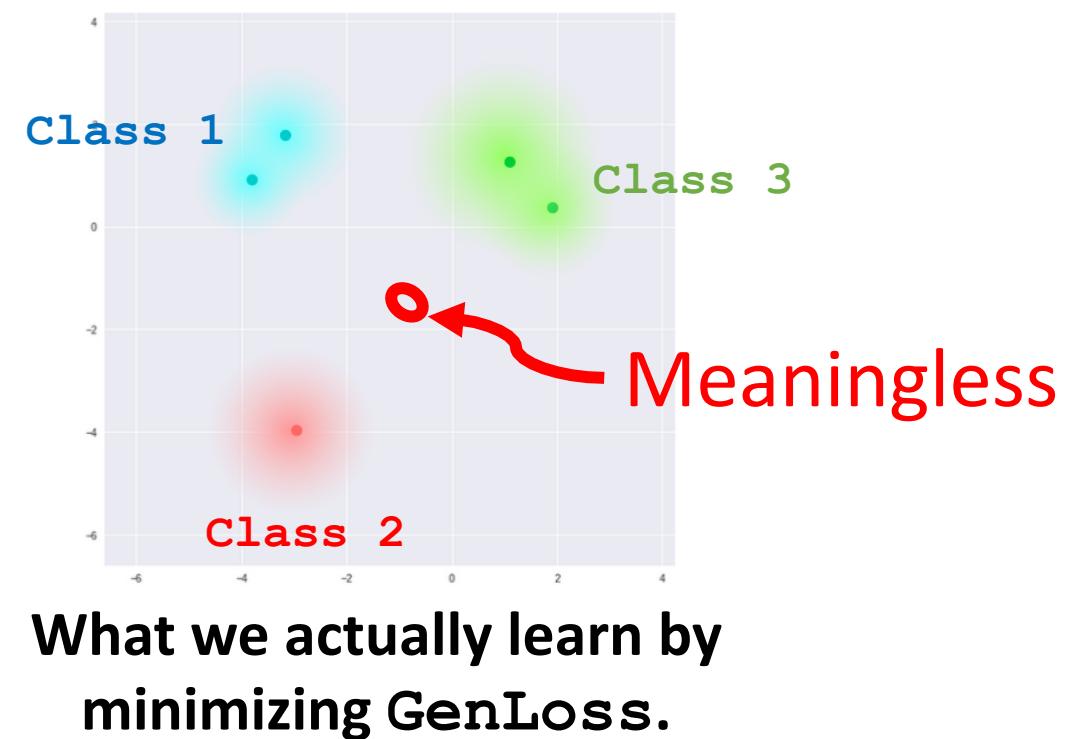
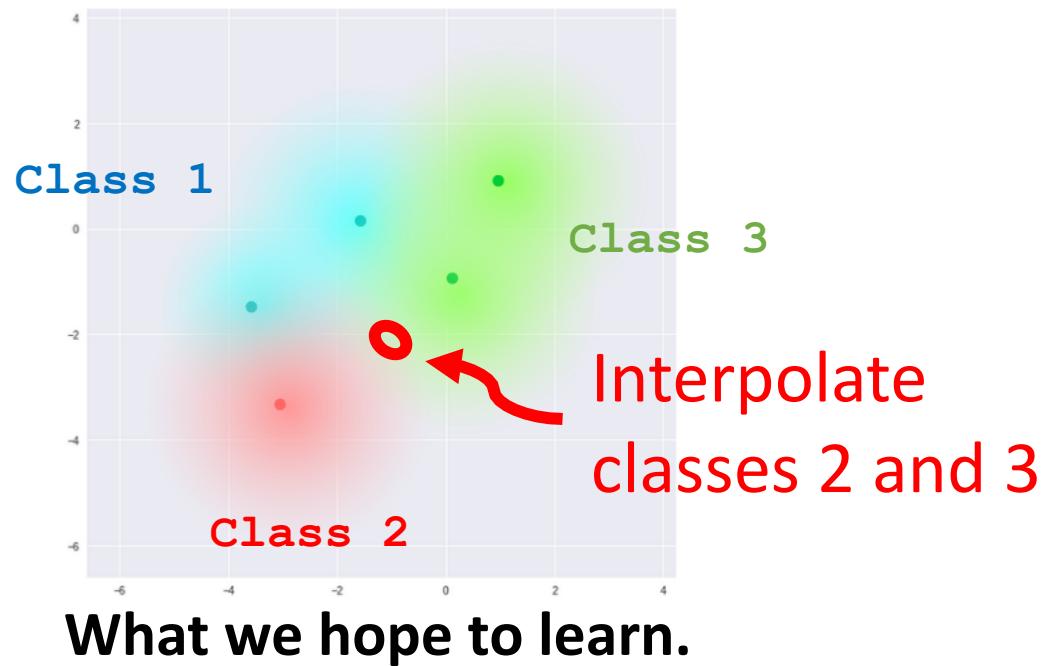


What we actually learn by
minimizing GenLoss.

A problem with generation loss

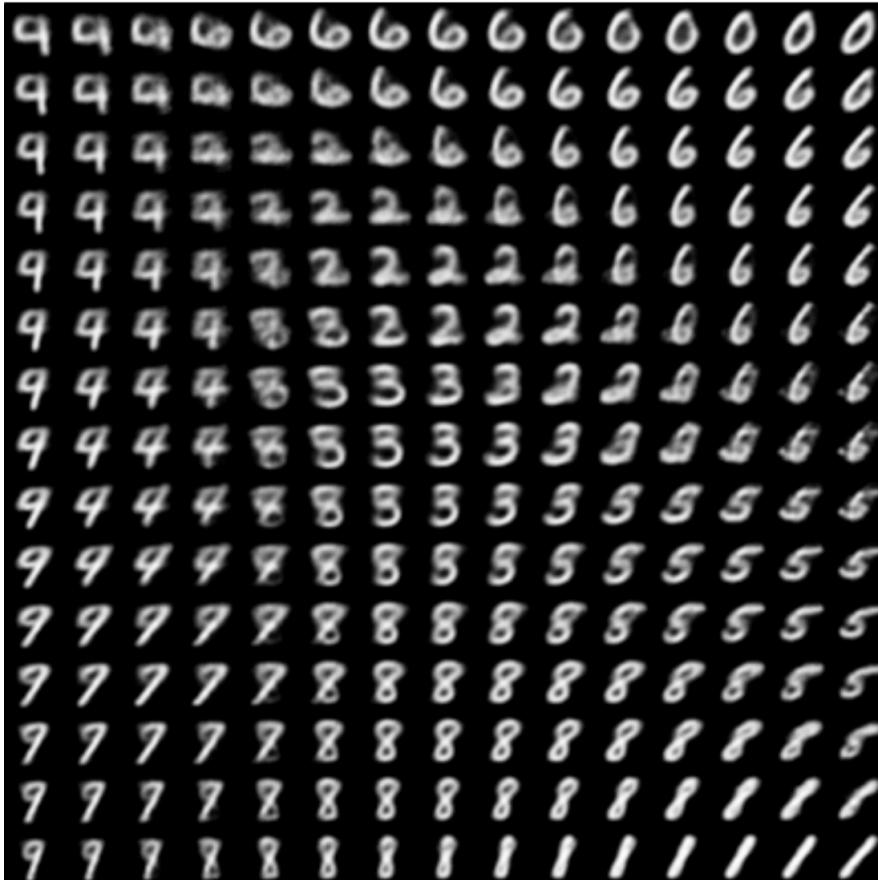
Difficulty: By minimizing generation loss, VAE becomes standard AE.

Dots are code vectors (μ); shadows illustrate std (σ).

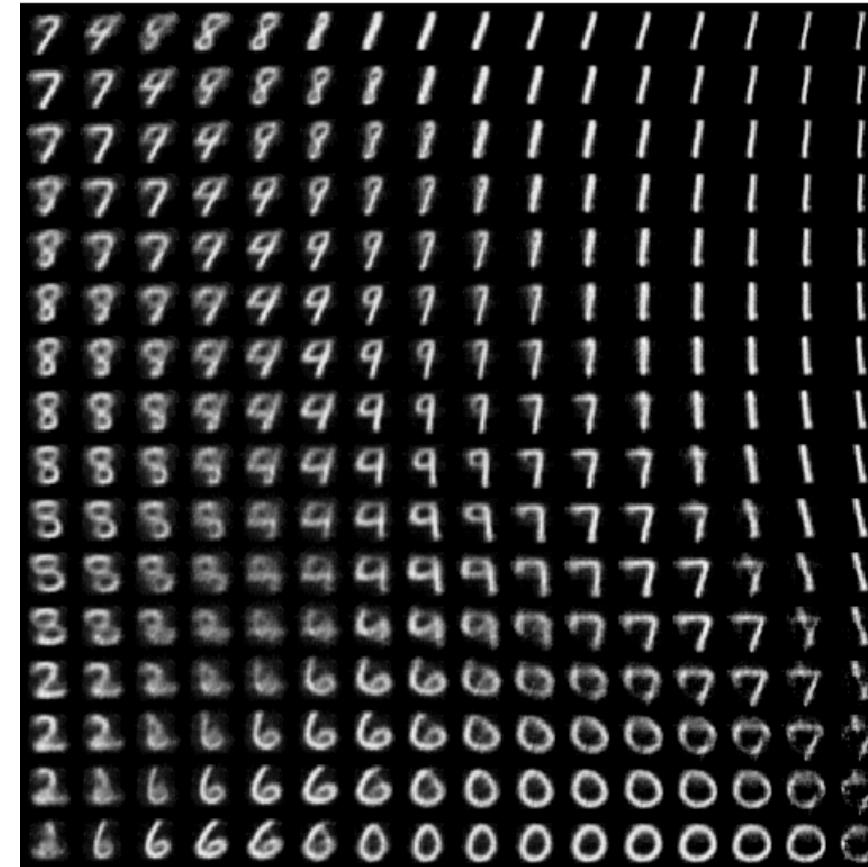


A problem with generation loss

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



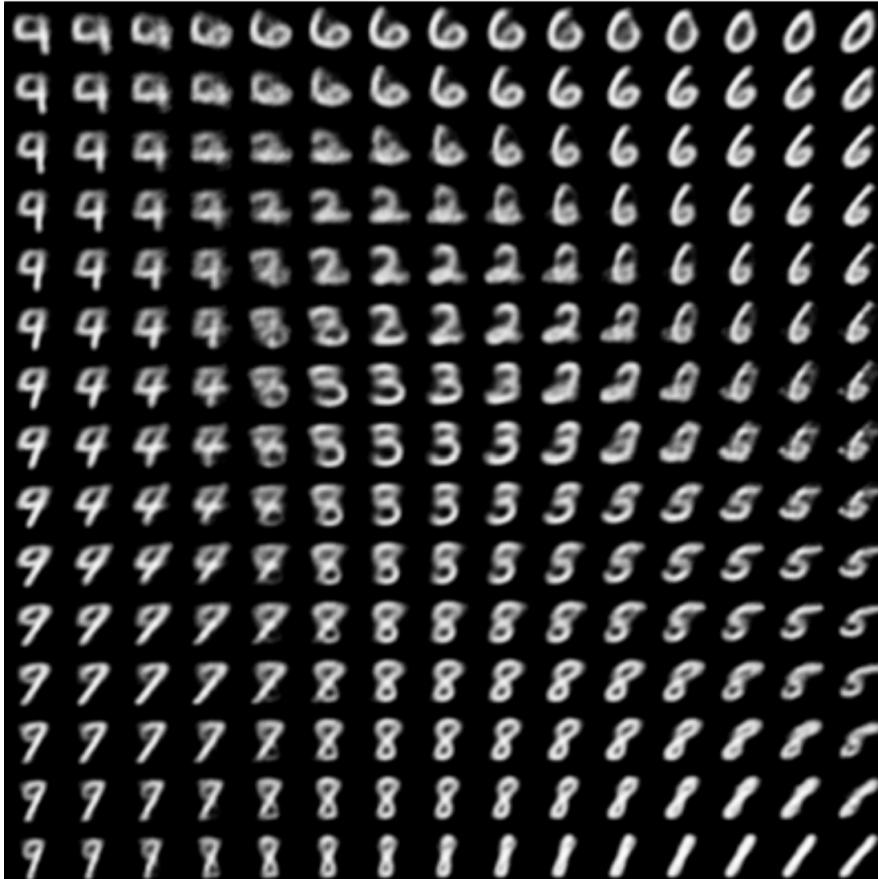
What we hope to learn.



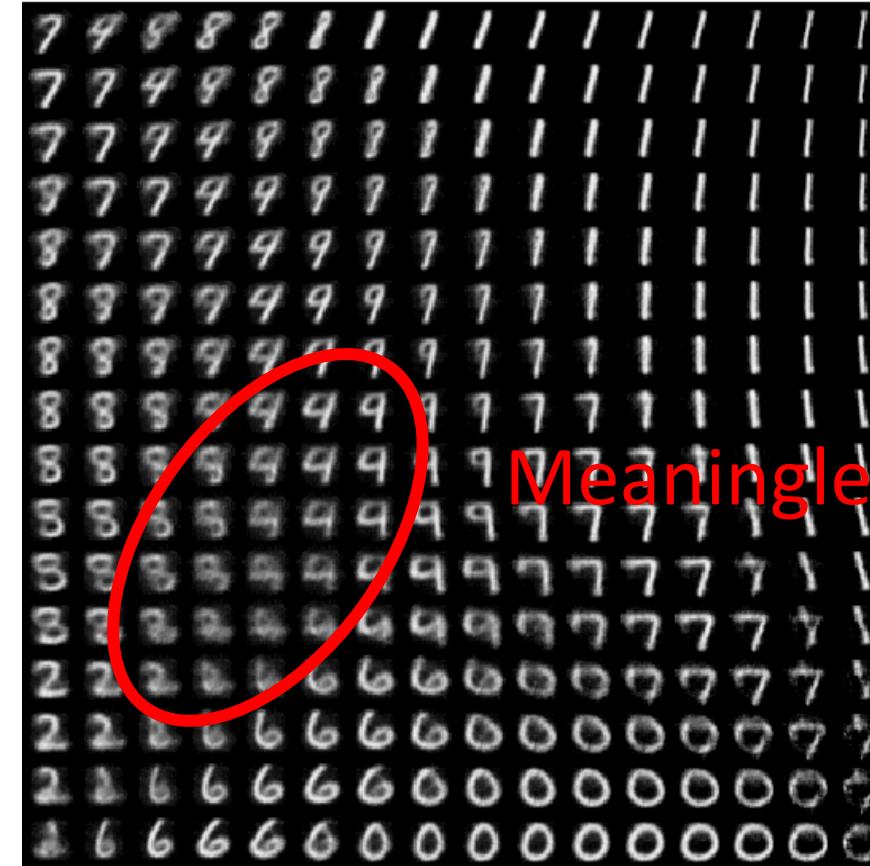
What we actually learn by
minimizing GenLoss.

A problem with generation loss

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



What we hope to learn.

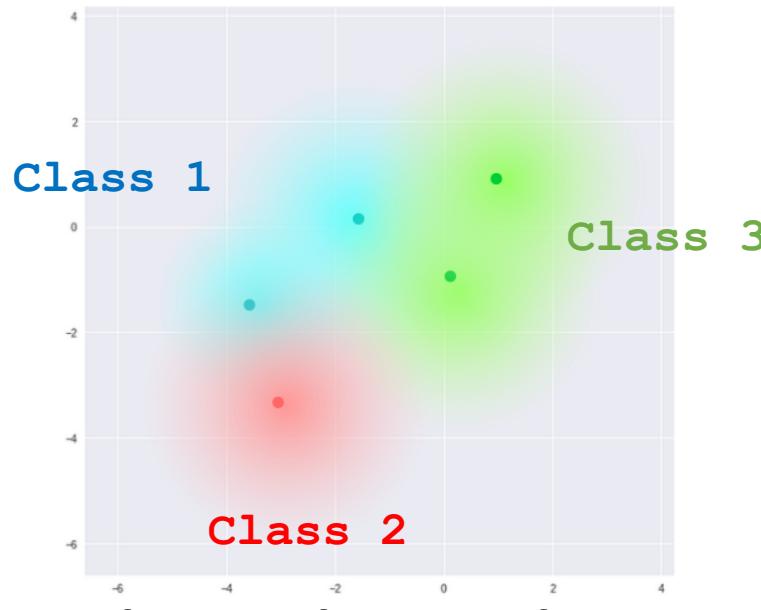


What we actually learn by
minimizing GenLoss.

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.

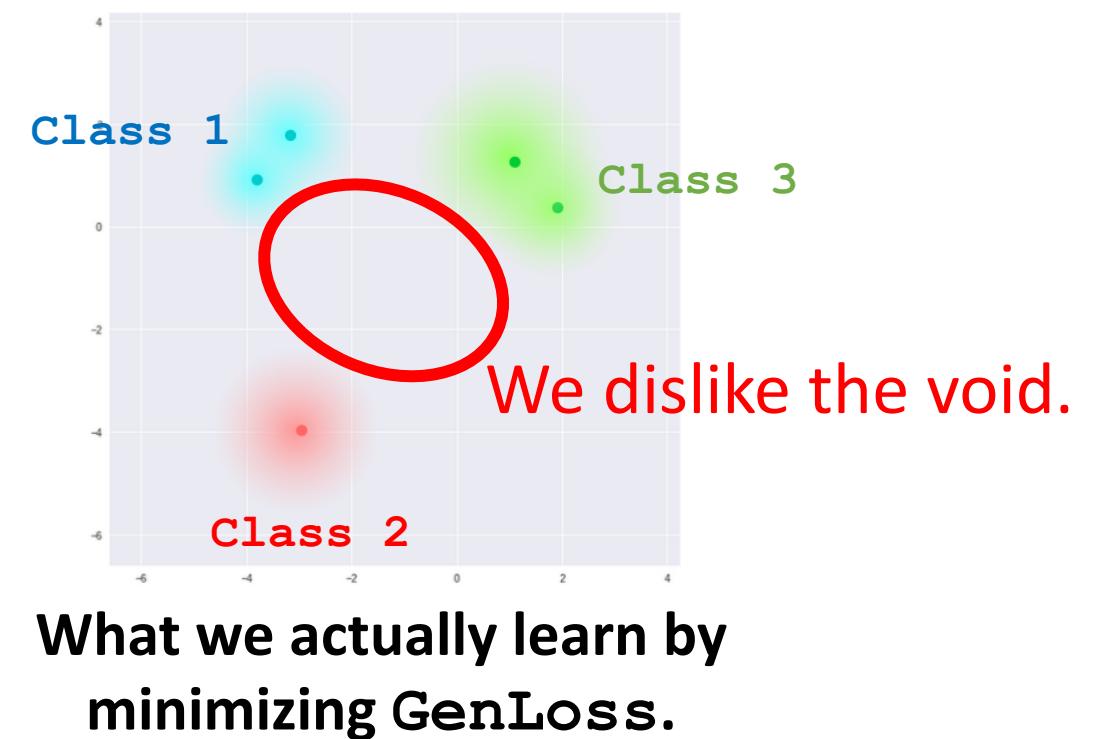
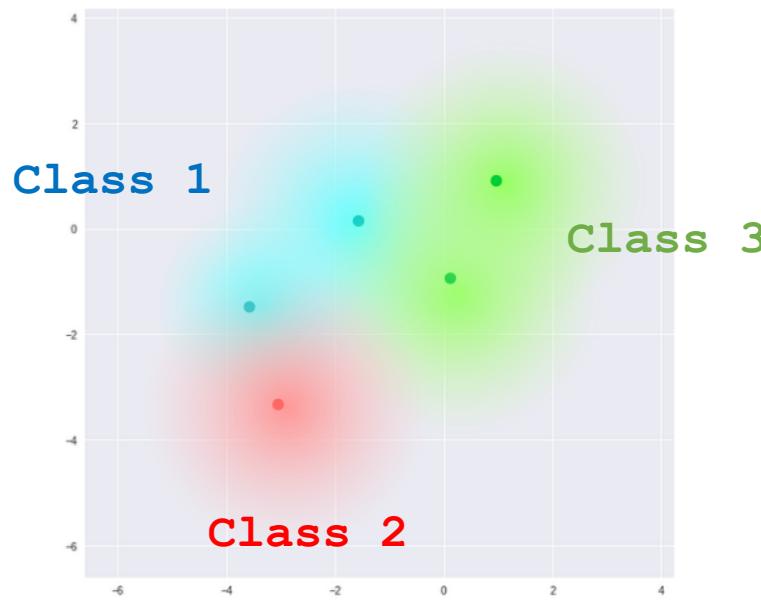


What we actually learn by
minimizing GenLoss.

KL LOSS

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

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



KL LOSS

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

- $\text{KLLoss} = D_{\text{KL}}(p(\mathbf{z}) \parallel \mathcal{N}(0, \mathbf{I}))$
- $\text{Loss} = \text{GenLoss} + \lambda \cdot \text{KLLoss}$

Generation Loss + KL Loss

- Generation loss:

GenLoss = dist(**input_img**, **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:

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

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

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)
```

- Latent loss:

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

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

- Loss = 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

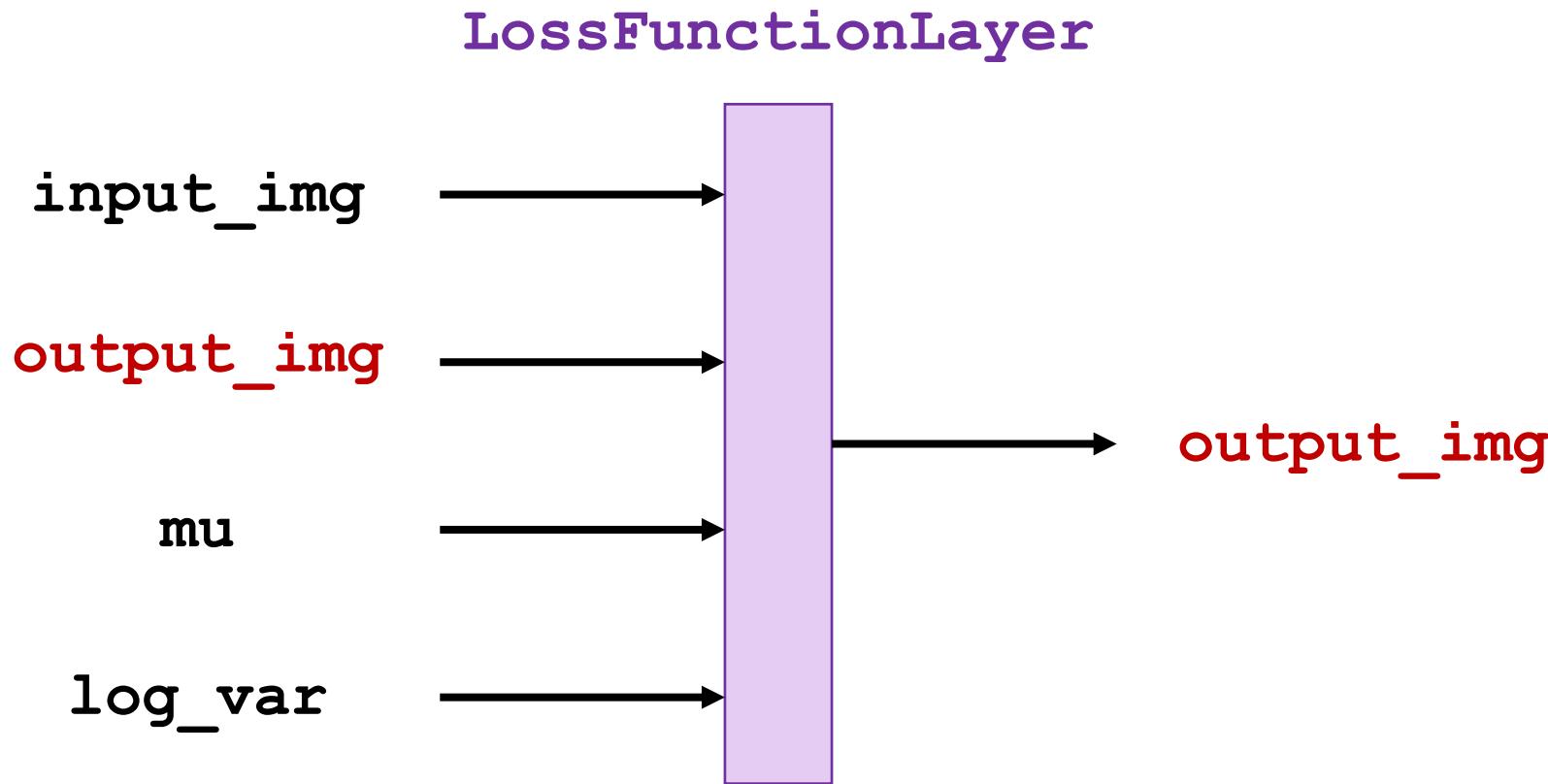
class LossFunctionLayer(keras.layers.Layer):
    param = 1E-3

    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

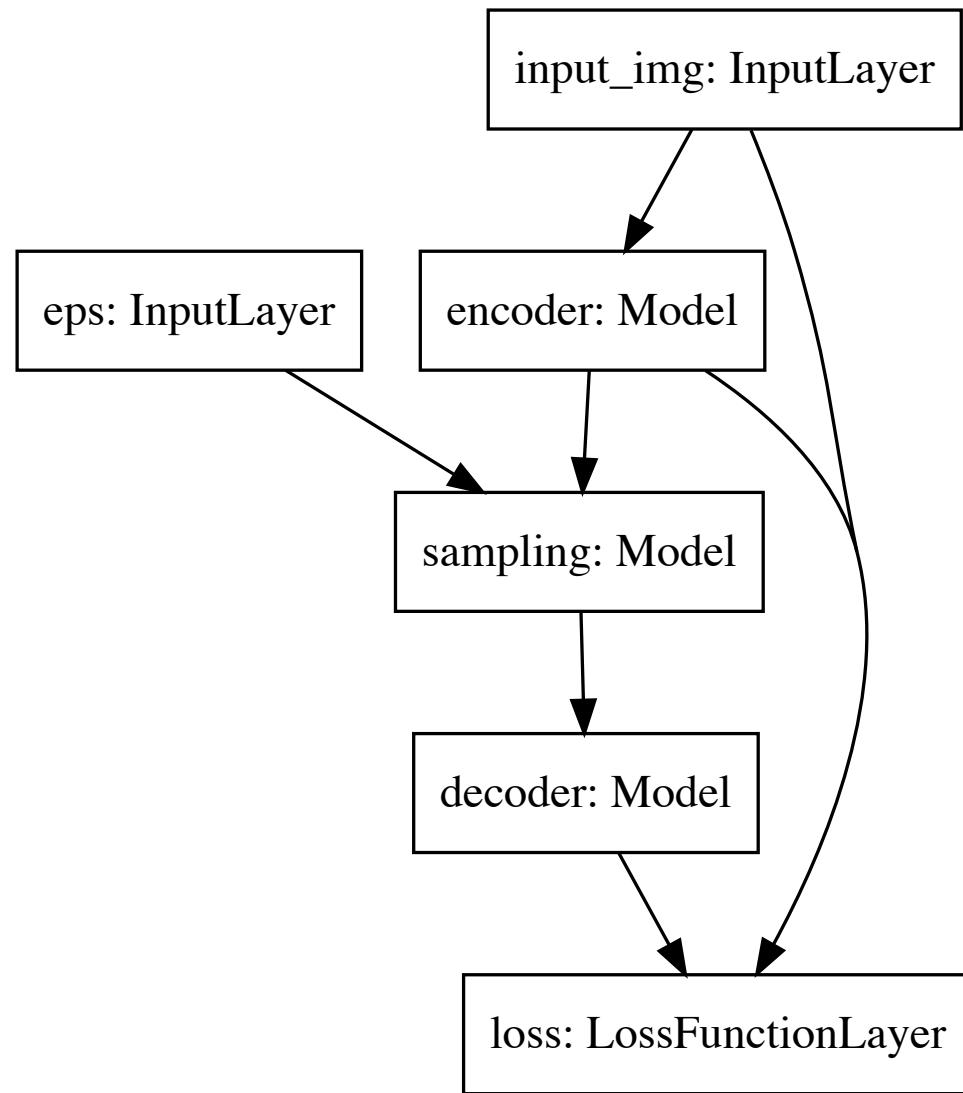
We have defined 4 modules:

- **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
```

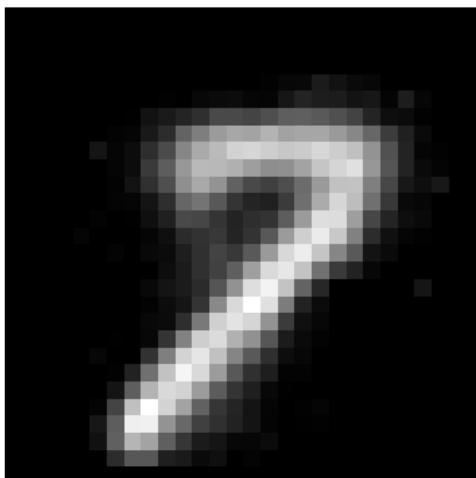
Visualize code vectors

arbitrary 2-dim vector

map the 2-dim vector to a 28×28 image

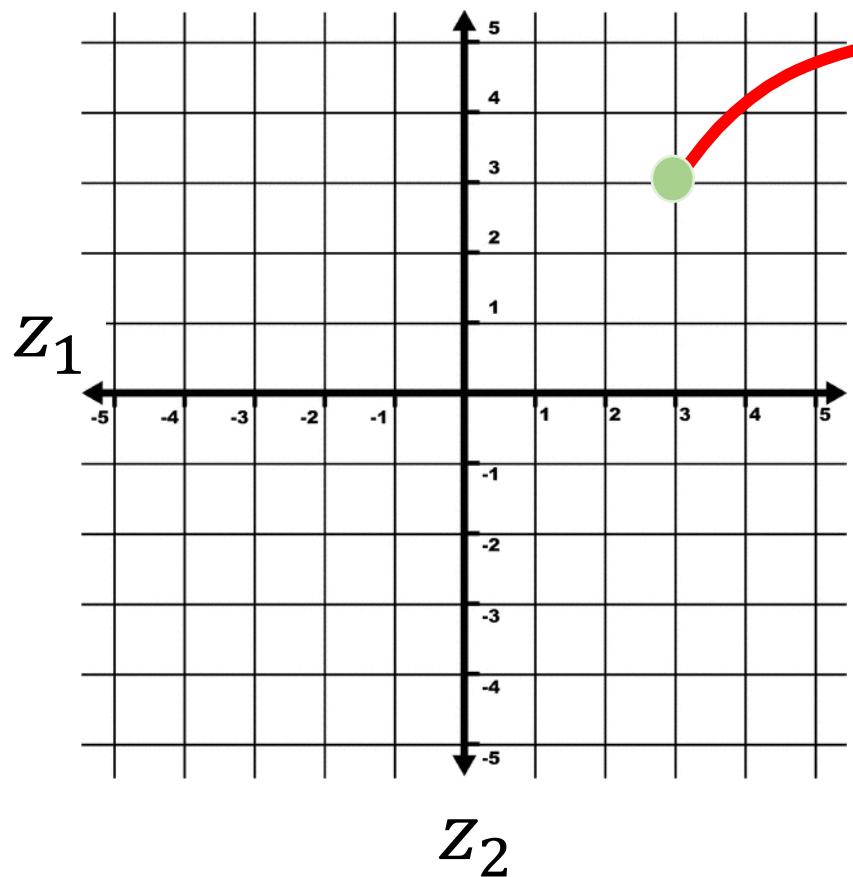
```
z_sample = np.array([0.1, 0.2]).reshape((1, 2))
x_decoded = decoder.predict(z_sample)[0].reshape((28, 28))
```

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

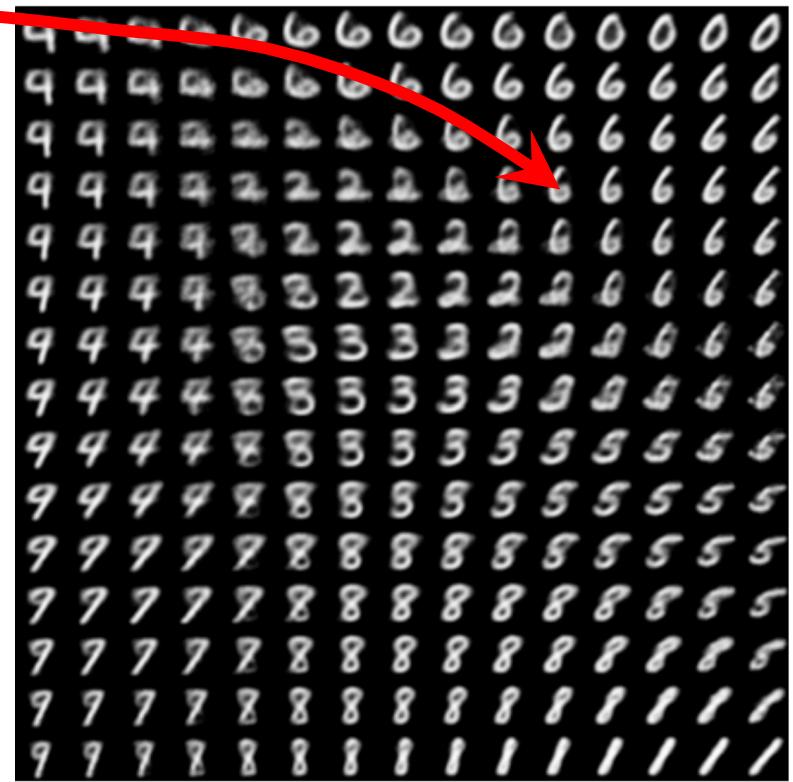


Visualize code vectors

Code vectors



Images



Summary

Variational Autoencoder (VAE)

- VAE = AE + probability tricks.
- The decoder network is (almost) continuous function.

Variational Autoencoder (VAE)

- VAE = AE + probability tricks.
- The decoder network is (almost) continuous function.
- Loss = Generation Loss + $\lambda \cdot$ KL Loss.
- Application: edit images in the latent space.
 - Average faces.
 - Add smile.