Variational Autoencoder for Image Generation

Shusen Wang

Preliminary: Distance between Two Probability Distributions

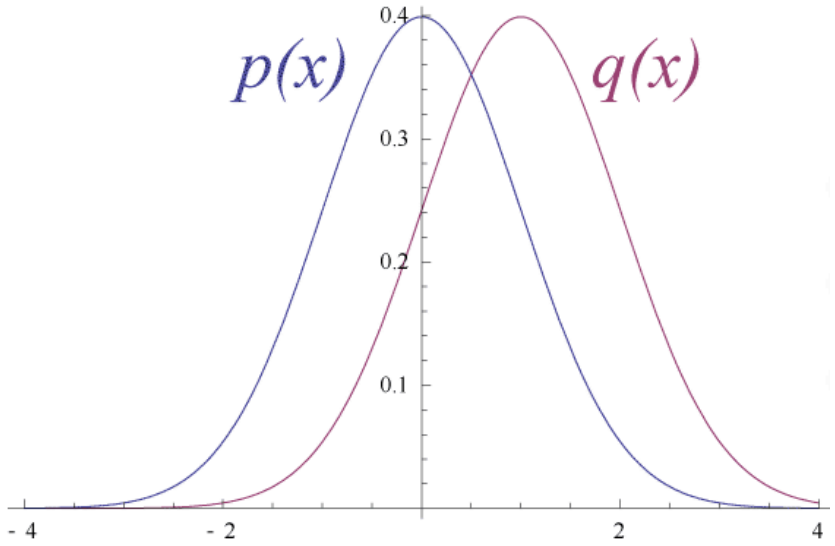
Distance between Two Vectors

- Distance between two vectors is measured by vector norms.
- Examples:
  - ℓ_1 distance: $\left|\left|\mathbf{a}-\mathbf{b}\right|\right|_1 = \sum_j \left|a_j - b_j\right|$.
  - 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

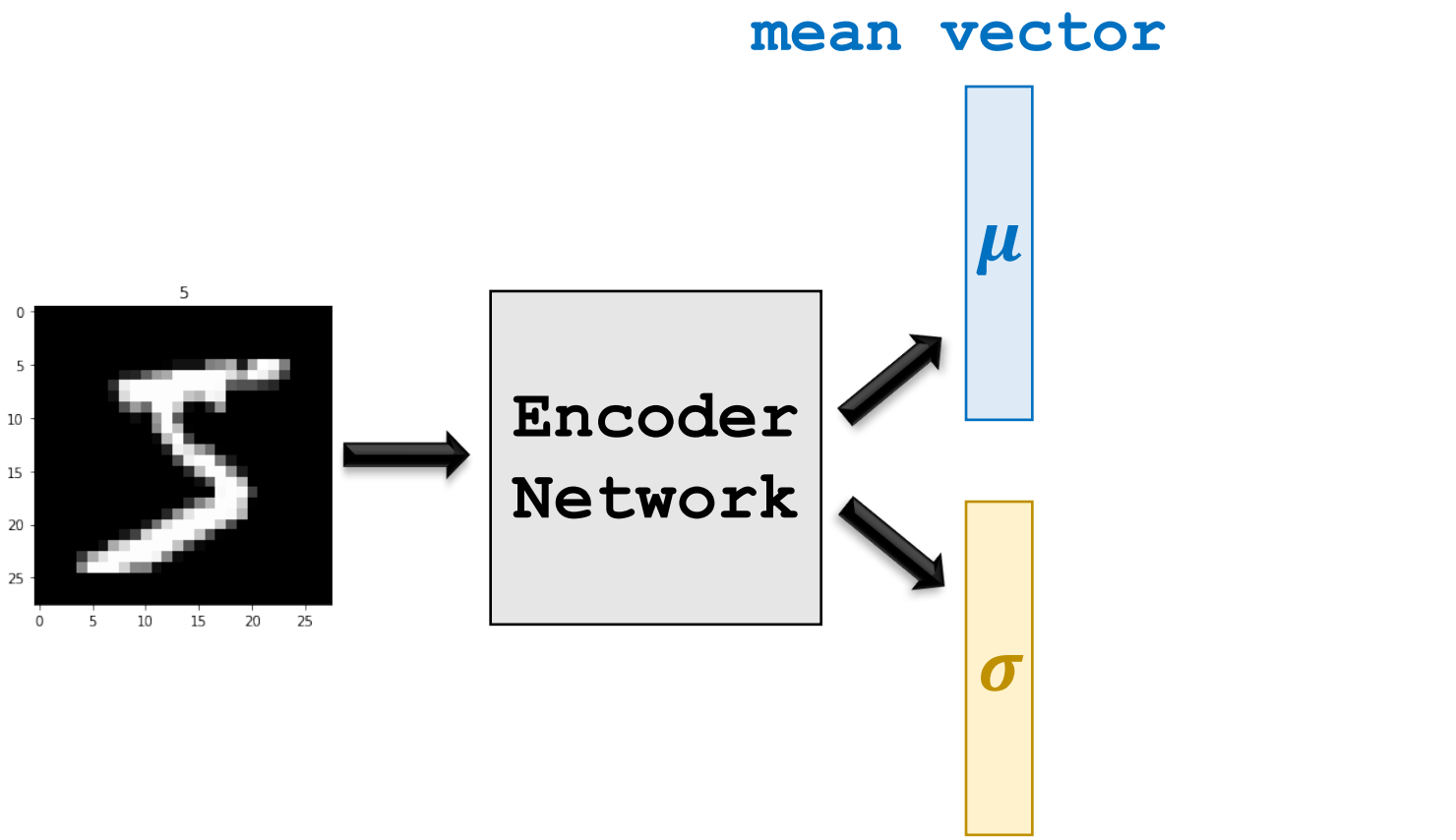
$$D_{\mathrm{KL}}(p||q) = -\sum_i p(i) \log \frac{q(i)}{p(i)}.$$

- For continuous probability distributions, the KL divergence is

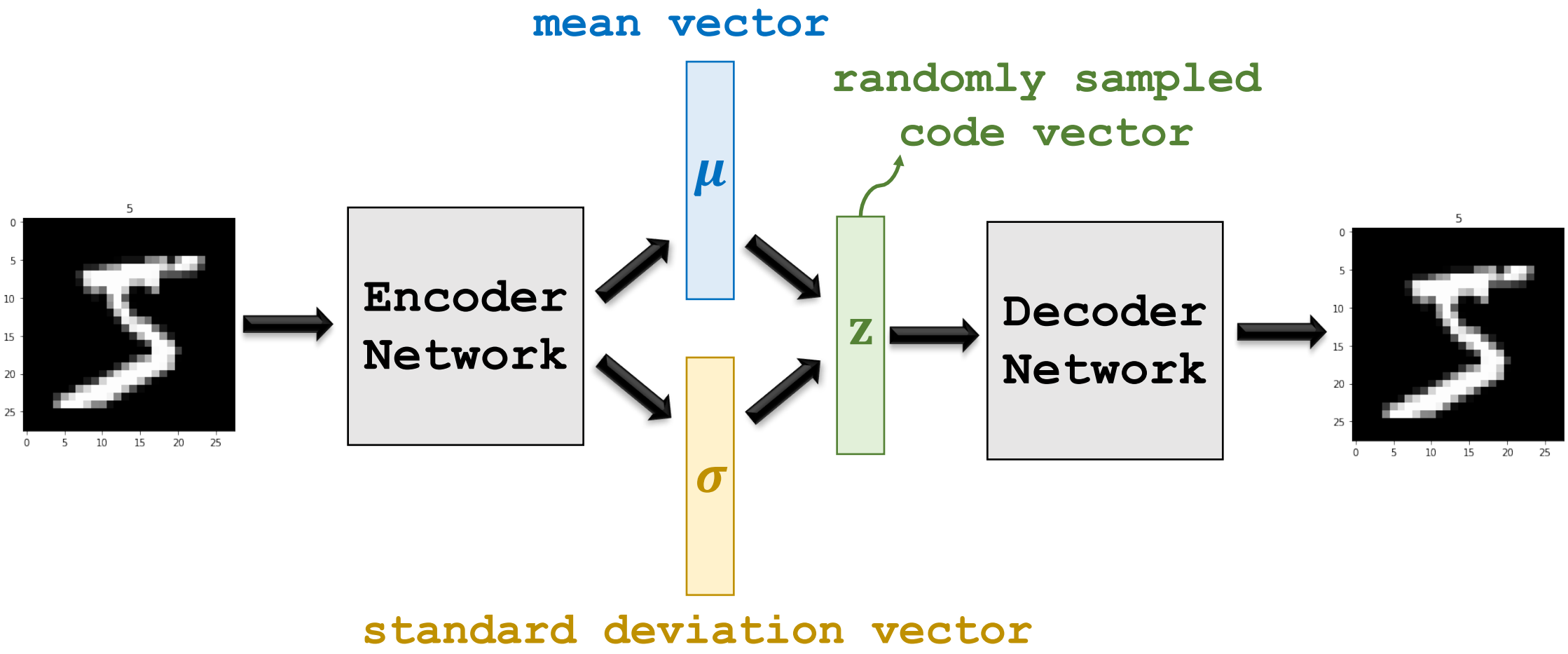
$$D_{\mathrm{KL}}(p||q) = -\int_{-\infty}^{+\infty} p(x) \log \frac{q(x)}{p(x)}\, dx.$$

Variational Autoencoder

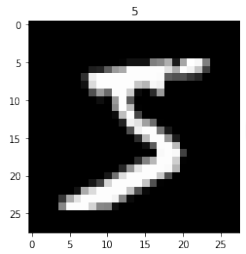
Variational Autoencoder (VAE)



Variational Autoencoder (VAE)



VAE Sampling

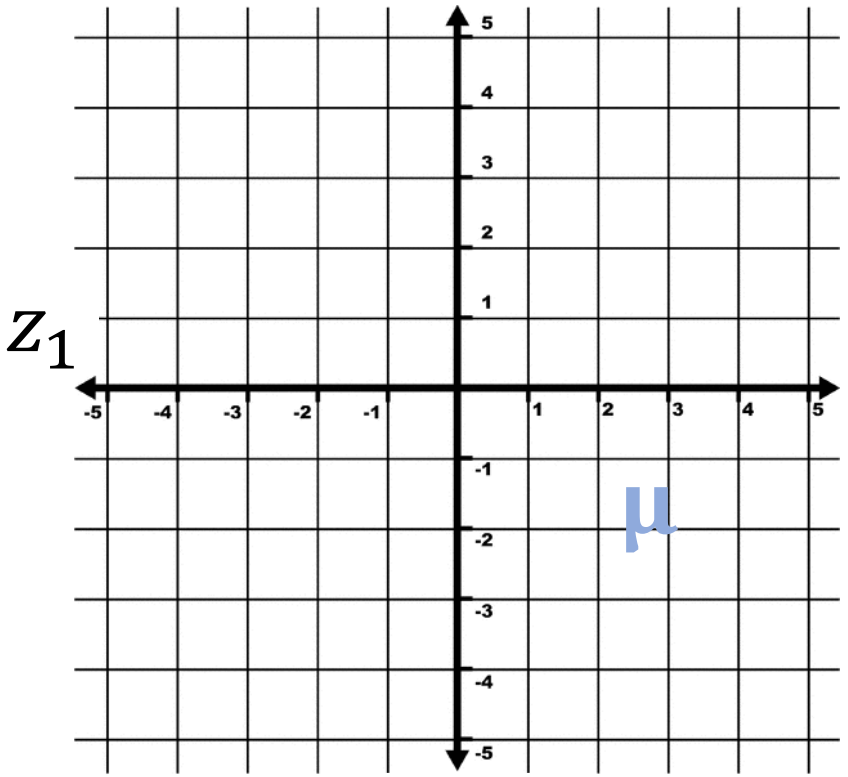


Encoder Network

mean vector

μ

Code vectors

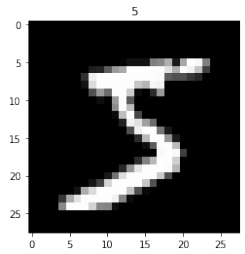


z_1

z_2

VAE Sampling

Code vectors



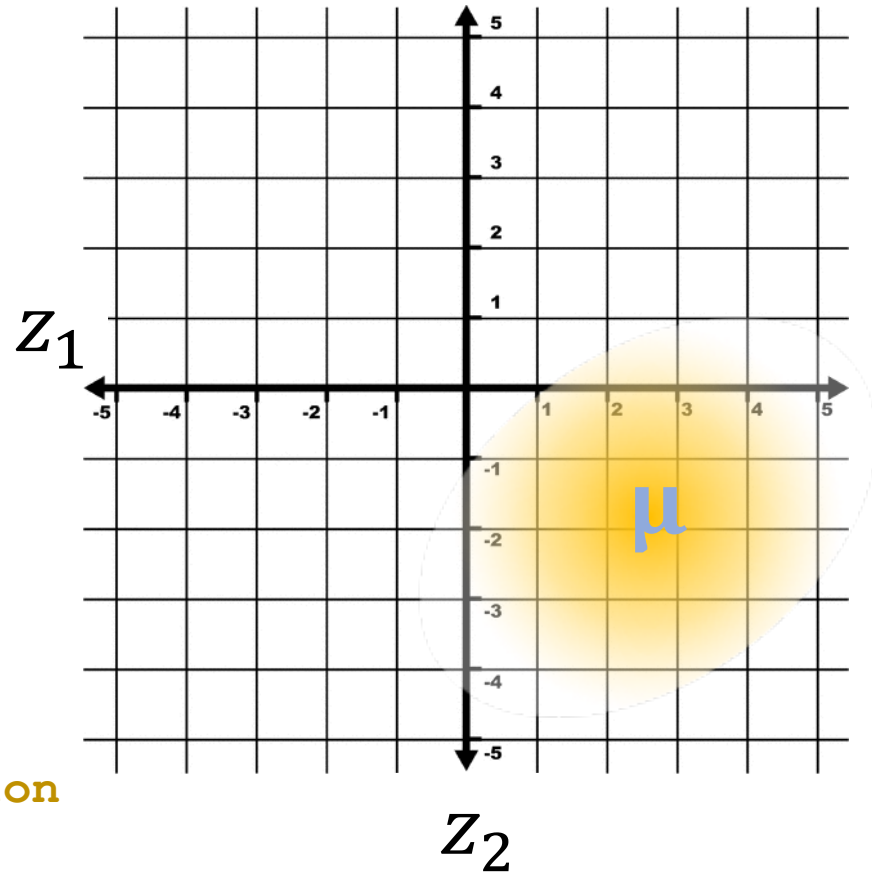
Encoder Network

mean vector

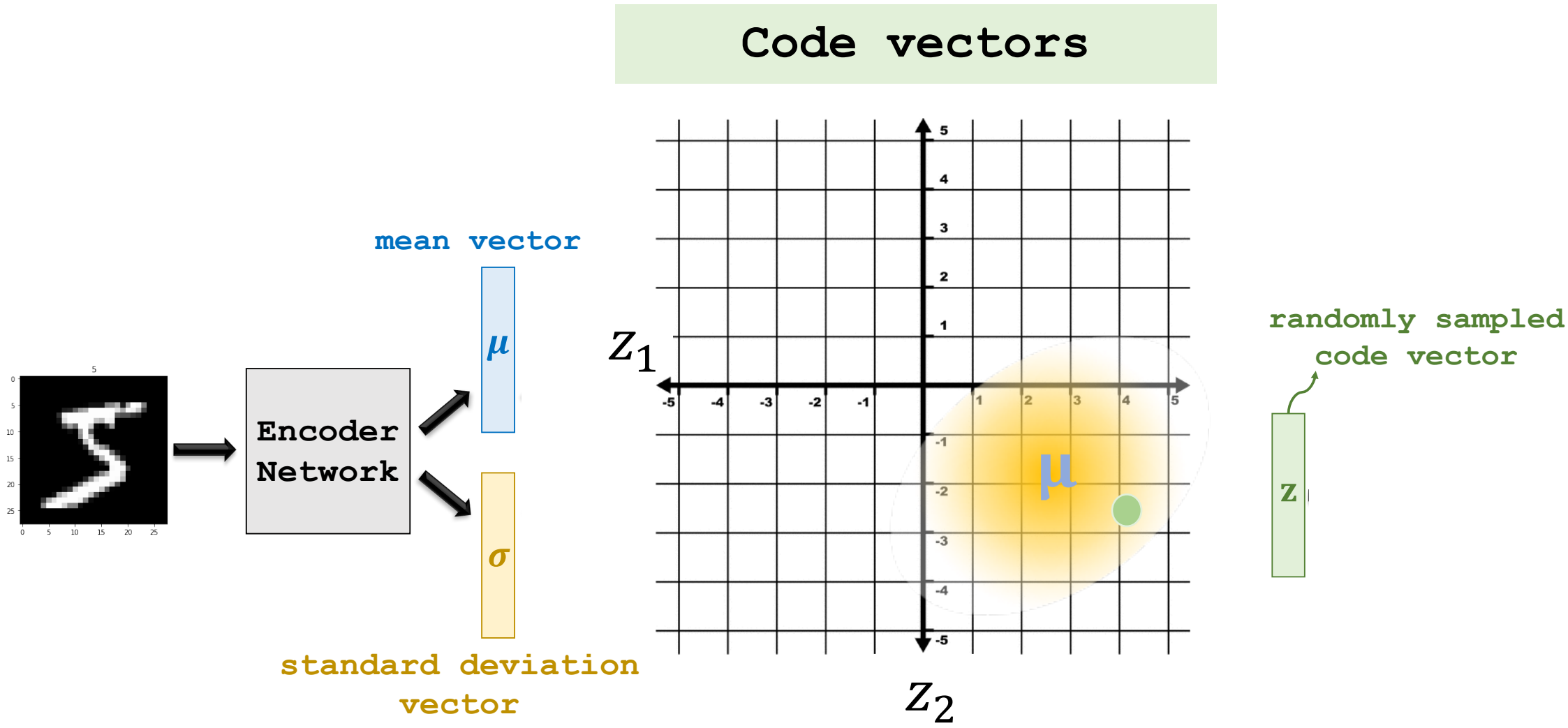
μ

σ

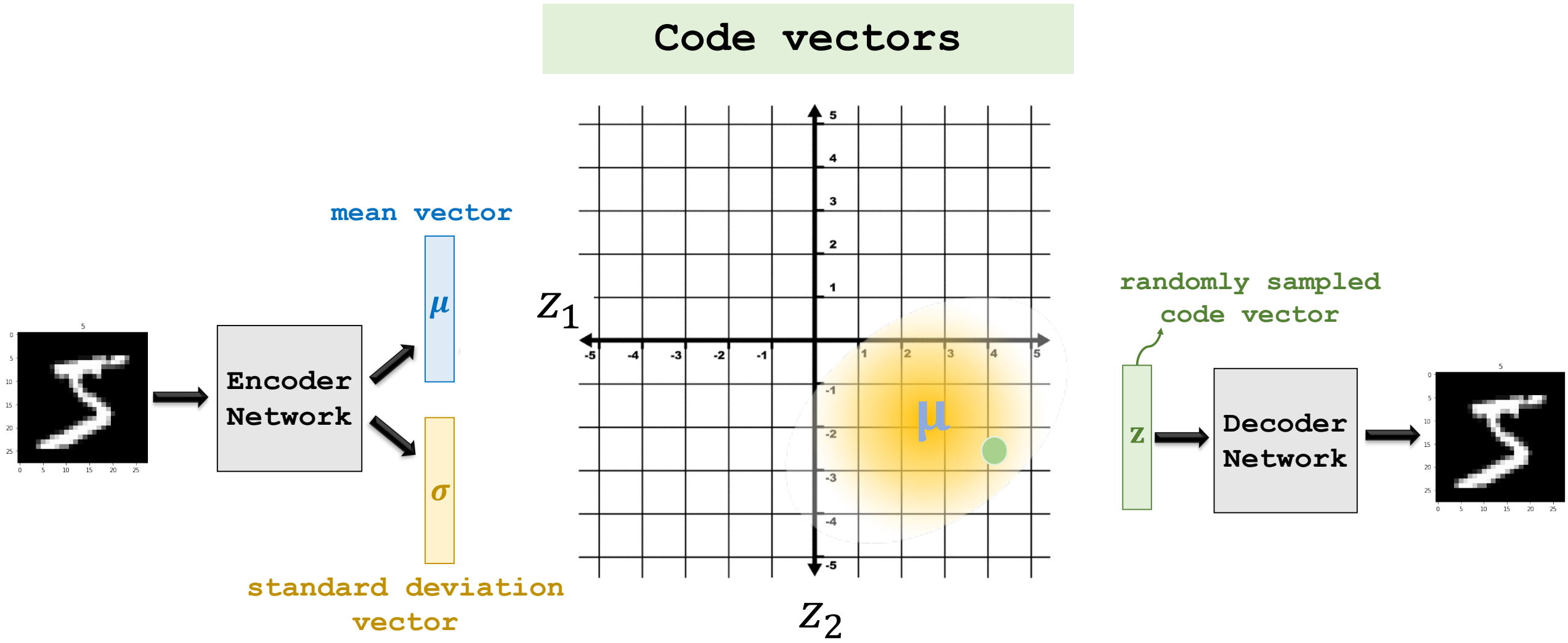
standard deviation vector



VAE Sampling



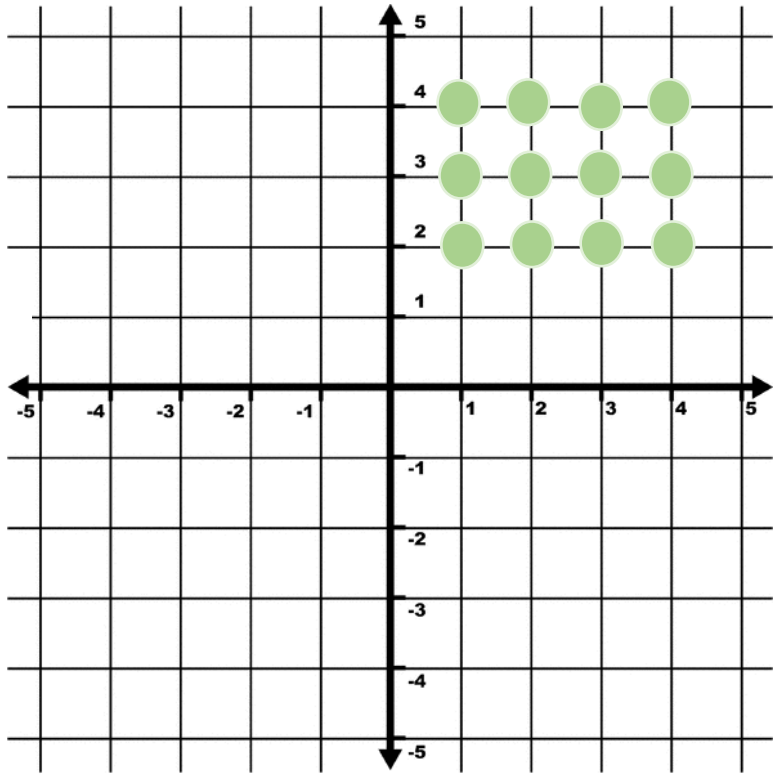
VAE Sampling



Visualize Code Vectors

Visualize code vectors.

Code vectors

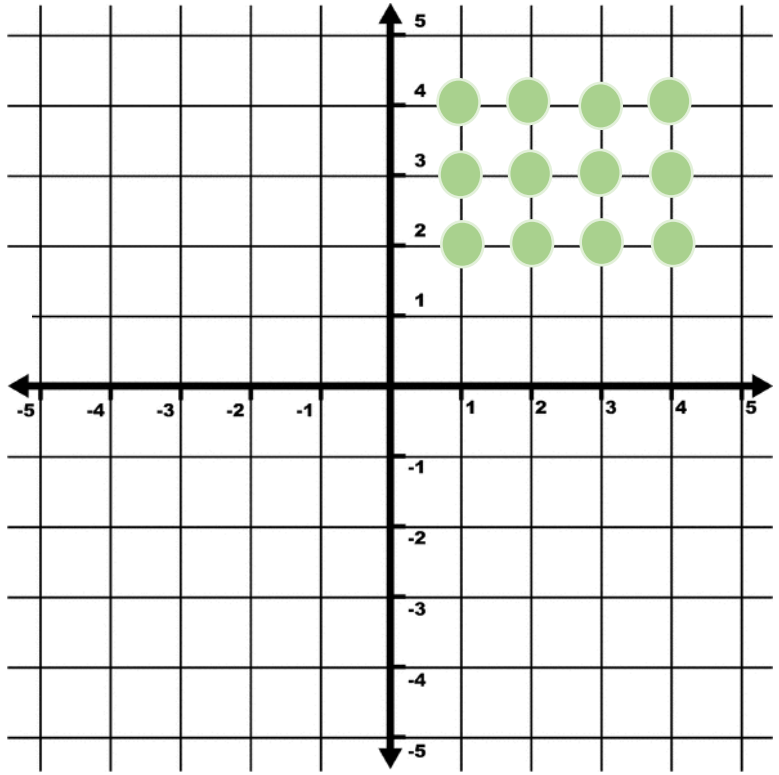


- Get a set of code vectors from the grid:

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

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:

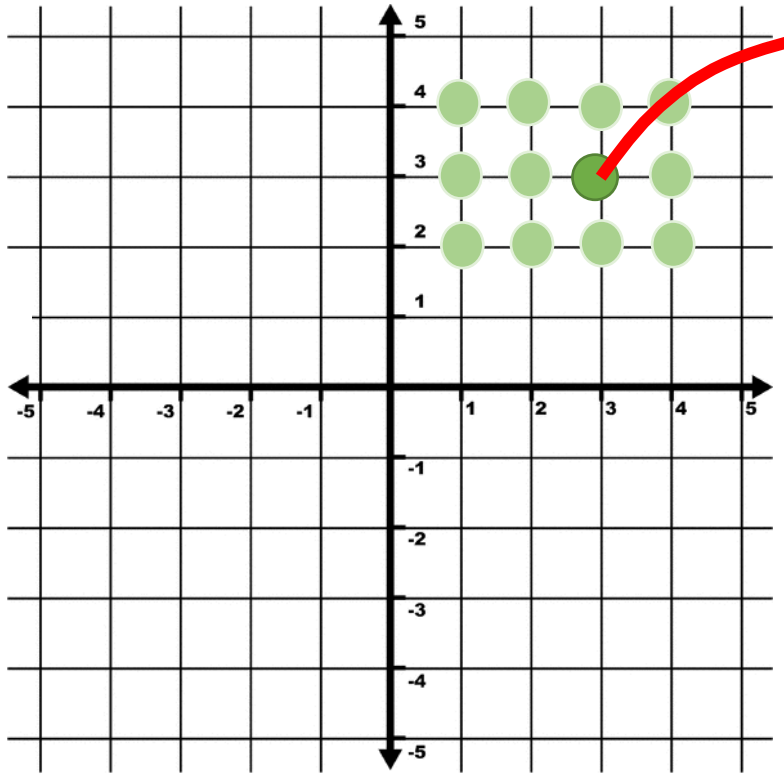
$$\text{image}_i = \text{Decoder}(\mathbf{z}_i).$$

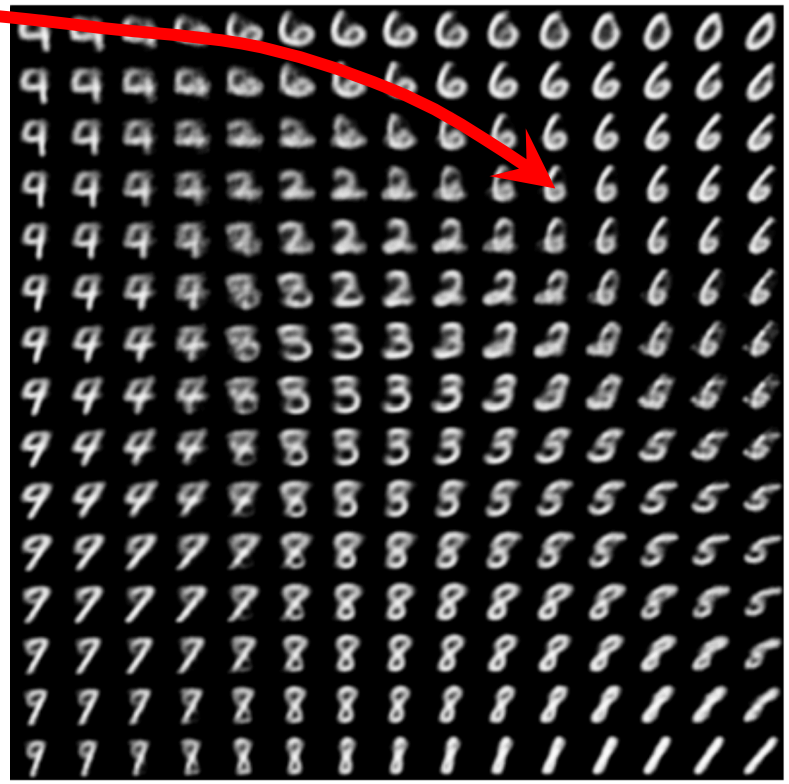
Visualize code vectors.

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

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

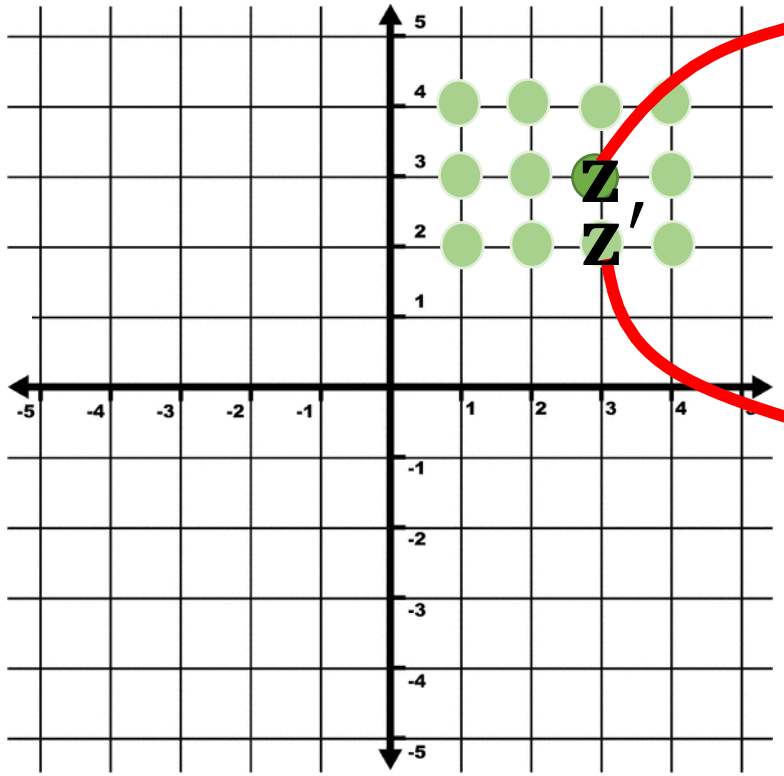
  are similar as well.

The decoder behaves like a continuous function.

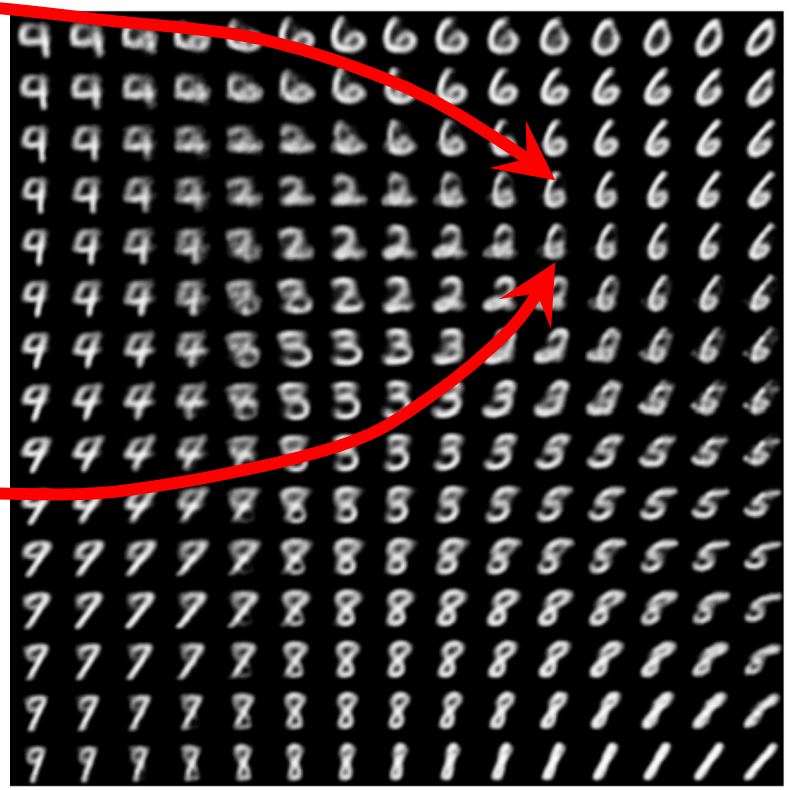
Code vectors

Images

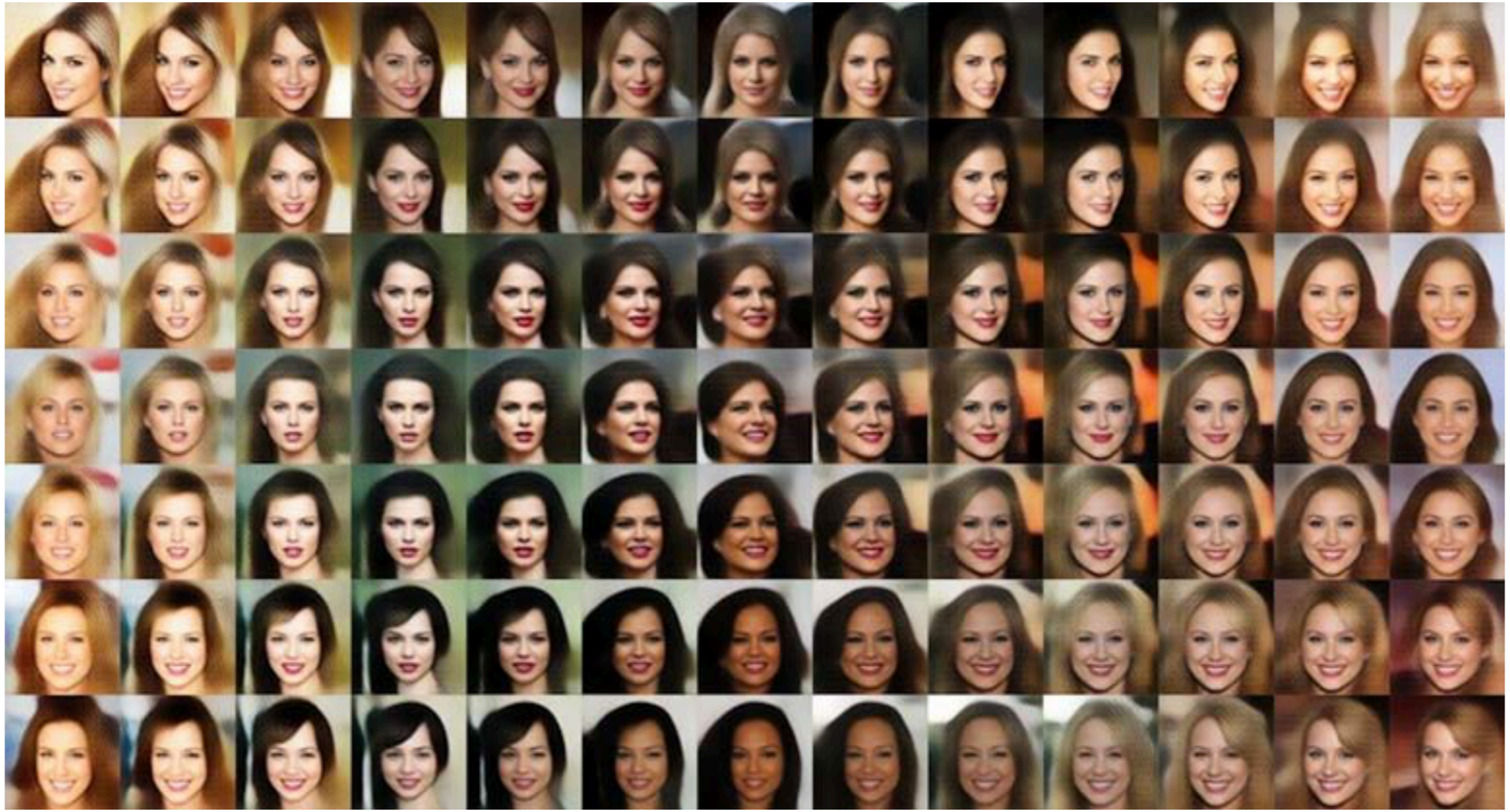
Decoder



Decoder

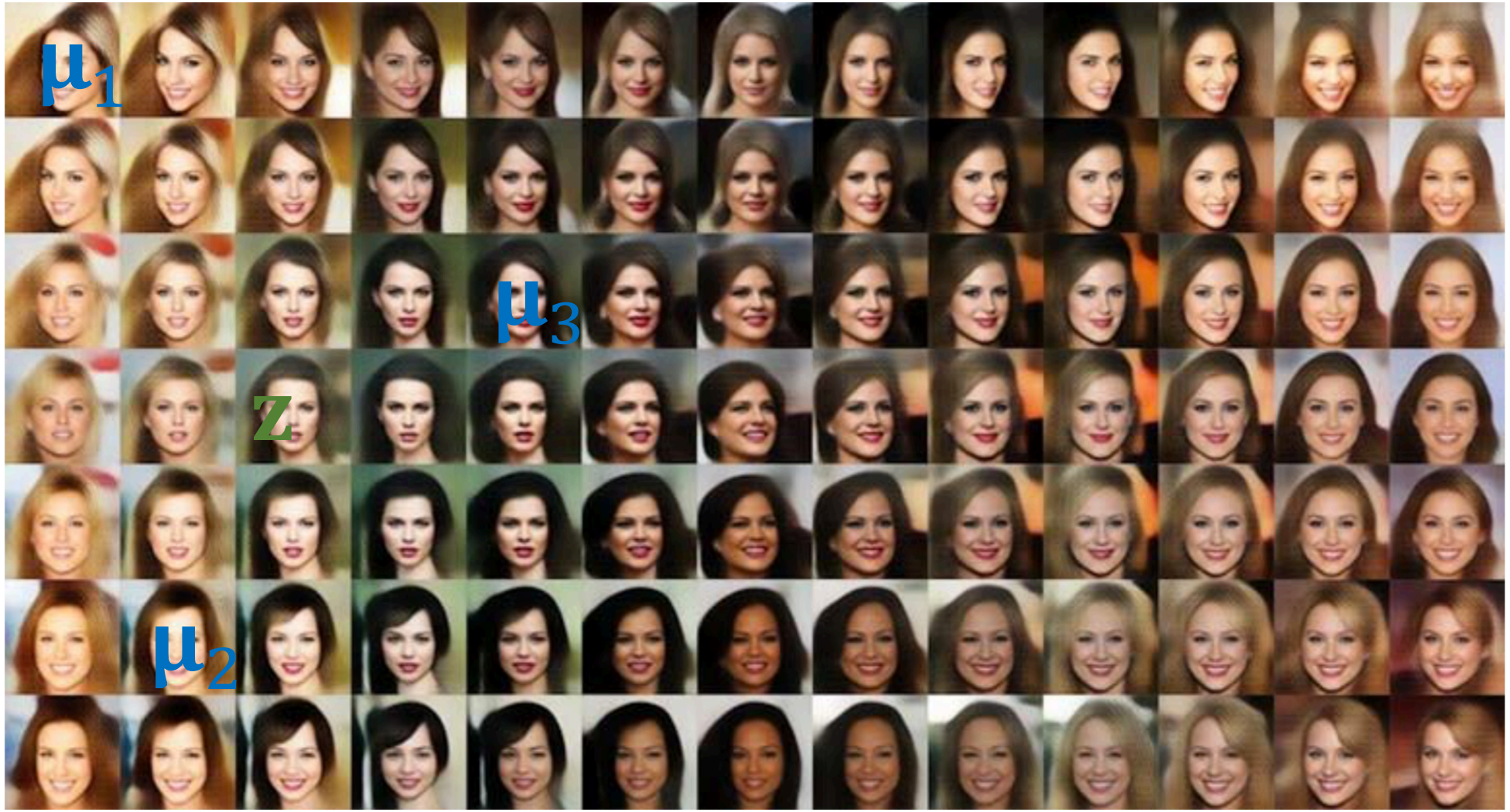


The decoder behaves like a continuous function.



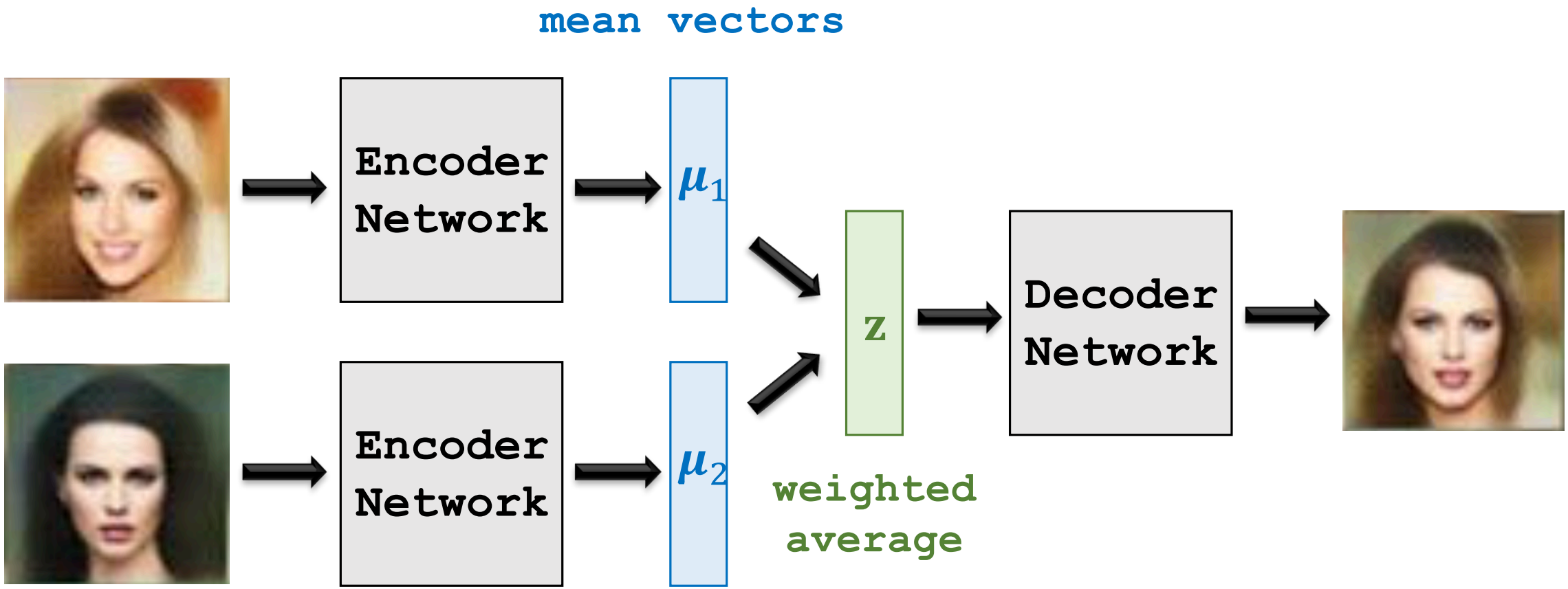
Visualization of the generated faces (by Tom White)

Averaging images via averaging code vectors

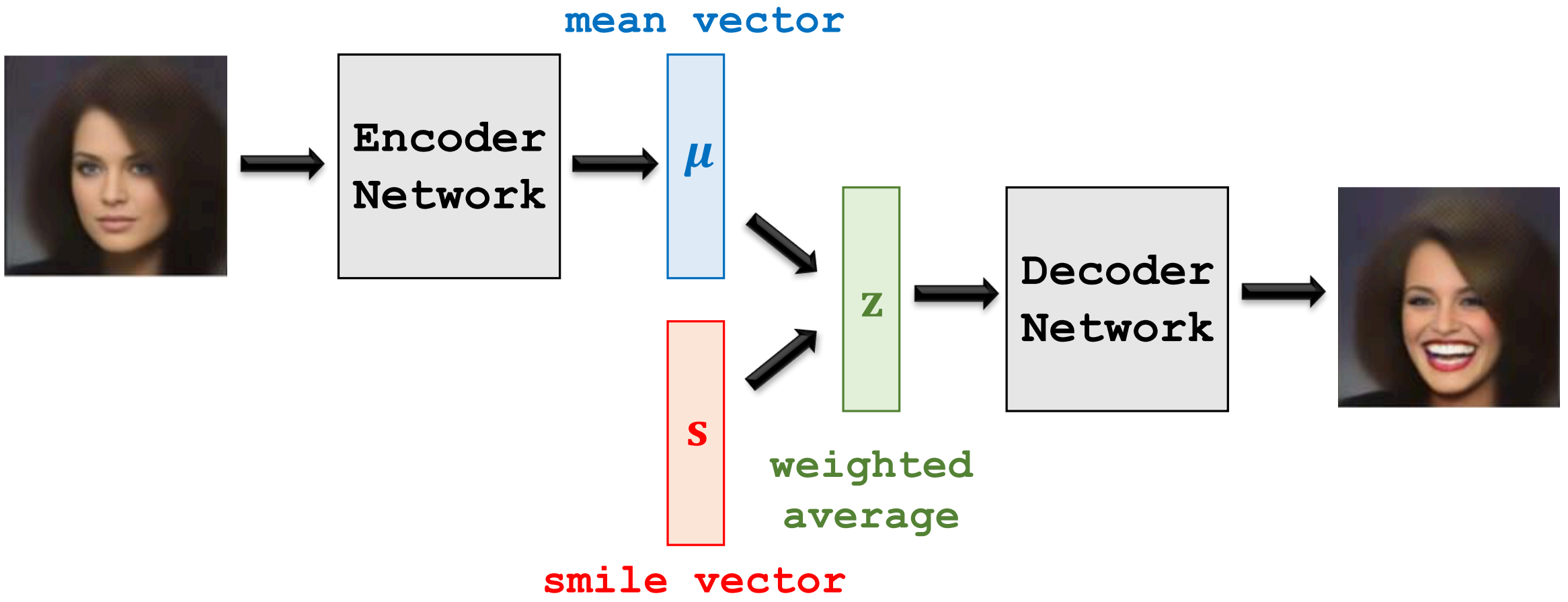


Visualization of the generated faces (by Tom White)

Averaging images via averaging code vectors

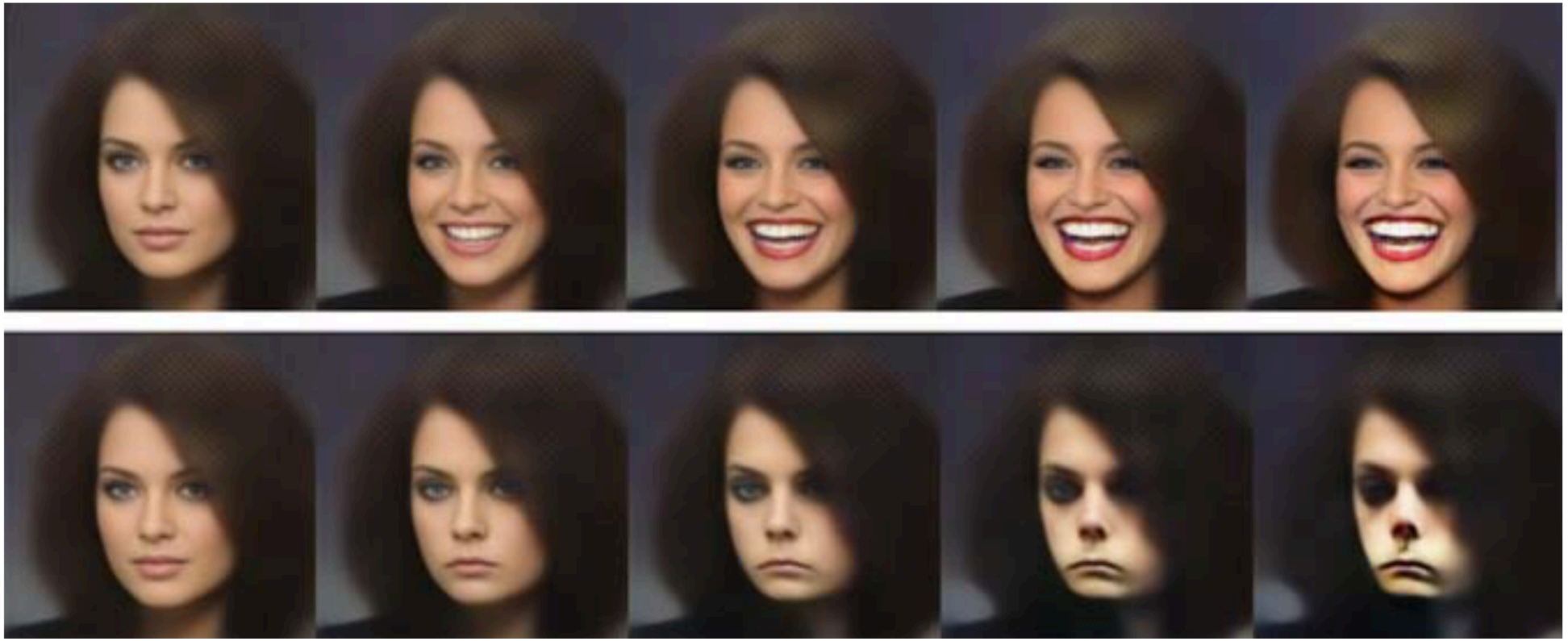


Editing images via editing code vectors



Editing images via editing code vectors

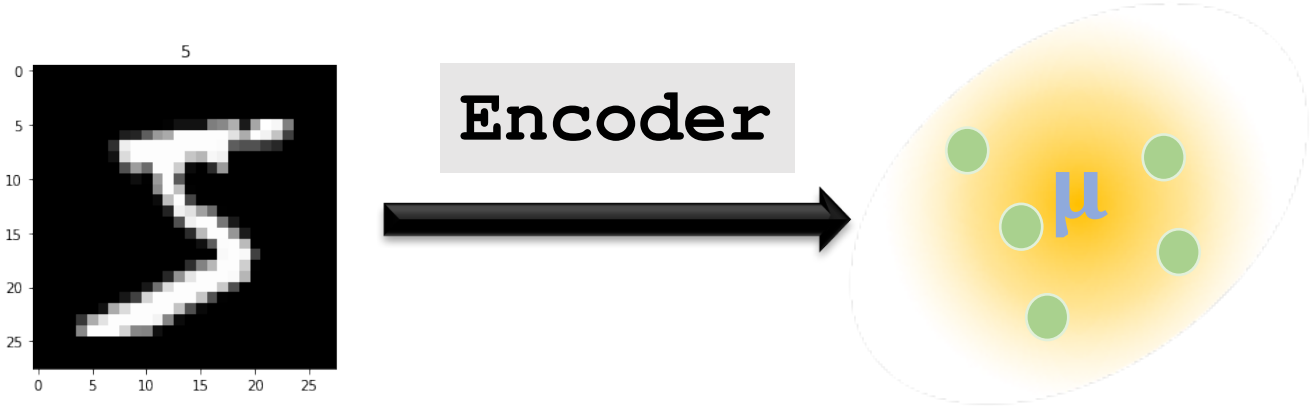
μ $\mu + s$ $\mu + 2s$ $\mu + 3s$ $\mu + 4s$



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

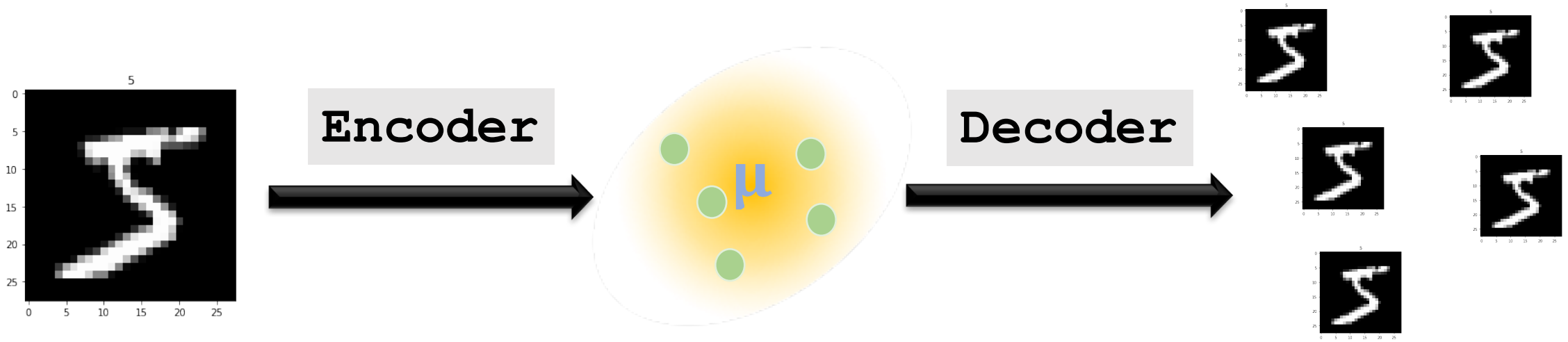
What makes the decoder continuous?

- 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 $i = 1, 2, 3, \cdots$.



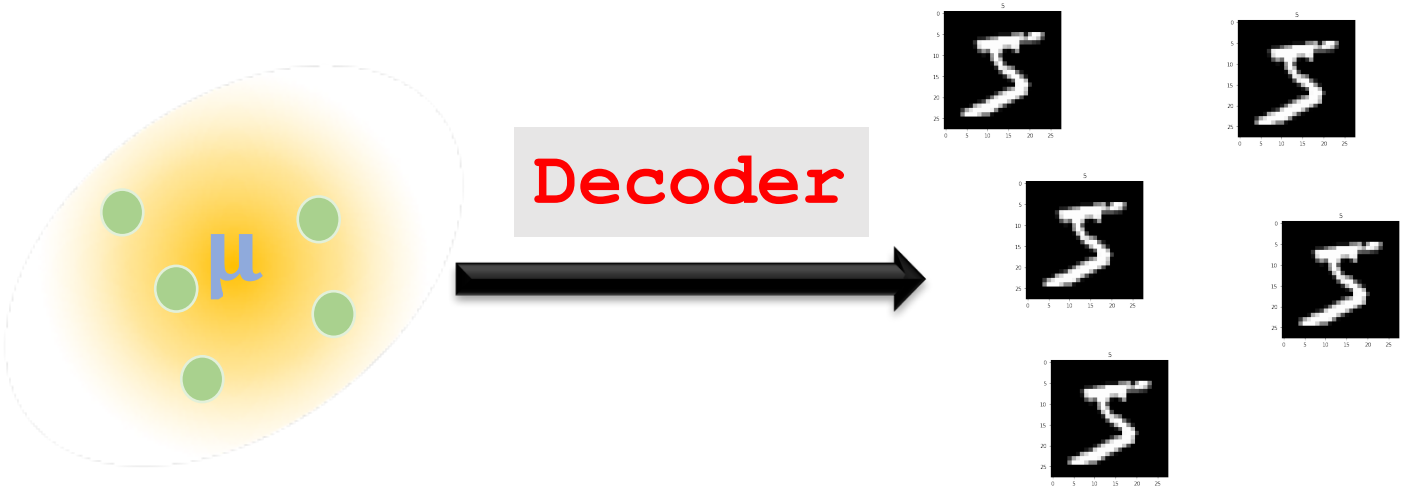
What makes the decoder continuous?

- 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).



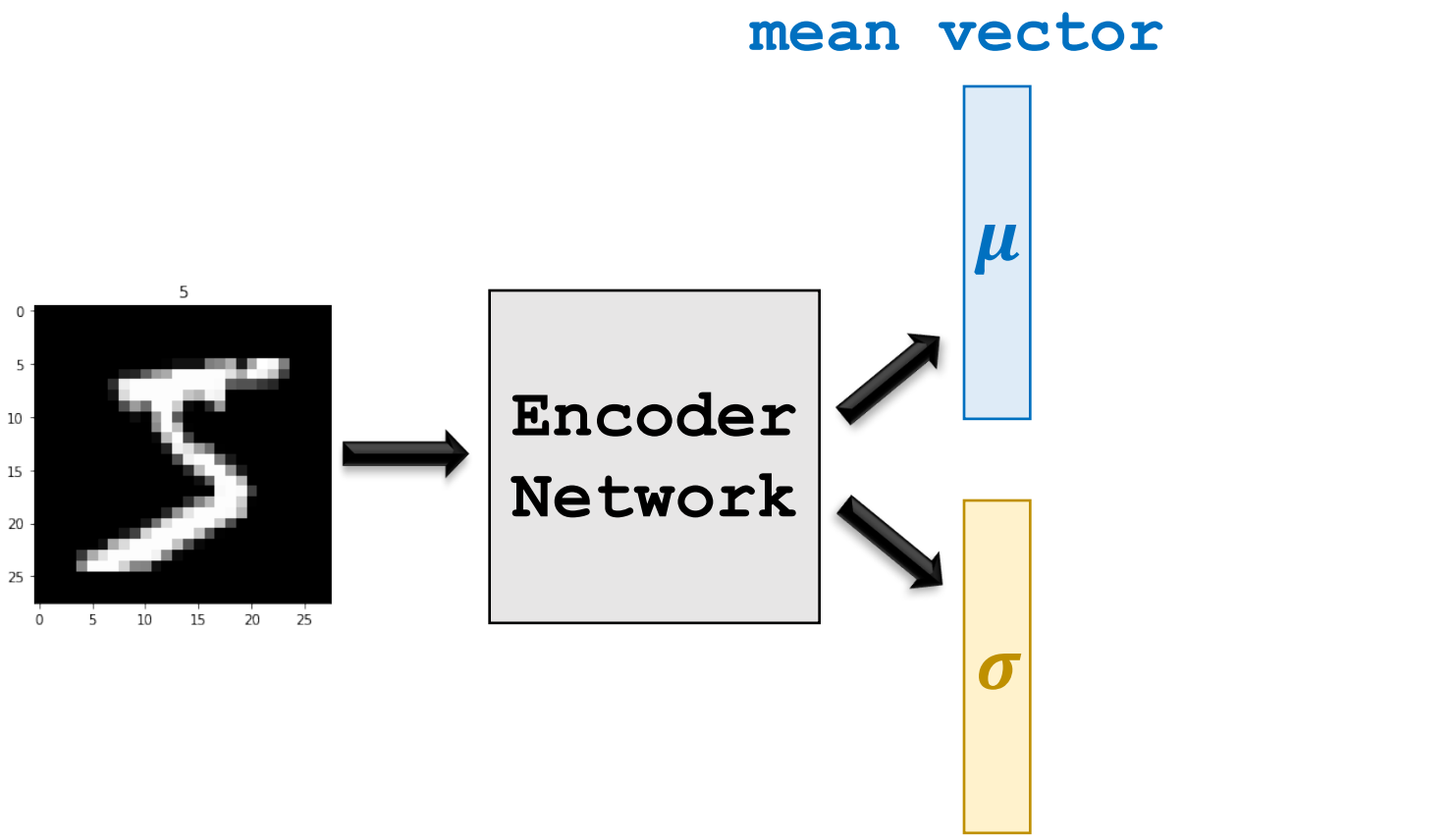
What makes the decoder continuous?

- 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.



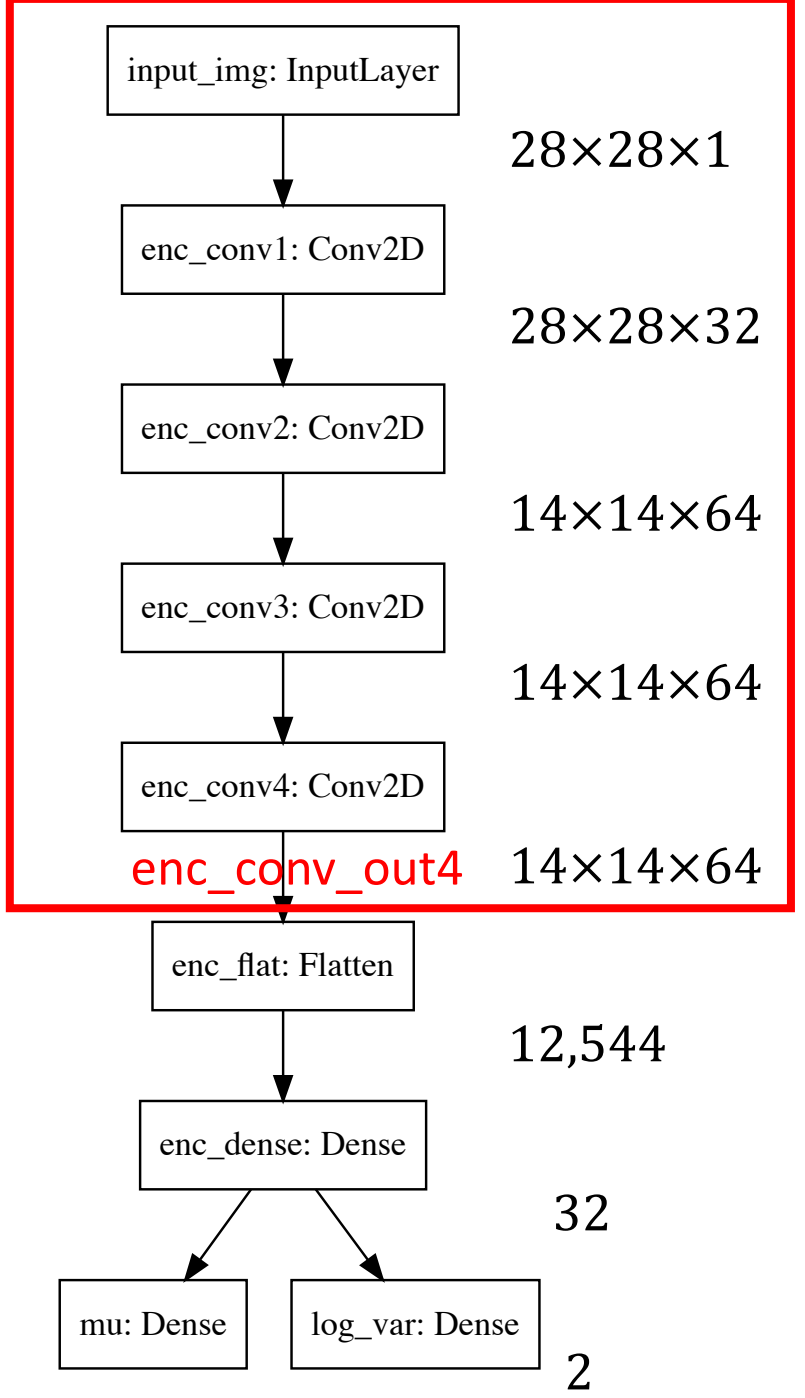
Build the Networks

1. The Encoder Network



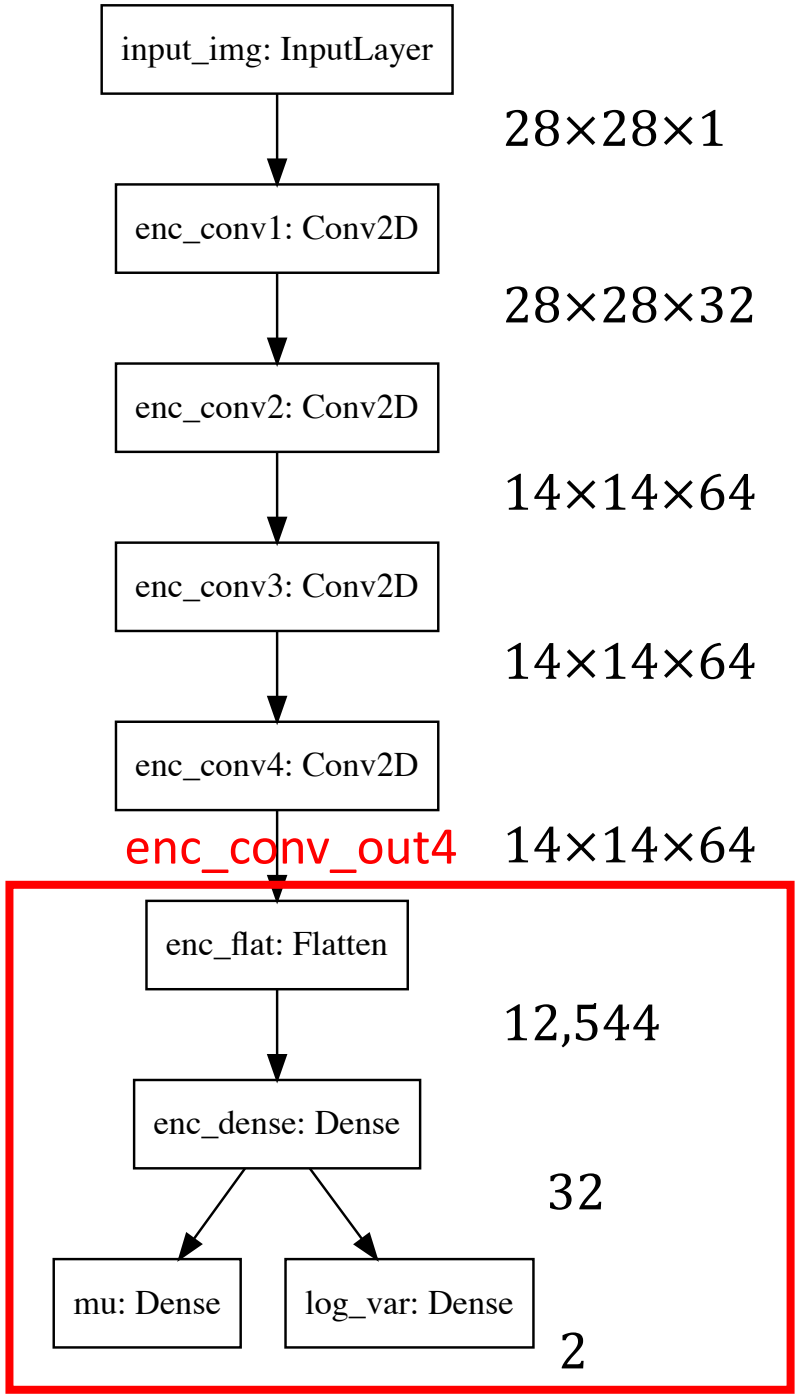
1. The Encoder Network

```
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)
```



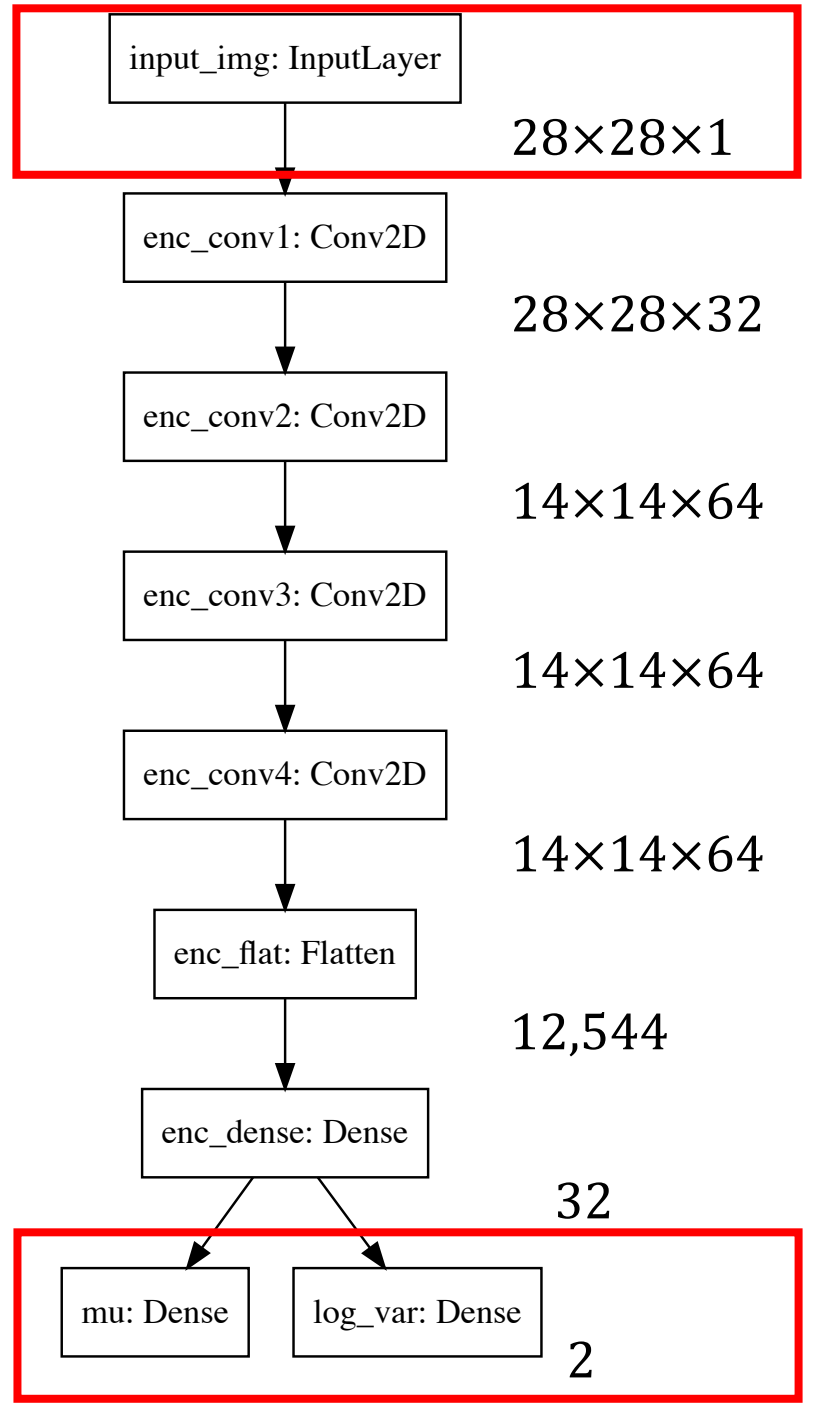
1. The Encoder Network

```
# 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)
```

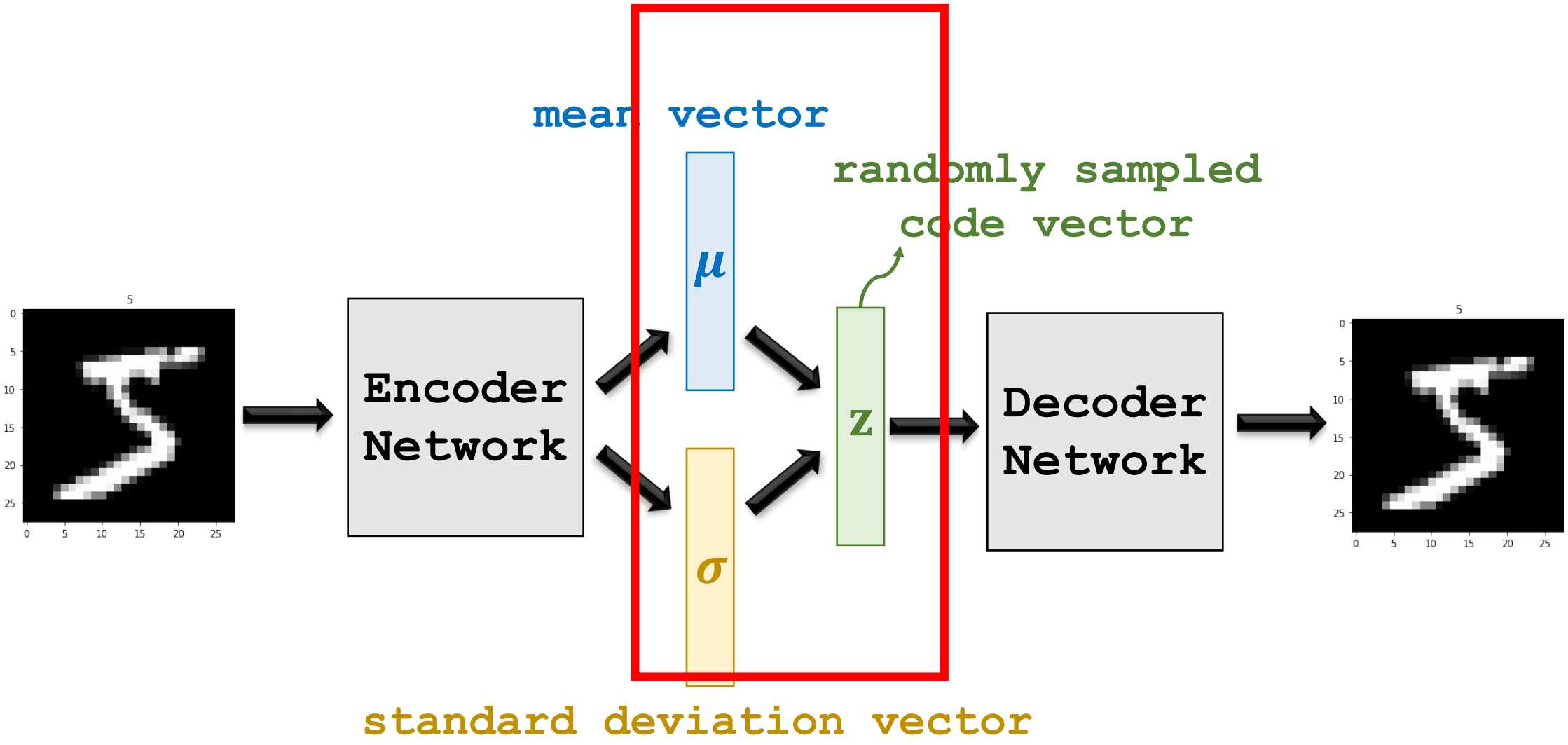


1. The Encoder Network

```
# 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

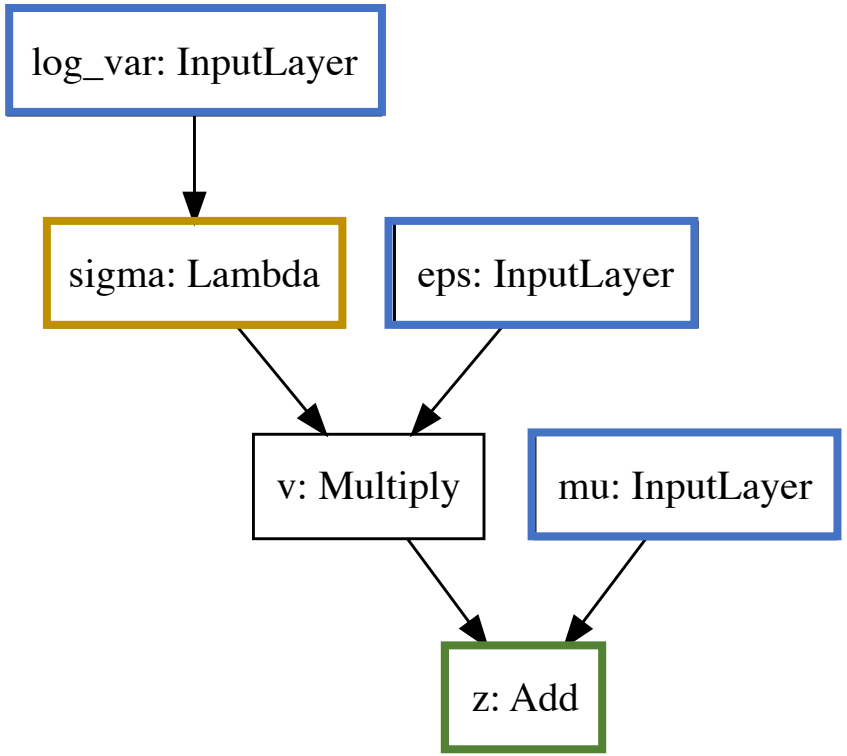


2. The Sampling Network

- The i-th entry 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 $\mathcal{N}\left(0, \sigma_i^2\right)$.)
  - $z_i = \mu_i + v_i$.

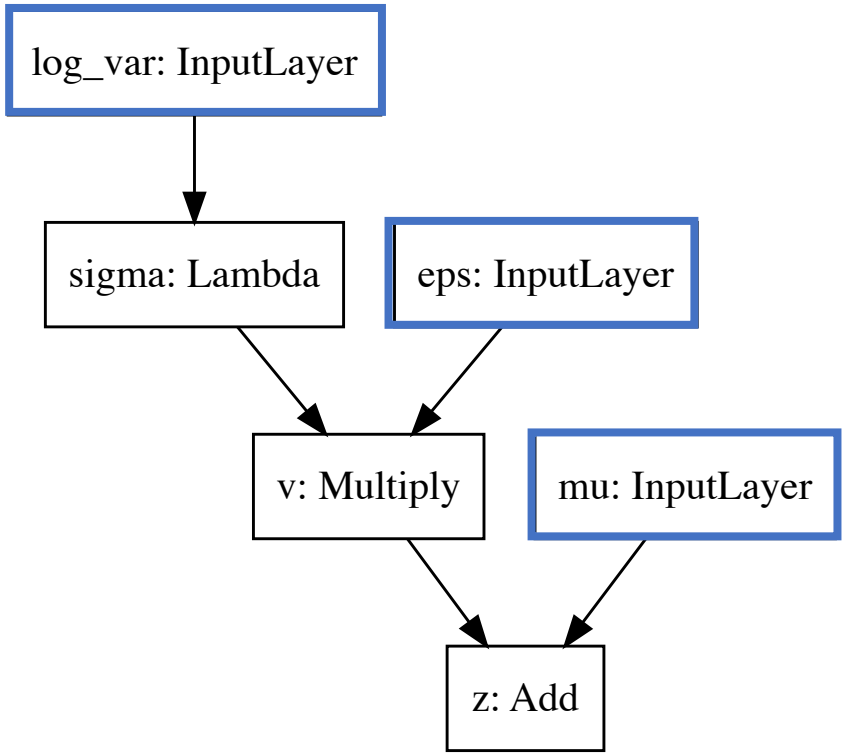
2. The Sampling Network

- 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 $\mathbf{z}$.



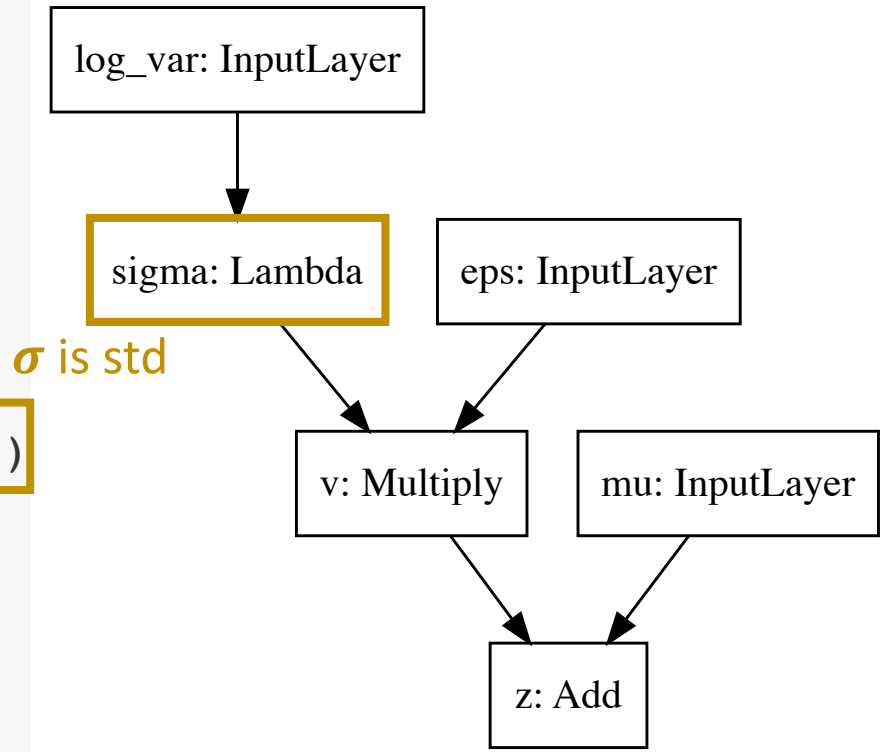
2. The Sampling Network

```
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')
```



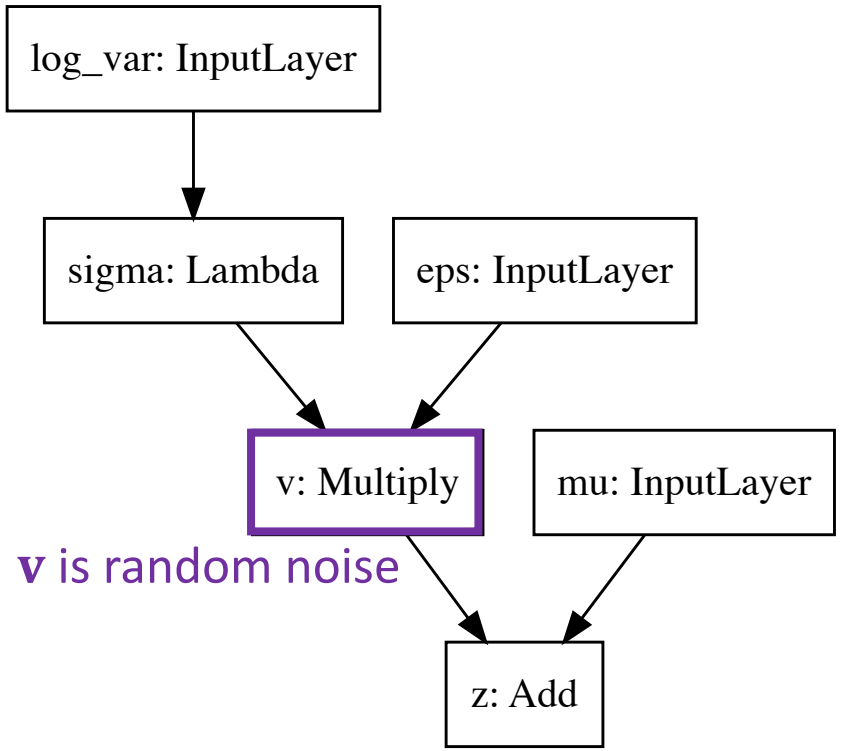
2. The Sampling Network

```
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

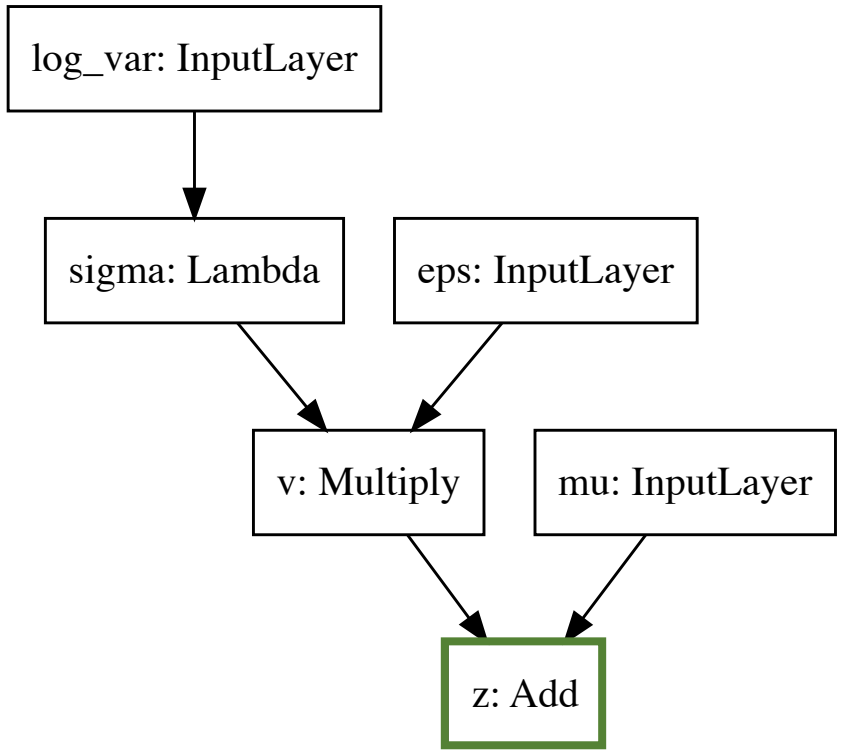
```
from keras.layers import Lambda, Multiply, Add
from keras import backend as K

# 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

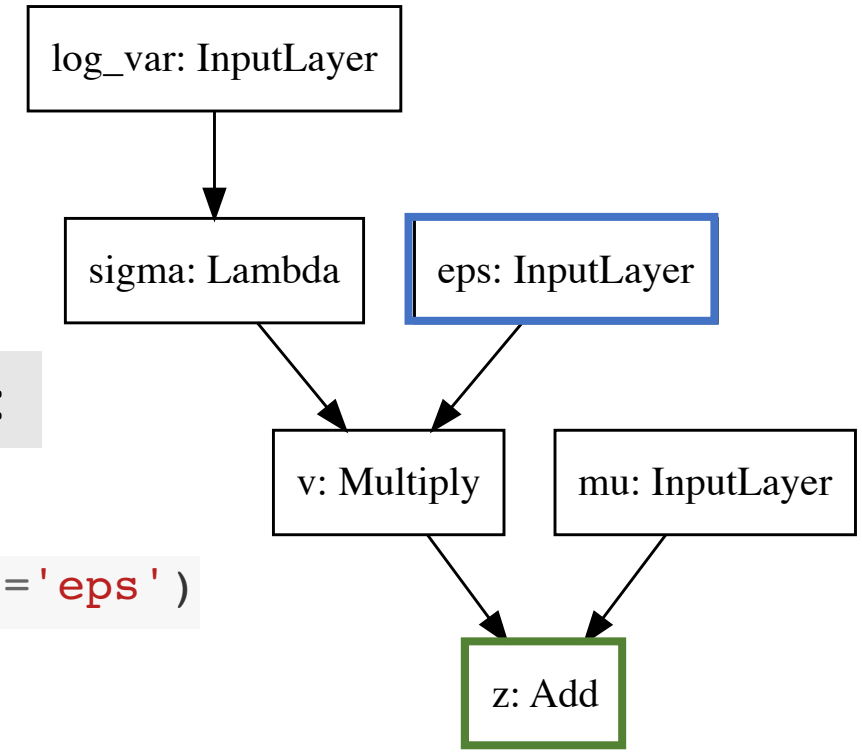
```
from keras.layers import Lambda, Multiply, Add
from keras import backend as K

# 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 $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_k)$?

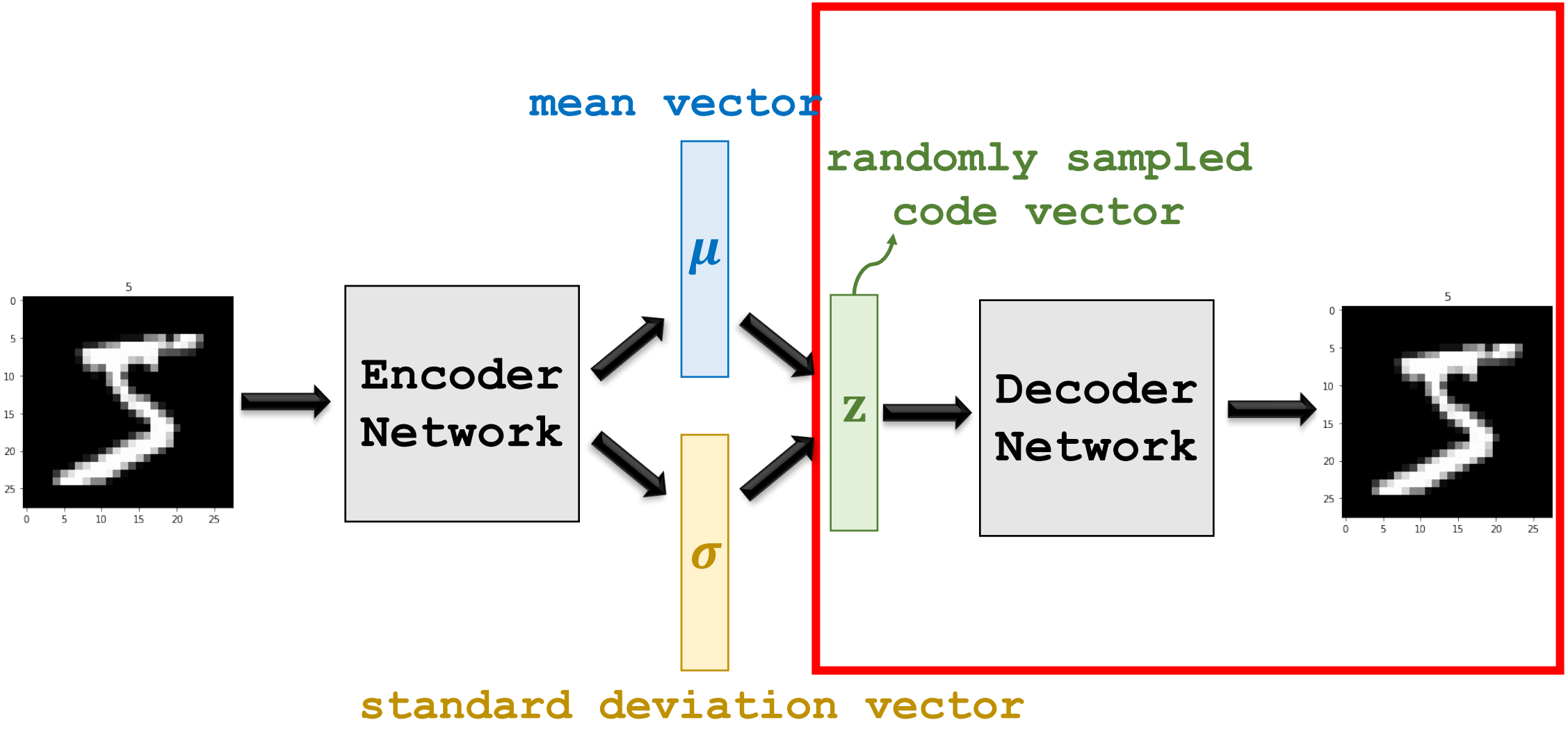


During training, set the input layer "eps" to the following code:

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

z is the sampled latent vector

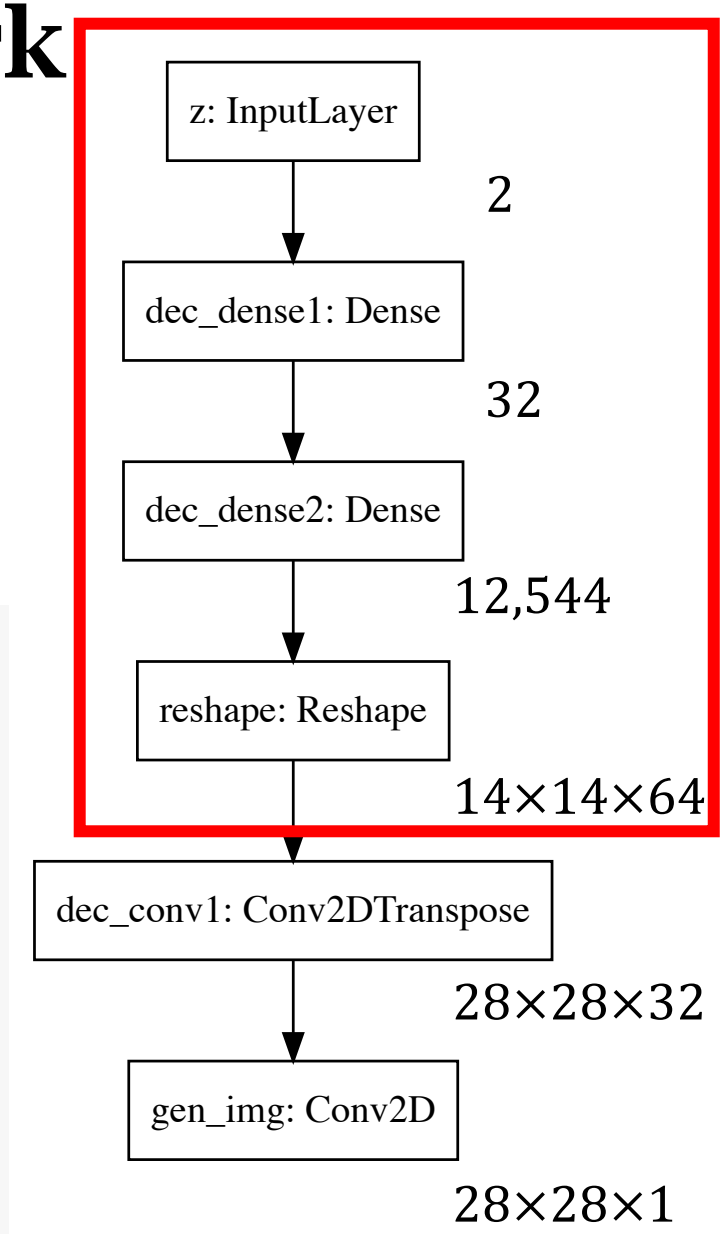
3. The Decoder Network



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)
```

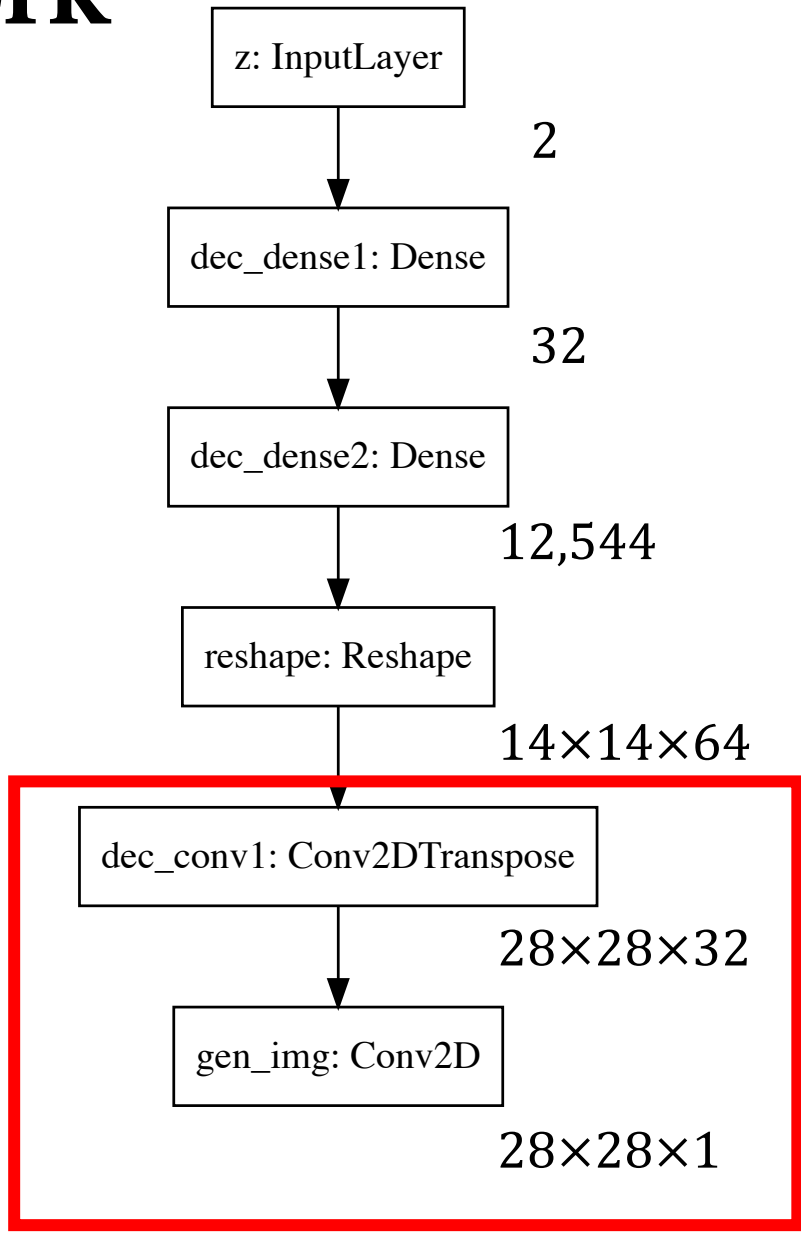
```
from keras.layers import Dense, Reshape, Conv2D, Conv2DTranspose

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)
```

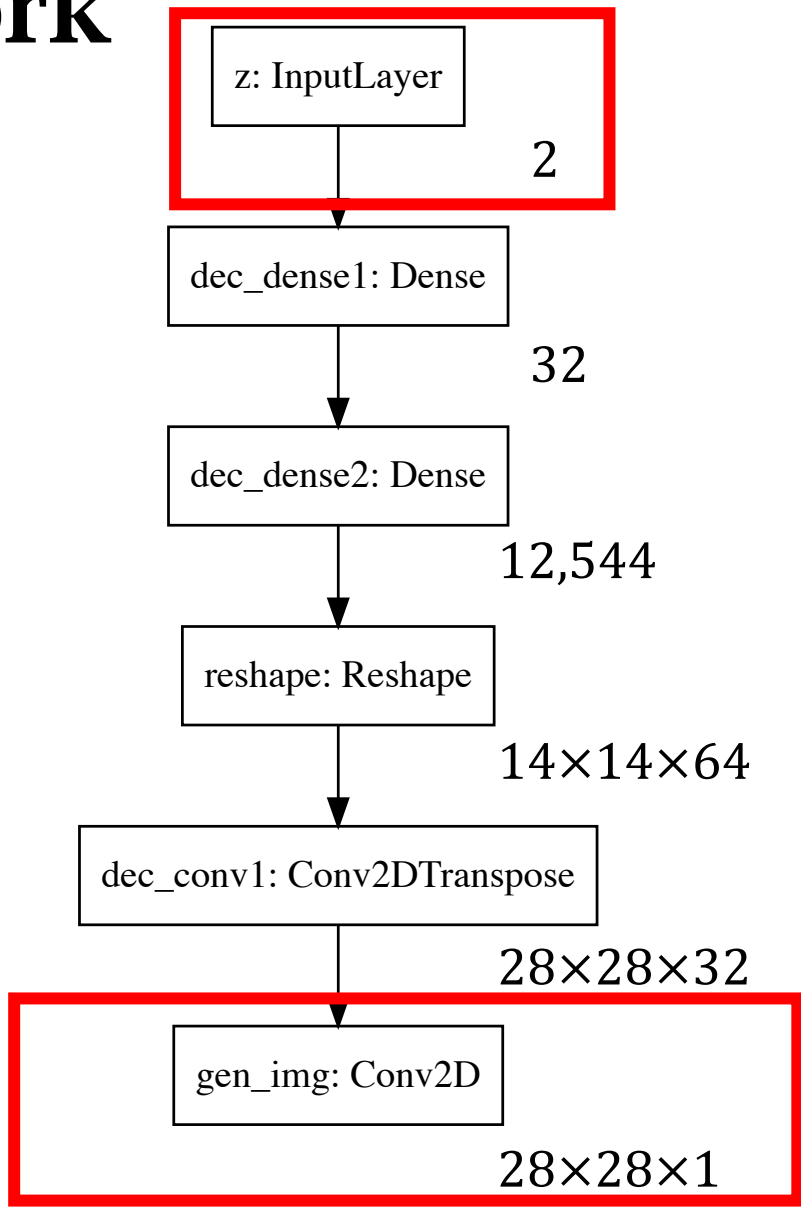
```
decoder = models.Model(inputs=z,
                       outputs=gen_img,
                       name='decoder')
```



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)
```

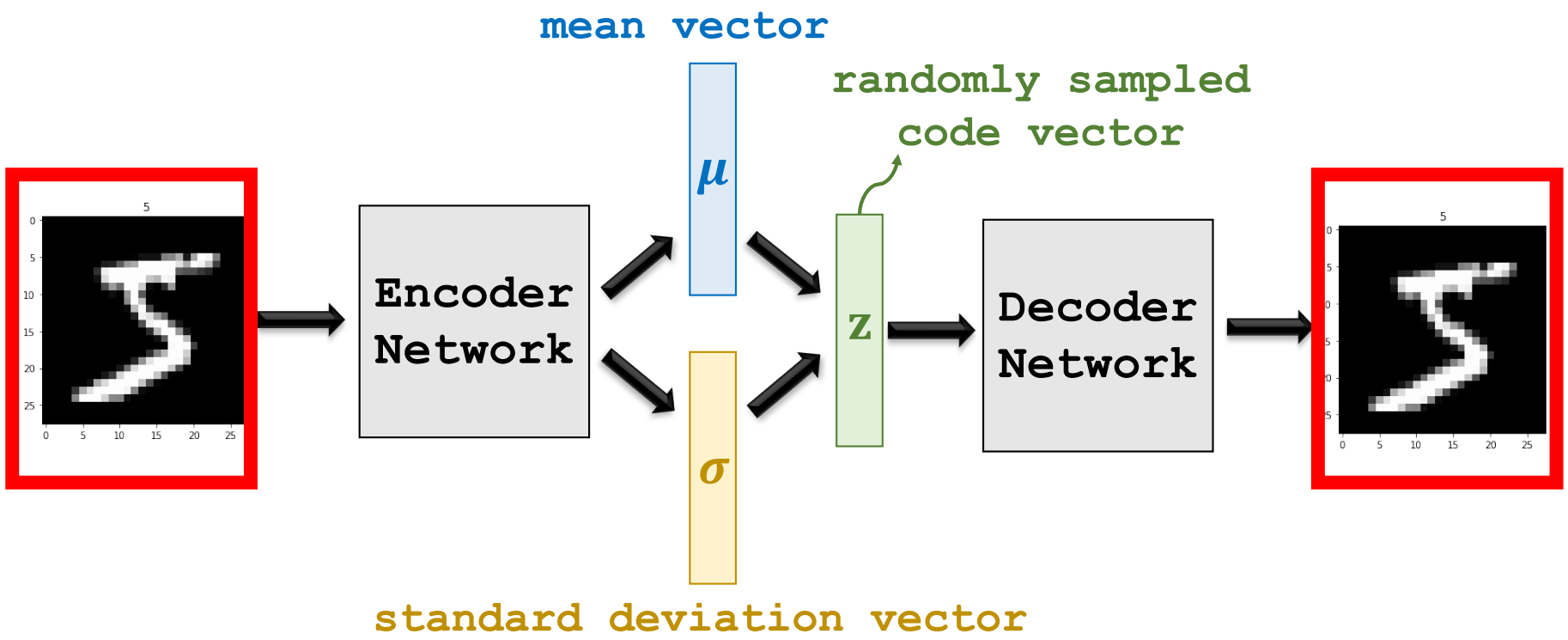
```
decoder = models.Model(inputs=z,
                       outputs=gen_img,
                       name='decoder')
```



Loss Functions

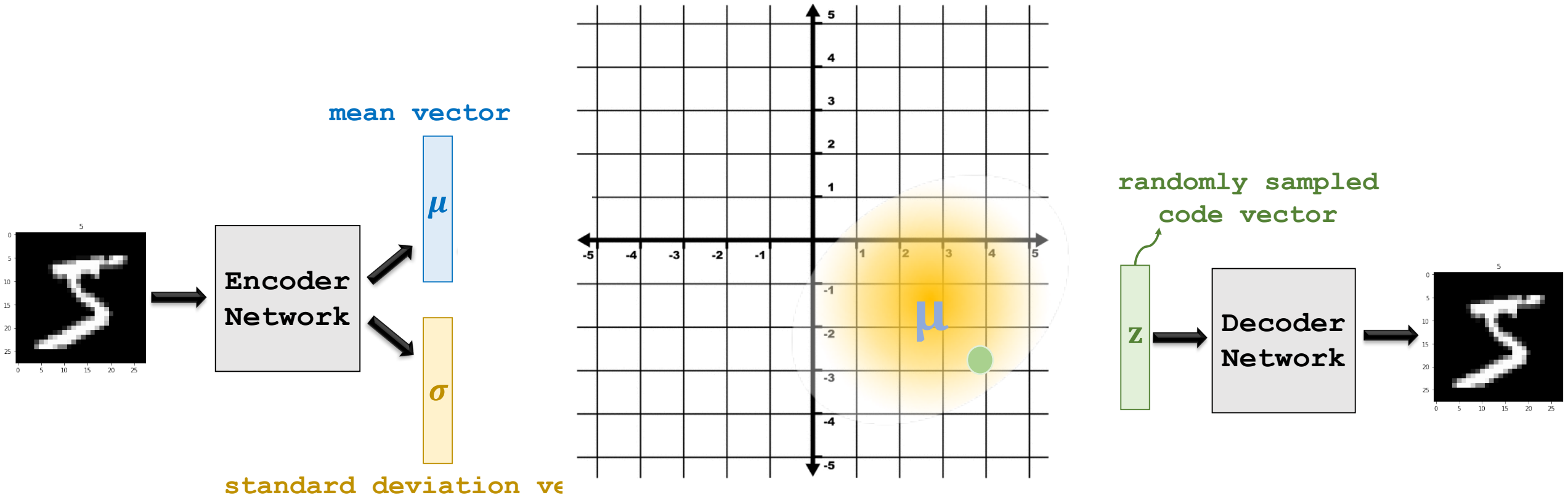
Generation Loss

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



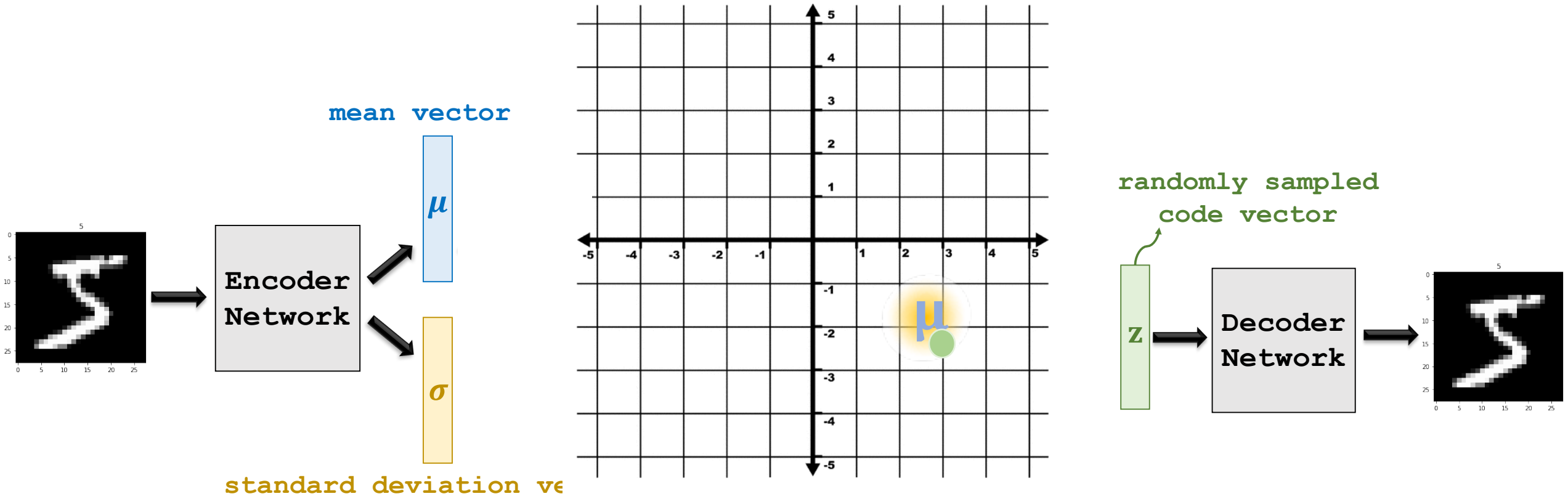
A problem with generation loss

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



A problem with generation loss

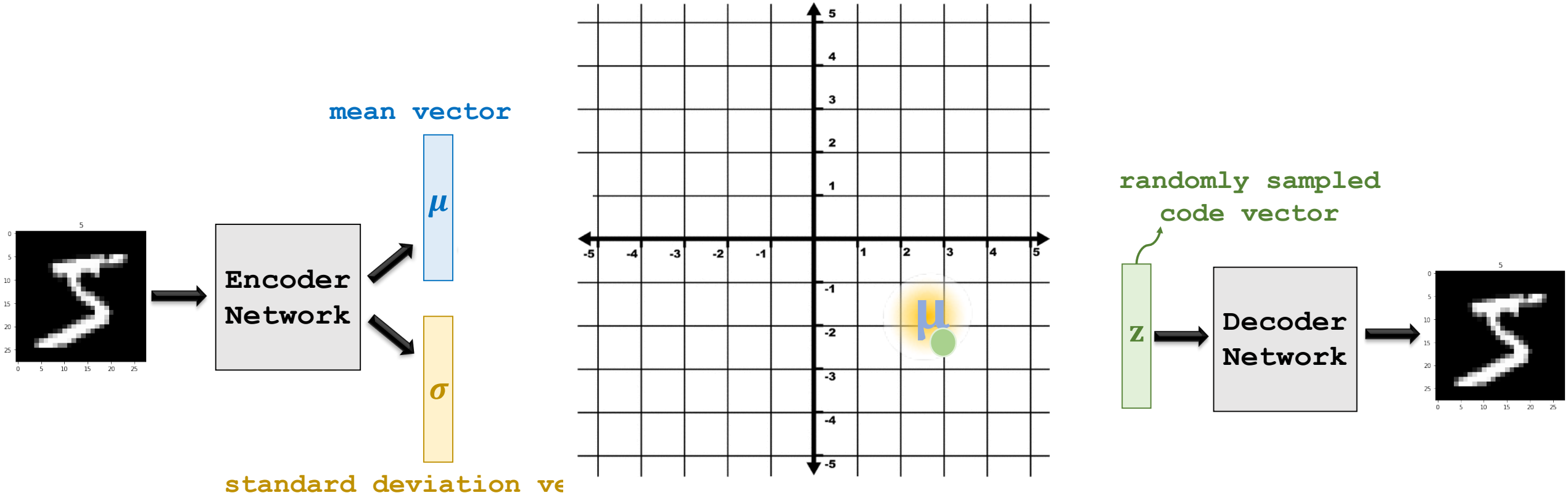
Difficulty: The encoder network will learn an std vector $\sigma \to 0$.



A problem with generation loss

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

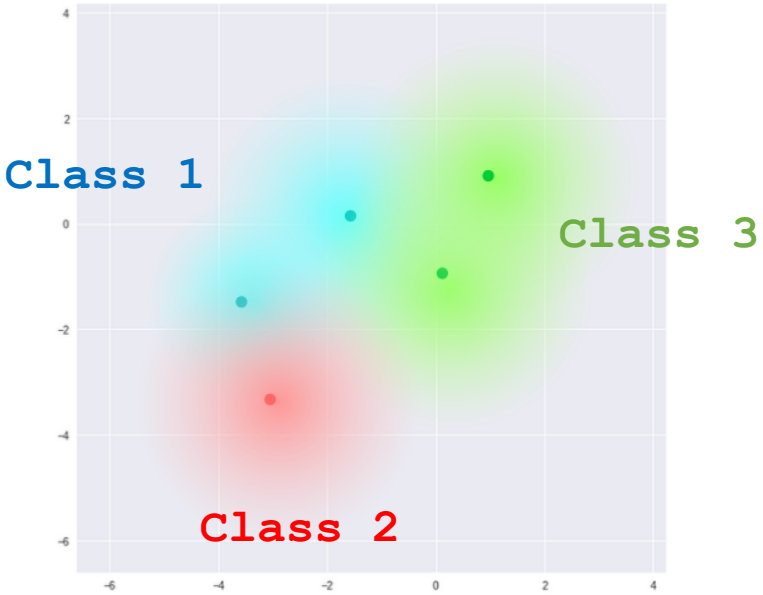
- Minimize `GenLoss` ➔ encourage z close to μ ➔ encourage small σ.
- VAE degrades to the standard AE (VAE with $\sigma = 0$ is exactly AE).



A problem with generation loss

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

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



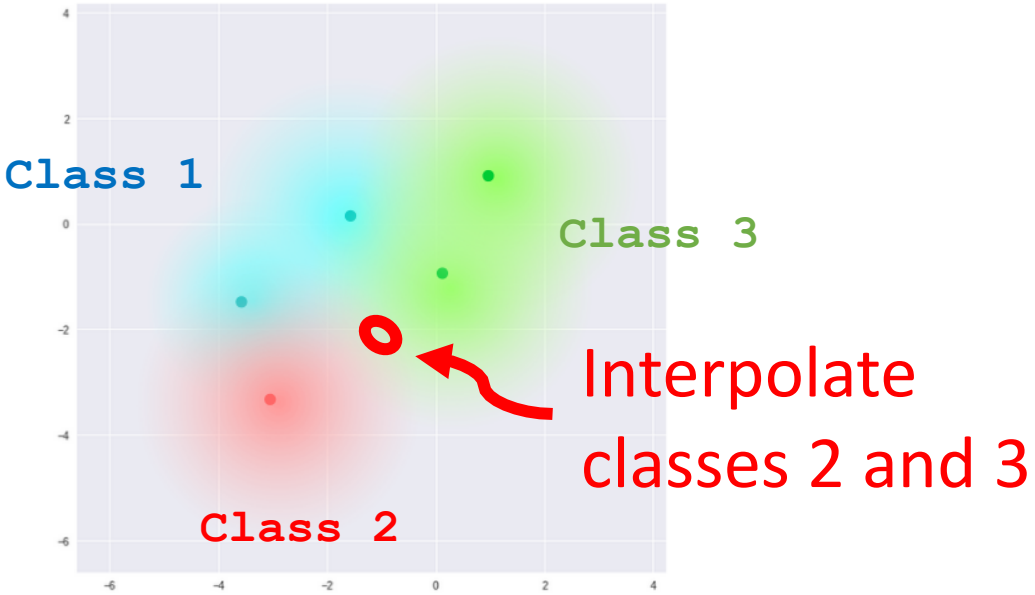
What we hope to learn.

What we actually learn by minimizing `GenLoss`.

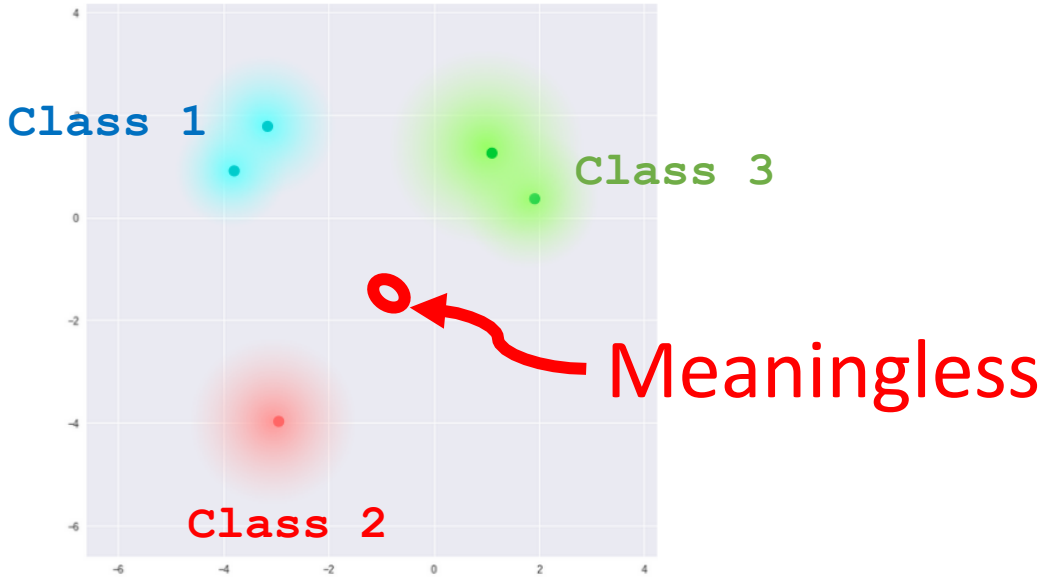
A problem with generation loss

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

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



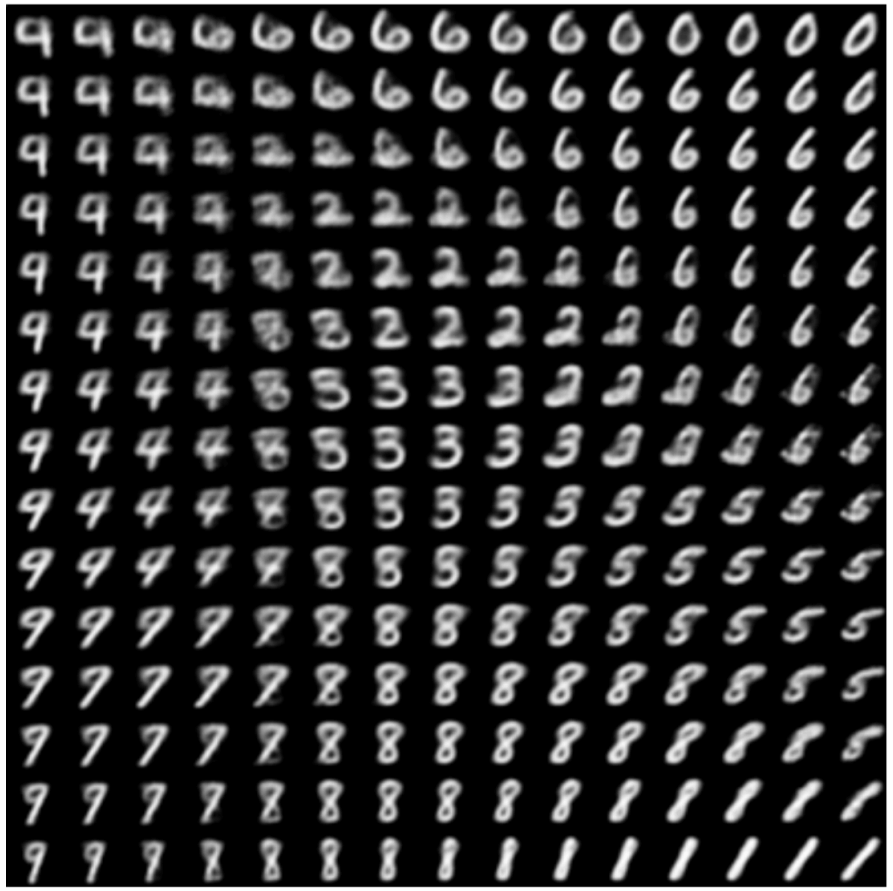
What we hope to learn.



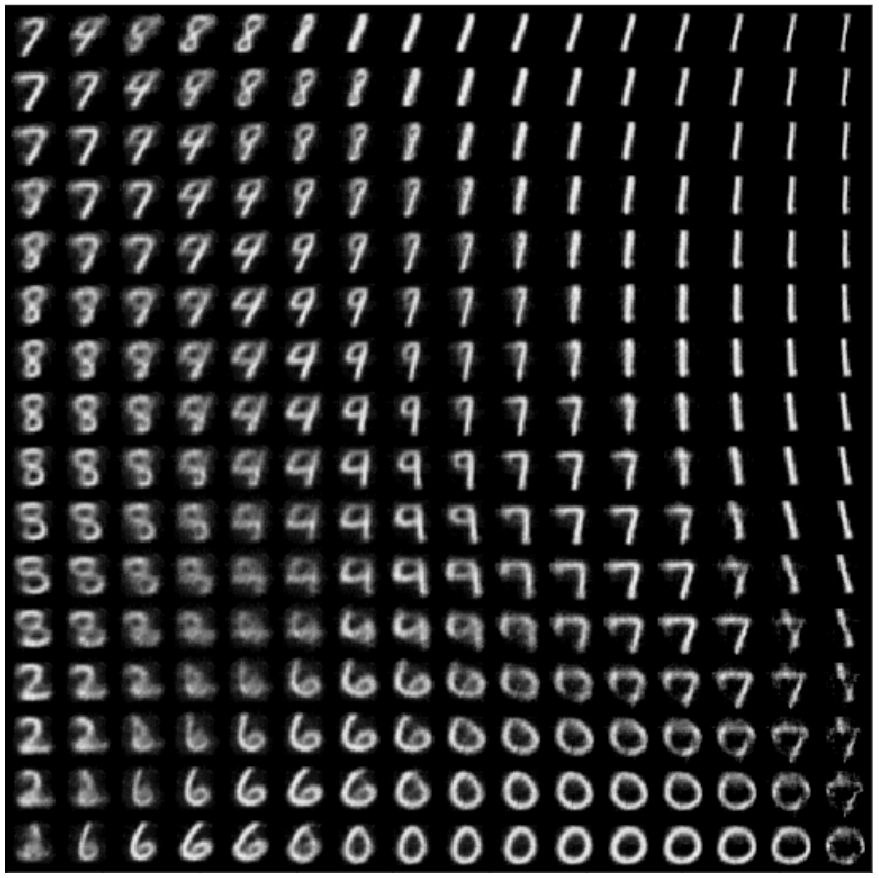
What we actually learn by minimizing `GenLoss`.

A problem with generation loss

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



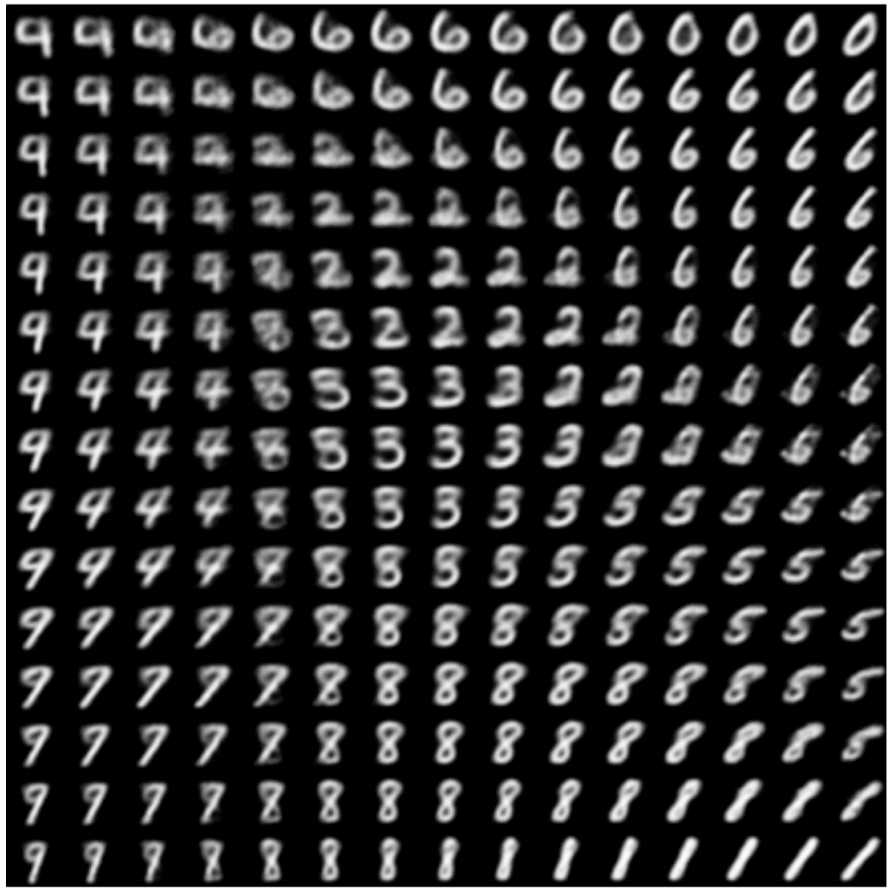
What we hope to learn.



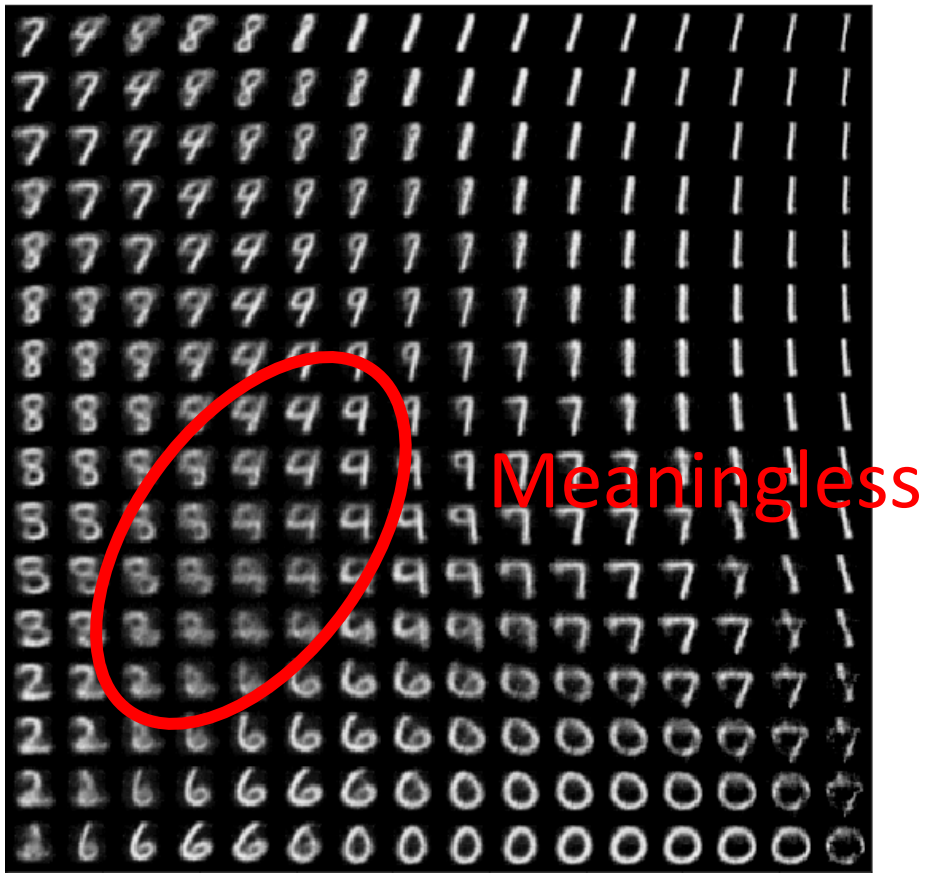
What we actually learn by minimizing `GenLoss`.

A problem with generation loss

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



What we hope to learn.

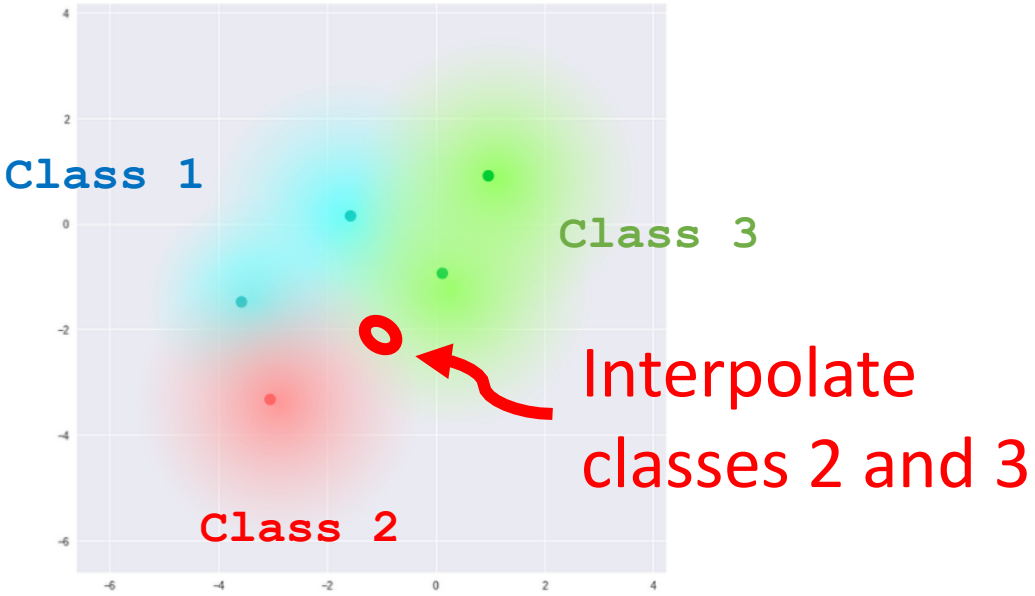


What we actually learn by minimizing `GenLoss`.

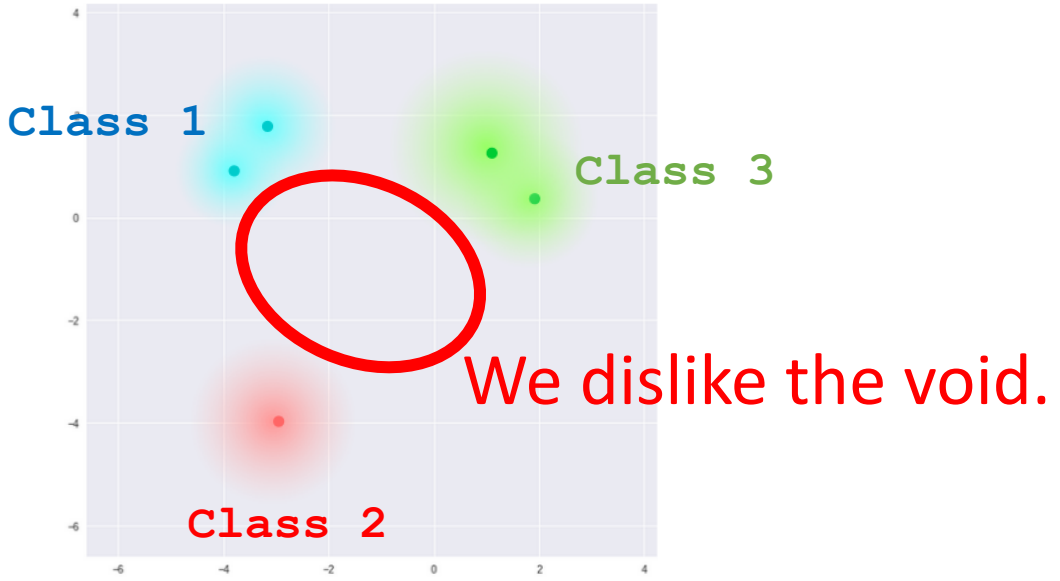
KL Loss

Idea: Encourage the distribution of z to be unit Gaussian.

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



What we hope to learn.

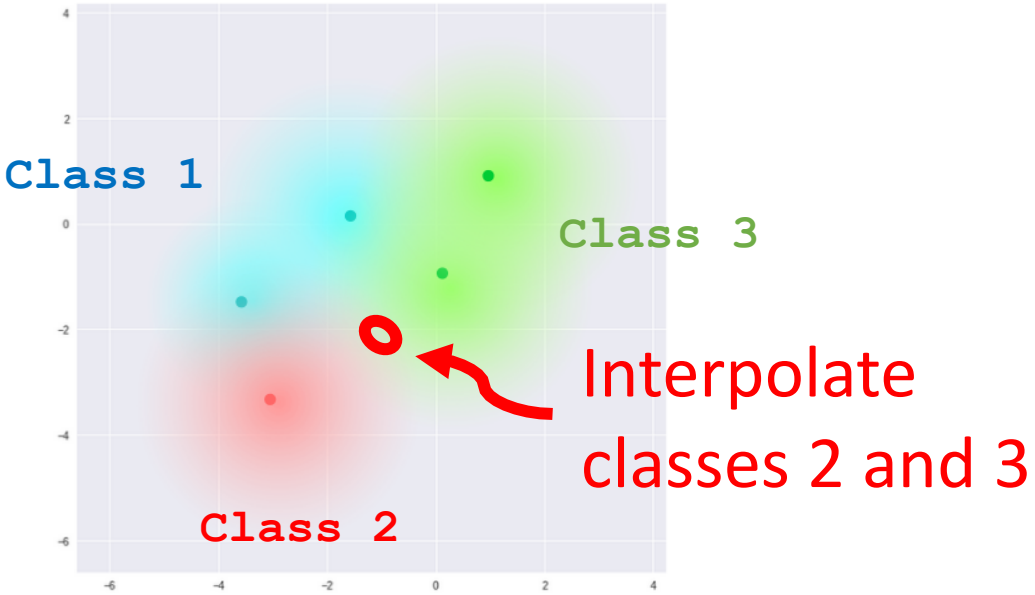


What we actually learn by minimizing `GenLoss`.

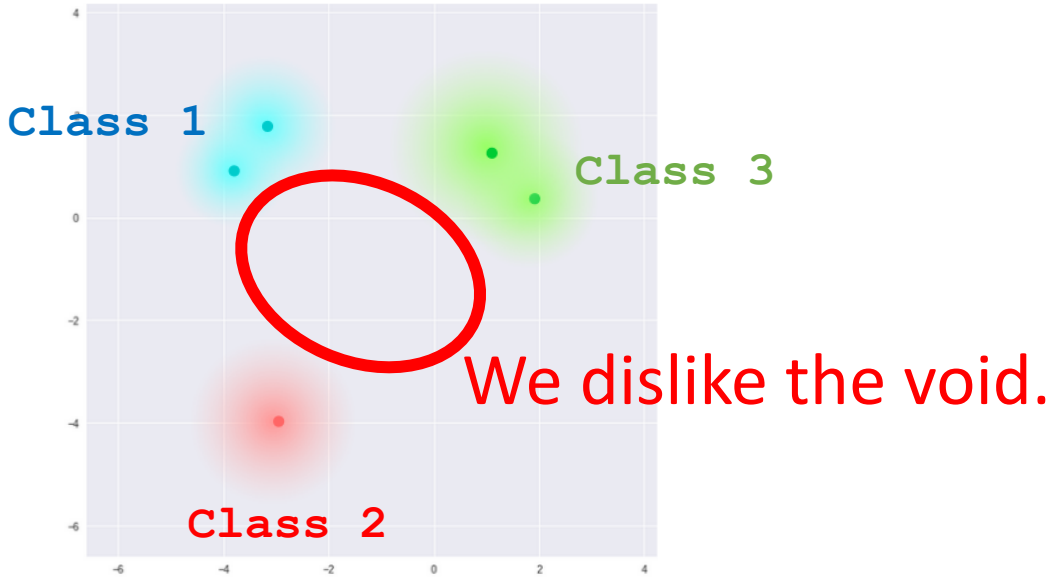
KL Loss

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:

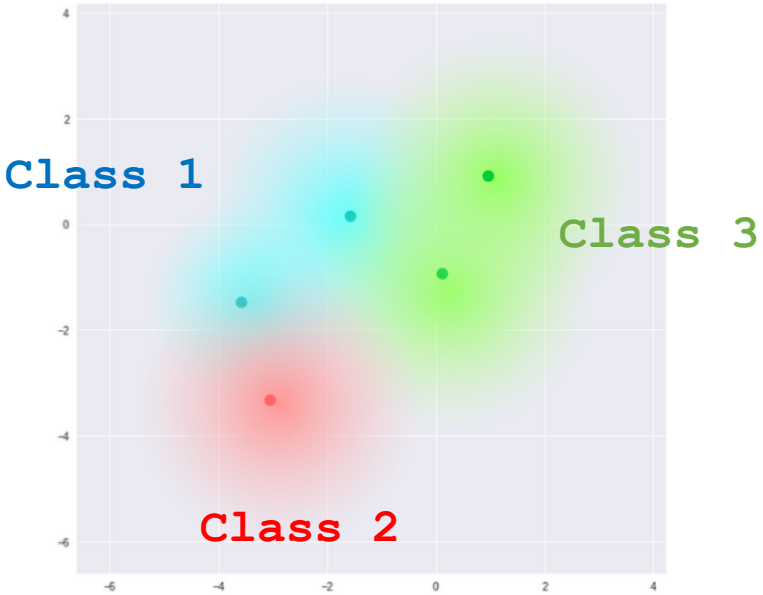
$$\text{KLLoss} = D_{\text{KL}}\left(p(\mathbf{z}) \;||\; \mathcal{N}(0, \mathbf{I})\right)$$

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

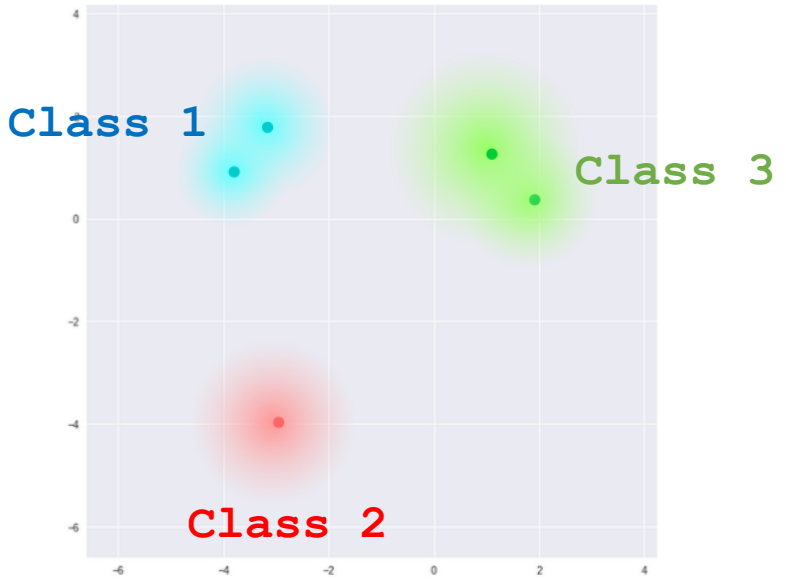
Another interpretation:

- 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

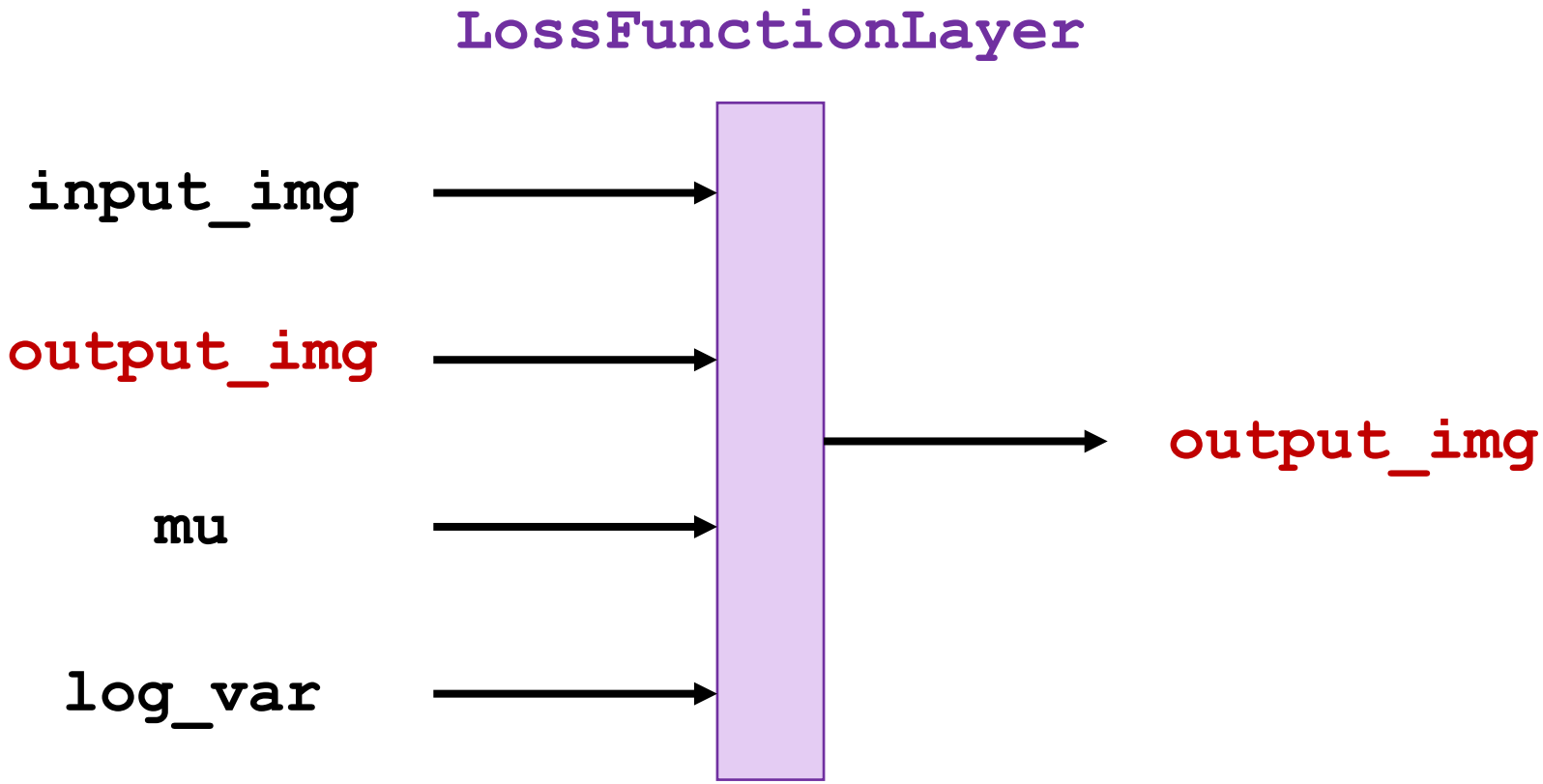
tuning hyper-parameter

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')
```

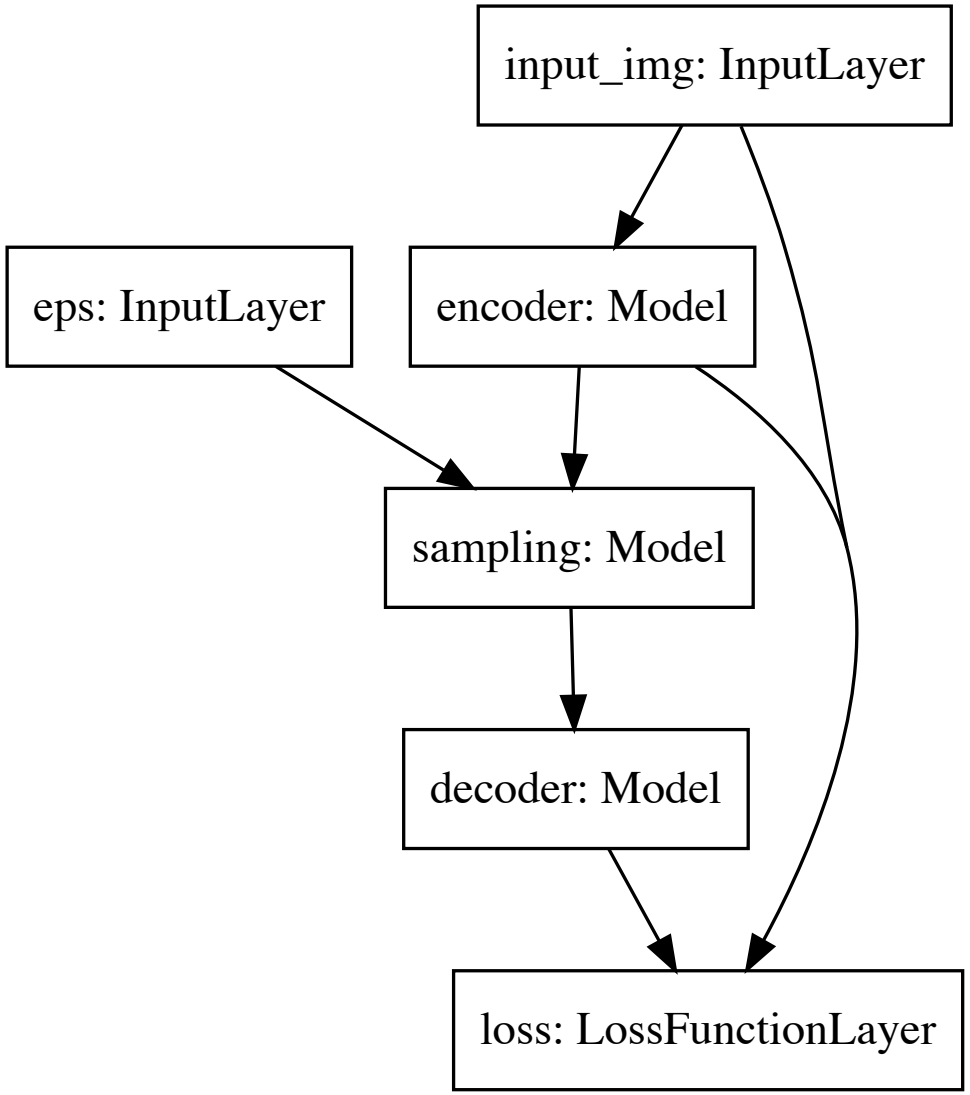
Connect the modules

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)
)
```

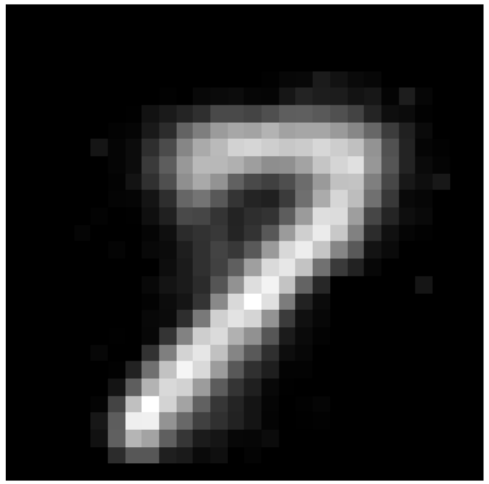
```
Train on 60000 samples, validate on 10000 samples
Epoch 1/50
60000/60000 [==============================] - 231s 4ms/step - loss: 0.6717 - val_loss: 0.2194
Epoch 2/50
60000/60000 [==============================] - 251s 4ms/step - loss: 0.2139 - val_loss: 0.2083
Epoch 3/50
60000/60000 [==============================] - 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()
```

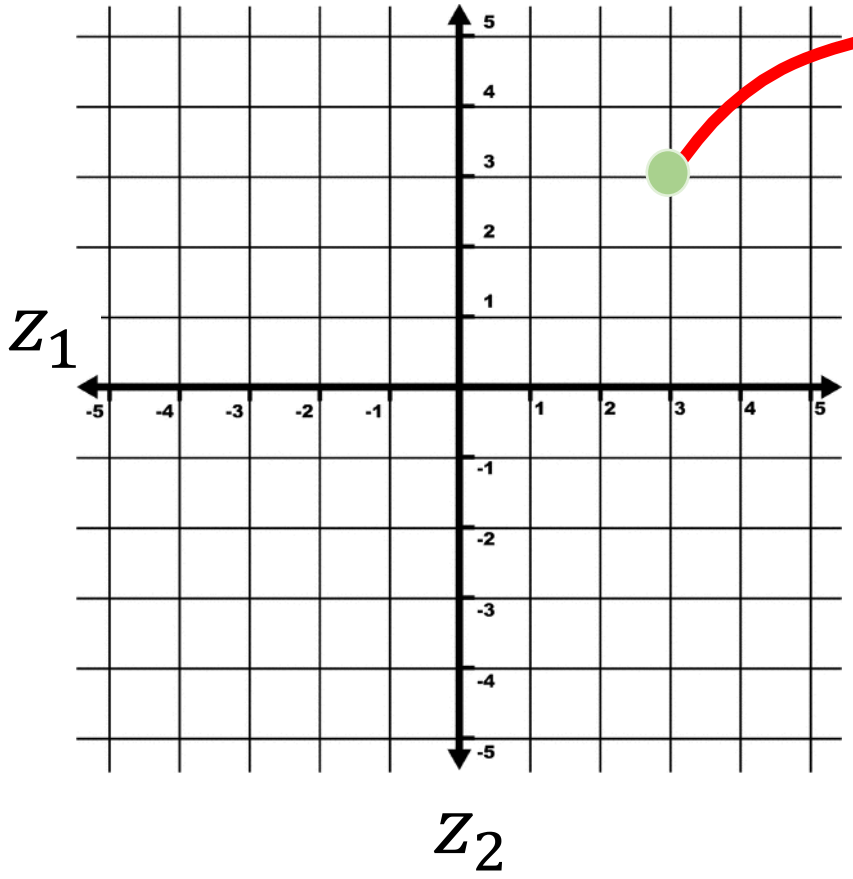


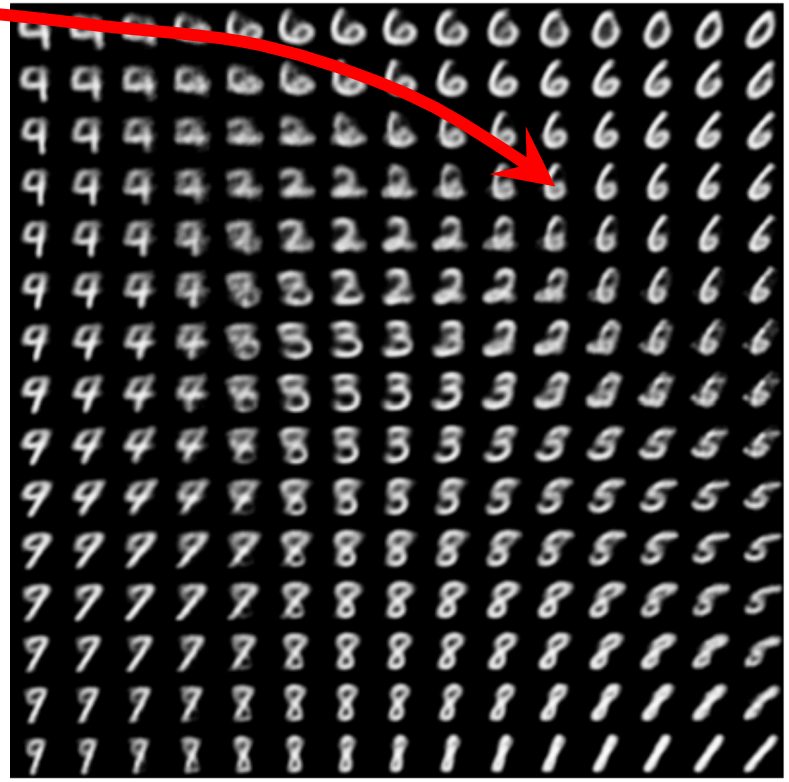
Visualize code vectors

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.