COPYRIGHT WARNING: Copyright in these original lectures is owned by Monash University. You may transcribe, take notes, download or stream lectures for the purpose of your research and study only. If used for any other purpose, (excluding exceptions in the Copyright Act 1969 (Cth)) the University may take legal action for infringement of copyright.

Do not share, redistribute, or upload the lecture to a third party without a written permission!

FIT3181 Deep Learning

Week 09: Representation Learning (II): Autoencoder and Variational Auto-Encoder

Lecturer: Lim Chern Hong

Email: lim.chernhong@monash.edu

Department of Data Science and AI
Faculty of Information Technology, Monash University, Australia

Outline

- Revision of some basic knowledge
- Learning efficient representations
- Auto-Encoder
  - Standard Auto-Encoder, Sparse, Contractive, Denoising Auto-Encoders
- Stochastic Auto-Encoder
  - Variational Auto-Encoder
- Further reading recommendation
  - [Hands-On, ch15]
  - [Deep learning, ch14]

Revision of some basic knowledge

Revision of basic knowledge

- Given two **discrete distributions** $p = [p_i]_{i=1}^d$ ($p_i \geq 0$ and $\sum_{i=1}^d p_i = 1$) and $q = [q_i]_{i=1}^d$ ($q_i \geq 0$ and $\sum_{i=1}^d q_i = 1$).
- **Kullback-Leibler (KL) divergence between $\boldsymbol{p}, \boldsymbol{q}$**

$$KL(p,q) = \sum_{i=1}^{d} p_i \log \frac{p_i}{q_i}$$

- **Cross-entropy (CE) divergence between $\boldsymbol{p}, \boldsymbol{q}$**

  $CE(p,q) = -\sum_{i=1}^d p_i \log q_i = KL(p,q) + H(p)$ where $H(p) = -\sum_{i=1}^d p_i \log p_i$ is the entropy of p.

- **Cross-entropy between two Bernoulli distributions**
  - Given $0 \leq a, b \leq 1$, we **have two Bernoulli distributions** $Ber(a)$ and $Ber(b)$, the **CE divergence** between them is

$$CE([a, 1-a], [b, 1-b]) = -a \log b - (1-a)\log(1-b)$$

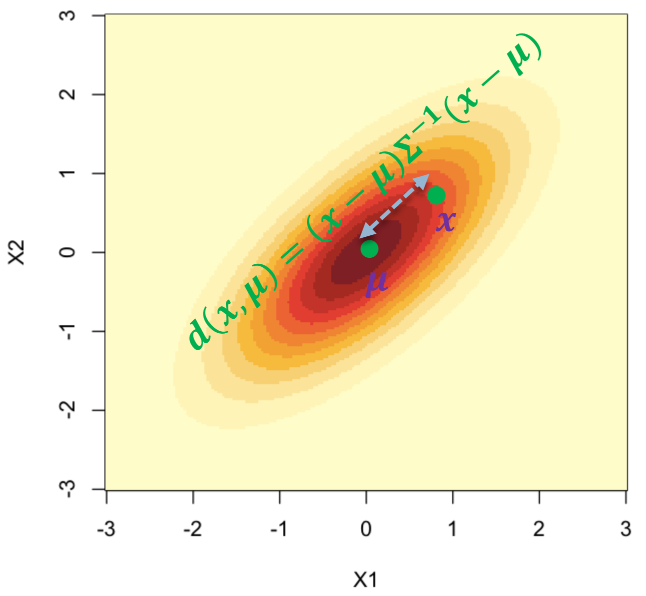
Revision of basic knowledge

- Given **two continuous distributions** with the **probability density functions** (pdf) $p(x)$ and $q(x)$ respectively
- **KL divergence between p and q**

$$KL(p,q) = \int p(\boldsymbol{x}) \log \frac{p(\boldsymbol{x})}{q(\boldsymbol{x})} d\boldsymbol{x}$$

- **Multivariate Gaussian distribution in $\mathbb{R}^d$**
  - $N(\boldsymbol{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{det(2\pi\Sigma)^{1/2}} \exp\{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{x}-\boldsymbol{\mu})\}$
- **KL divergence between two Gaussian distributions in $\mathbb{R}^d$**
  - $\boldsymbol{KL}\left(\boldsymbol{N}\left(\boldsymbol{\mu}, \boldsymbol{diag}(\boldsymbol{\sigma}^2)\right), \boldsymbol{N}(\boldsymbol{0}, \boldsymbol{I})\right) = \frac{1}{2}\left(\|\boldsymbol{\sigma}\|_2^2 + \|\boldsymbol{\mu}\|_2^2 - \boldsymbol{d} - \sum_{i=1}^{d} \log(\boldsymbol{\sigma}_i^2)\right)$

The derivation of the general case: https://mr-easy.github.io/2020-04-16-kl-divergence-between-2-gaussian-distributions/



Learning efficient representation

The importance of efficient/appropriate representation

Roman Numeral Table							
1	I	14	XIV	27	XXVII	150	CL
2	II	15	XV	28	XXVIII	200	CC
3	III	16	XVI	29	XXIX	300	CCC
4	IV	17	XVII	30	XXX	400	CD
5	V	18	XVIII	31	XXXI	500	D
6	VI	19	XIX	40	XL	600	DC
7	VII	20	XX	50	L	700	DCC
8	VIII	21	XXI	60	LX	800	DCCC
9	IX	22	XXII	70	LXX	900	CM
10	X	23	XXIII	80	LXXX	1000	M
11	XI	24	XXIV	90	XC	1600	MDC
12	XII	25	XXV	100	C	1700	MDCC
13	XIII	26	XXVI	101	CI	1900	MCM

MathATube.com

What is LX divided by V?

The importance of efficient/appropriate representation

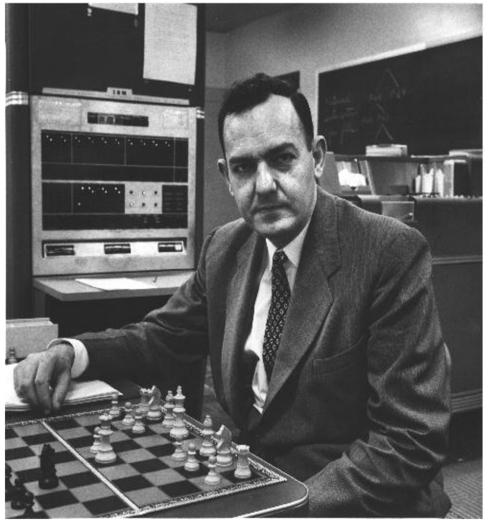
Roman Numeral Table							
1	I	14	XIV	27	XXVII	150	CL
2	II	15	XV	28	XXVIII	200	CC
3	III	16	XVI	29	XXIX	300	CCC
4	IV	17	XVII	30	XXX	400	CD
5	V	18	XVIII	31	XXXI	500	D
6	VI	19	XIX	40	XL	600	DC
7	VII	20	XX	50	L	700	DCC
8	VIII	21	XXI	60	LX	800	DCCC
9	IX	22	XXII	70	LXX	900	CM
10	X	23	XXIII	80	LXXX	1000	M
11	XI	24	XXIV	90	XC	1600	MDC
12	XII	25	XXV	100	C	1700	MDCC
13	XIII	26	XXVI	101	CI	1900	MCM

MathATube.com

What is 60 divided by 5?

Efficient/appropriate representations

William Chase



Herbert Simon

- The relationship between memory, perception, and pattern matching was studied by William Chase and Herbert Simon in 1970s
  - How can our brain memorise complicated things?
  - How can our brain work out efficient internal representations?

Efficient/appropriate representations

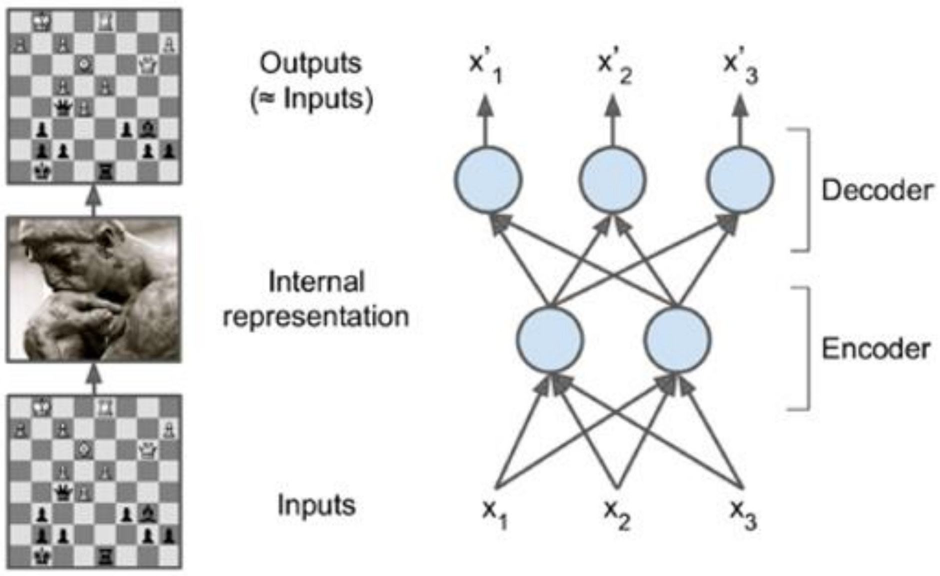
- Expert chess players can memorise the positions of all the pieces in a game within 5 seconds
  - A task most of us would find **impossible**
  - Expert chess players do not have **much better memory** than us. How can they do that?

- This is only the case when all pieces are placed in realistic positions from actual games, not when the pieces are placed in random positions

Efficient/appropriate representations

Expert chess player

- **Encoding**
  - Expert chess player do not memorise the positions, they instead memorise the patterns
  - Transform all the piece positions to the patterns in memory
- **Decoding**
  - Reconstruct the piece positions from the patterns in memory



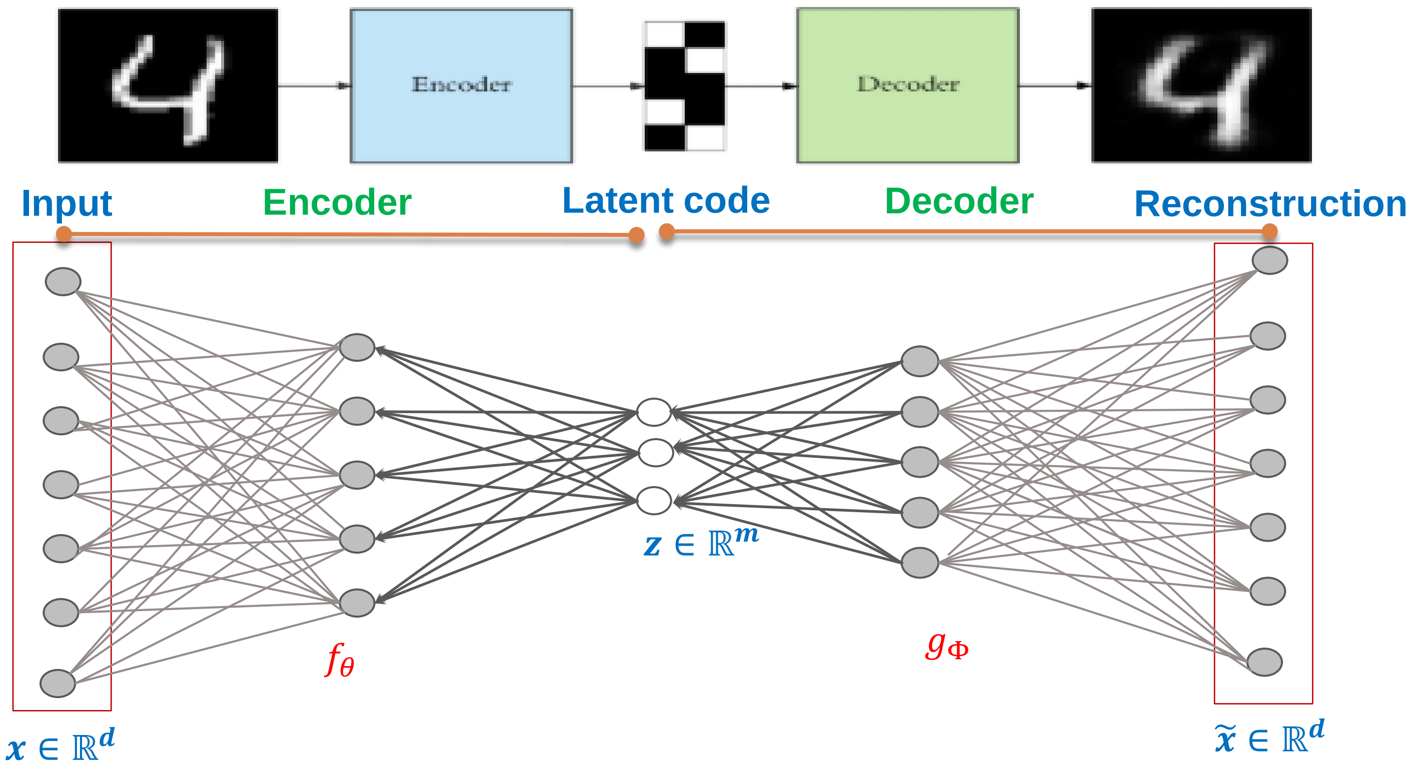
(Source: Hands-On Ch15)

Auto-Encoder

- **Encoding (Encoder)**
  - Transform input data to internal representation or (lossy/lossless) summary
- **Decoding (Decoder)**
  - Reconstruct input data from internal representations or (lossy/lossless) summary

Deep Auto-Encoder

Auto-Encoder

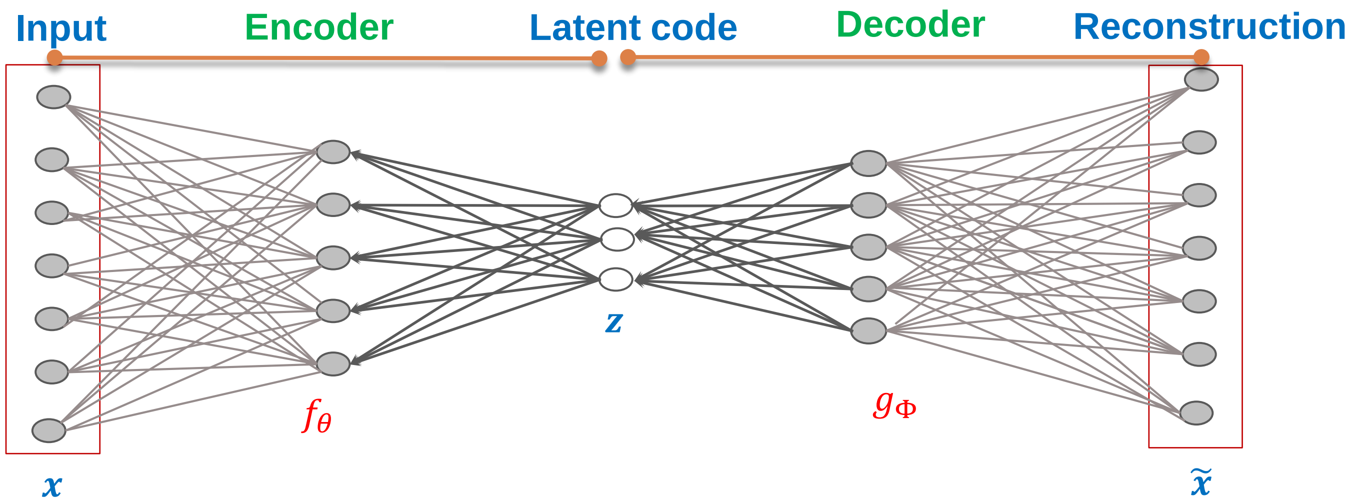


- $\boldsymbol{x} \sim \mathbb{P}$
  - $\mathbb{P}$ is the data distribution over $\mathbb{R}^{d}$
- **Encoding**
  - $\boldsymbol{z} = f_{\theta}(\boldsymbol{x}) \in \mathbb{R}^{m}$
- **Decoding**
  - $\widetilde{\boldsymbol{x}} = g_{\Phi}(\boldsymbol{z})$ is said to be the **reconstruction** of $\boldsymbol{x}$.

- How to justify that the **latent code** $\boldsymbol{z}$ can **preserve crucial information** of its input $\boldsymbol{x}$?
- How **accurate** we can **reconstruct** $\boldsymbol{x}$ from $\boldsymbol{z}$?
  - Reconstruction error: $d(\boldsymbol{x}, \widetilde{\boldsymbol{x}})$
  - Distance between $\boldsymbol{x}$ and $\widetilde{\boldsymbol{x}}$

Auto-Encoder

Reconstruction error



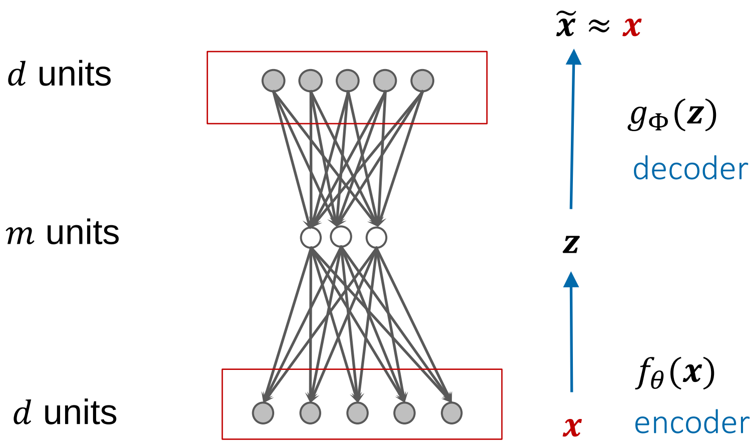
- Minimize **reconstruction error** over training set $D = \{\boldsymbol{x}_1, \dots, \boldsymbol{x}_N\}$
  - $\min_{\theta,\Phi} \mathbb{E}_{\boldsymbol{x}\sim\mathbb{P}} [d(\boldsymbol{x}, \tilde{\boldsymbol{x}})] = \min_{\theta,\Phi} \mathbb{E}_{\boldsymbol{x}\sim\mathbb{P}} [d(\boldsymbol{x}, g_\Phi(f_\theta(\boldsymbol{x}))]$
  - $\min_{\theta,\Phi} \frac{1}{N} \sum_{i=1}^N d(\boldsymbol{x}_i, \tilde{\boldsymbol{x}}_i) = \min_{\theta,\Phi} \frac{1}{N} \sum_{i=1}^N d(\boldsymbol{x}_i, g_\Phi(\boldsymbol{z}_i)) = \min_{\theta,\Phi} \frac{1}{N} \sum_{i=1}^N d(\boldsymbol{x}_i, \overbrace{g_\Phi(\underbrace{f_\theta(\boldsymbol{x}_i)}_{\boldsymbol{z}_i})}^{\tilde{\boldsymbol{x}}_i})$

- How to **define** $d(\boldsymbol{x}, \tilde{\boldsymbol{x}})$?
  - $\boldsymbol{x}, \tilde{\boldsymbol{x}} \in \mathbb{R}^d$: $d(\boldsymbol{x}, \tilde{\boldsymbol{x}}) = \frac{1}{2} \|\boldsymbol{x} - \tilde{\boldsymbol{x}}\|_2^2$ (L2 distance)
  - $\boldsymbol{x}, \tilde{\boldsymbol{x}} \in [0,1]^d$ (applied **sigmoid** on the output): $d(\boldsymbol{x}, \tilde{\boldsymbol{x}}) = \sum_{i=1}^d CE([x_i, 1 - x_i], [\tilde{x}_i, 1 - \tilde{x}_i])$

$$= \sum_{i=1}^d [-x_i \log \tilde{x}_i - (1 - x_i) \log(1 - \tilde{x}_i)]$$

Undercomplete Auto-Encoder

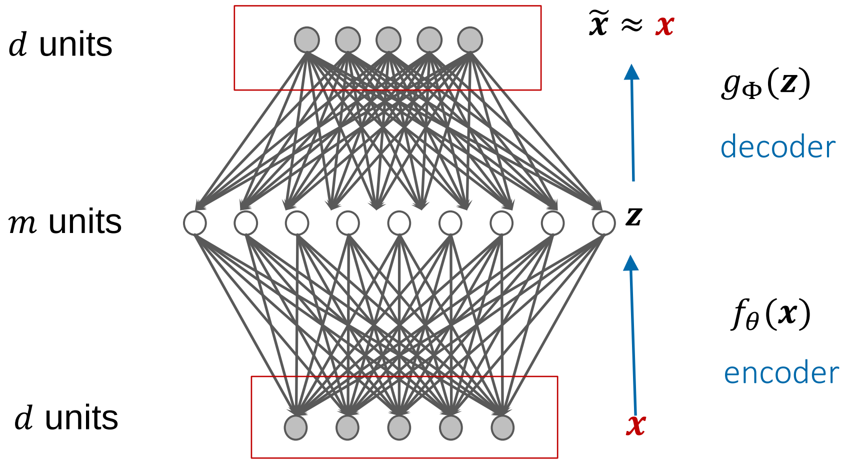
- Why would one want to learn to copy the input to the output?
  - We are **not actually interested** in the output
  - The hope is that by learning to perform copying from input to output via the intermediate code z, this code will capture useful and key properties of the data
- Ensure the code z learn useful information is through 'compression'
  - *Letting dimension of the code z to be smaller than the dimension of the input.*
  - This is called an undercomplete AE
- When the decoder is linear and mean squared error loss is used, this is identical to Principal Component Analysis (PCA)



Undercomplete AE when $m < d$

Overcomplete Auto-Encoder

- Overcomplete AE is when the latent code dimension m is greater than the input dimension d.
- In undercomplete AE, $m < d$, hence the code can learn salient features of the data,
  - For overcomplete case, the encoder/decoder could be **too powerful**, hence it can copy (even perfectly) without learning any useful code z !
- How to make overcomplete AE **useful**?
  - Regularization



Overcomplete AE when $m \geq d$

Implementation of Auto-Encoder

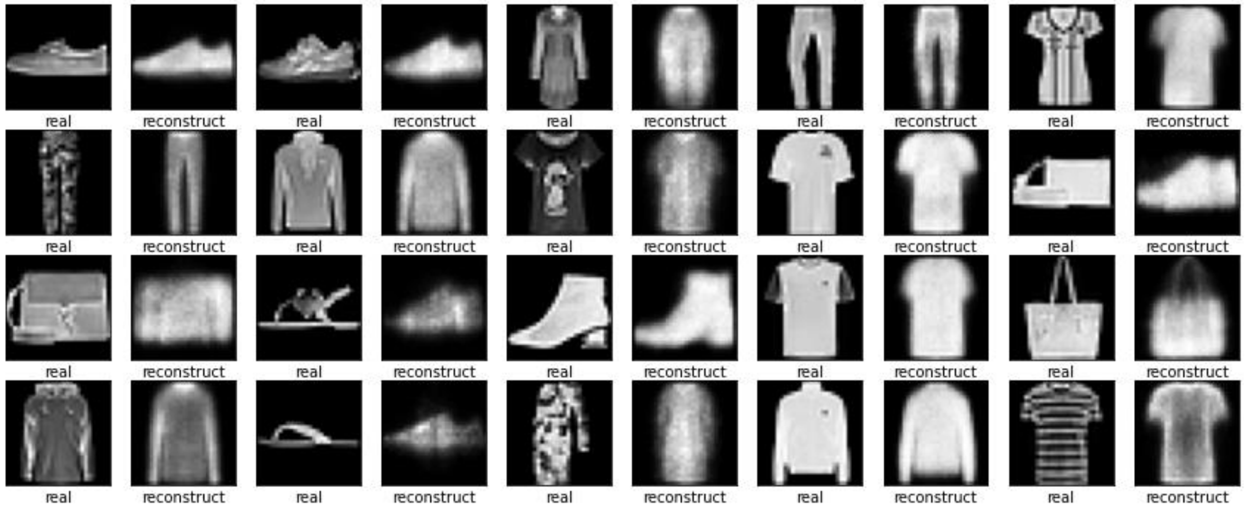
```python
class GeneralAE:
    def __init__(self, optimizer = keras.optimizers.SGD(lr=0.1)):
        self.encoder = None
        self.decoder = None
        self.auto_encoder = None
        self.optimizer = optimizer

    @staticmethod
    def rounded_accuracy(y_true, y_pred):
        return keras.metrics.binary_accuracy(tf.round(y_true), tf.round(y_pred))

    def encode(self, X=None):
        return self.encoder.predict(X)

    def decode(self, h=None):
        return self.decoder.predict(h)

    def reconstruct(self, X=None):
        self.auto_encoder.predict(X)
```

```python
    def show_reconstructions(self, X= None, n_cols = 5):
        reconstructions = self.auto_encoder.predict(X)
        n_images = len(X)
        n_rows = math.ceil(n_images/n_cols)
        fig = plt.figure(figsize=(2*n_cols*1.5, n_rows*1.5))
        plt.axis("off")
        for i in range(n_images):
            plt.subplot(n_rows, 2*n_cols, 2*i+1)
            plt.imshow(X[i], cmap="gray")
            plt.xlabel("real")
            plt.xticks([])
            plt.yticks([])
            plt.grid(False)
            plt.subplot(n_rows, 2*n_cols, 2*i+2)
            plt.imshow(reconstructions[i], cmap="gray")
            plt.xlabel("reconstruct")
            plt.xticks([])
            plt.yticks([])
            plt.grid(False)

    def build(self):
        pass

    def train(self, *args, **kwargs):
        self.auto_encoder.fit(*args, **kwargs)
```

Implementation of Standard Auto-Encoder

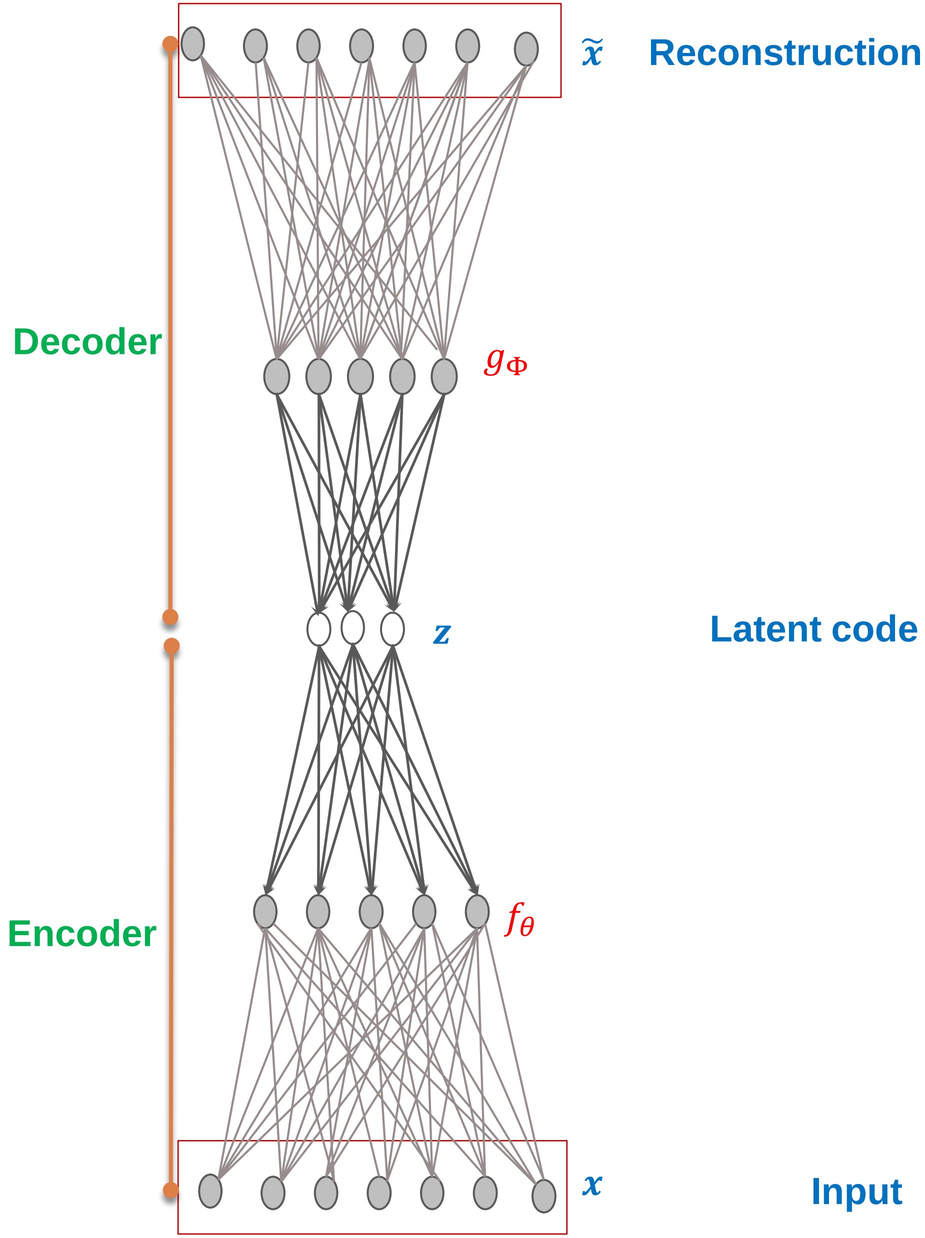
```
class StandardAE(GeneralAE):
    def __init__(self, optimizer = keras.optimizers.SGD(lr=0.1)):
        super(StandardAE, self).__init__(optimizer)

    def build(self):
        self.encoder = keras.models.Sequential([keras.layers.Flatten(input_shape=[28, 28]),
                                               keras.layers.Dense(100, activation="selu"),
                                               keras.layers.Dense(30, activation="selu")])

        self.decoder = keras.models.Sequential([keras.layers.Dense(100, activation="selu", input_shape=[30]),
                                                                   keras.layers.Dense(28 * 28, activation="sigmoid"),
                                                                   keras.layers.Reshape([28, 28])])

        self.auto_encoder = keras.models.Sequential([self.encoder, self.decoder])
        self.auto_encoder.compile(loss="binary_crossentropy", optimizer=self.optimizer, metrics=[GeneralAE.rounded_accuracy])
```

```
standard_ae.show_reconstructions(X_valid[:20], n_cols=5)
```



Sparse Auto-Encoder

- Given a latent code $\boldsymbol{z}$, the sparsity of $\boldsymbol{z}$ is defined as the number of **zero elements** in $\boldsymbol{z}$ or $m - ||\boldsymbol{z}||_0$
  - $\boldsymbol{z_1} = [\mathbf{1.2, 0, -1, 0, 1}] \rightarrow \boldsymbol{sparsity}(\boldsymbol{z}_1) = 2$
  - $\boldsymbol{z_2} = [\mathbf{2.4, 0, -2, 0, 0}] \rightarrow \boldsymbol{sparsity}(\boldsymbol{z}_2) = 3$
  - In general, sparser $\boldsymbol{z}$ is, more elements around 0 it has
- We want to find **sparse representation** of the latent code $\boldsymbol{z}$ of $\boldsymbol{x}$ that is still able to reconstruct well $\boldsymbol{x}$.
  - Hope that the training would **eliminate redundant** elements
- **Sparse auto-encoder**
  - $\min_{\theta,\Phi} \mathbb{E}_{\boldsymbol{x}\sim\mathbb{P}} [d(\boldsymbol{x}, g_\Phi(f_\theta(\boldsymbol{x})))] + \lambda\Omega(\boldsymbol{z})$
  - $\Omega(\boldsymbol{z})$ is a regularization which is usually a norm over $\boldsymbol{z}$
  - $\lambda > 0$ **is** regularization parameter.



Sparse Auto-Encoder

Regularization

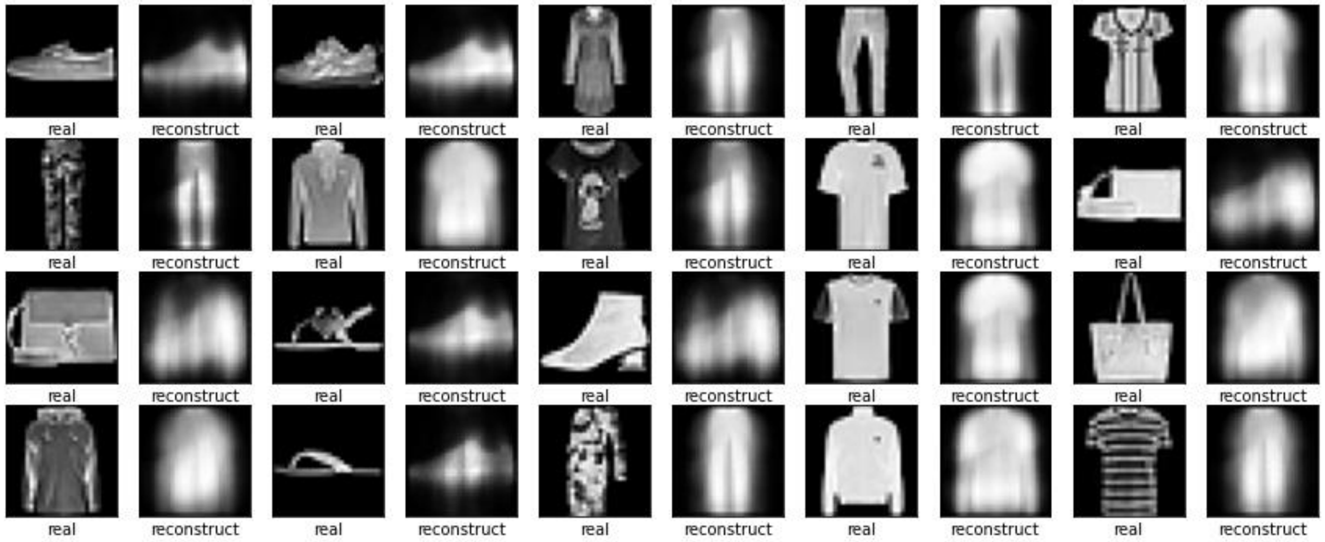
- **Sparse auto-encoder**
  - $\min_{\theta,\Phi} \mathbb{E}_{\boldsymbol{x}\sim\mathbb{P}} [d(\boldsymbol{x}, g_\Phi(f_\theta(\boldsymbol{x})))] + \lambda\Omega(\boldsymbol{z})$
  - $\Omega(\boldsymbol{z})$ is a regularization which is usually a norm over $\boldsymbol{z}$, $\lambda > 0$ **is** regularization parameter**.**

- Possible choices for **the regularization** $\Omega(z)$
  - **Norm-1:** $\Omega(\boldsymbol{z}) = \|\boldsymbol{z}\|_1 = \sum_{i=1}^{d} |z_i|$
  - **Norm-2:** $\Omega(\boldsymbol{z}) = \frac{1}{2}\|\boldsymbol{z}\|_2^2 = \frac{1}{2}\sum_{i=1}^{d} z_i^2$
  - **CE divergence:** $\Omega(\boldsymbol{z}) = d^{-1}\sum_{i=1}^{d} CE([z_i, 1 - z_i], [a, 1 - a]) = -\frac{1}{d}\sum_{i=1}^{d}[z_i \mathrm{log} a + (1 - z_i) log(1 - a)]$
    - $0 < a < 1$ is a very small number
    - CE is the **cross-entropy loss**

Implementation of Sparse Auto-Encoder

```
class SparseAE(GeneralAE):
    def __init__(self, optimizer = keras.optimizers.SGD(lr=0.1), regularizer = keras.regularizers.l1(l=0.01)):
        super(SparseAE, self).__init__(optimizer)
        self.regularizer = regularizer

    def build(self):
        self.encoder = keras.models.Sequential([keras.layers.Flatten(input_shape=[28, 28]),
                                                keras.layers.Dense(100, activation="selu"),
                                                keras.layers.Dense(30, activation="selu", activity_regularizer=self.regularizer)])

        self.decoder = keras.models.Sequential([keras.layers.Dense(100, activation="selu", input_shape=[30]),
                                                keras.layers.Dense(28 * 28, activation="sigmoid"),
                                                keras.layers.Reshape([28, 28])])

        self.auto_encoder = keras.models.Sequential([self.encoder, self.decoder])
        self.auto_encoder.compile(loss="binary_crossentropy", optimizer=self.optimizer, metrics=[GeneralAE.rounded_accuracy])
```

```
l1_sparse_ae.show_reconstructions(X_valid[:20], n_cols=5)
```

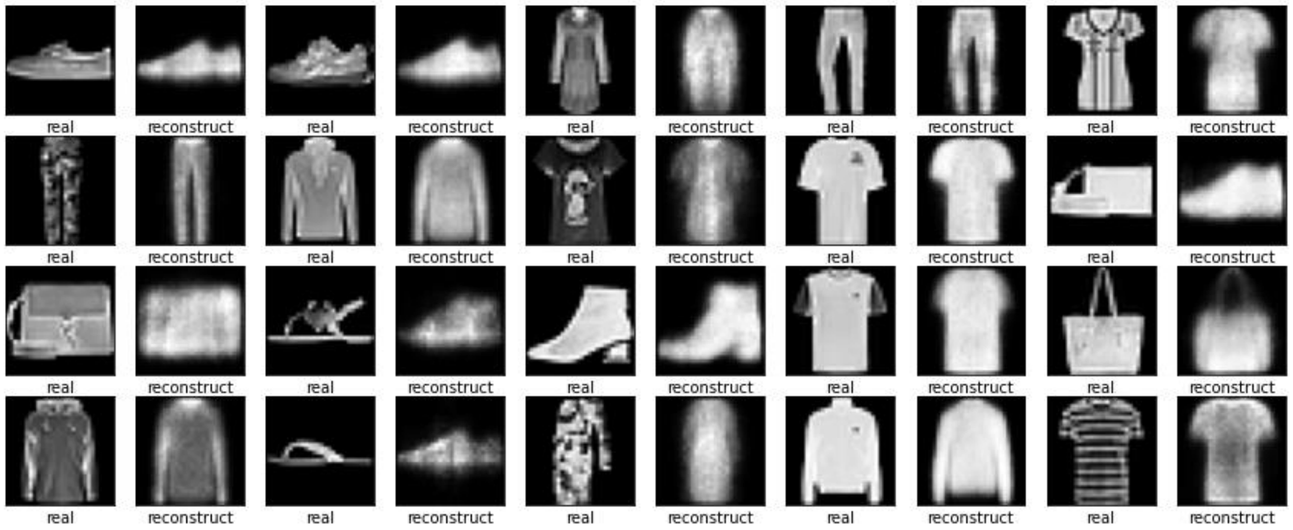


Implementation of Sparse Auto-Encoder

```
kl_divergence = keras.losses.kullback_leibler_divergence

class KLDivergenceRegularizer(keras.regularizers.Regularizer):
    def __init__(self, l, target=0.1):
        self.weight = l
        self.target = target
    def __call__(self, inputs):
        mean_activities = tf.reduce_mean(inputs, axis=0)
        return self.weight * (
            kl_divergence(self.target, mean_activities) + kl_divergence(1. - self.target, 1. - mean_activities))
```

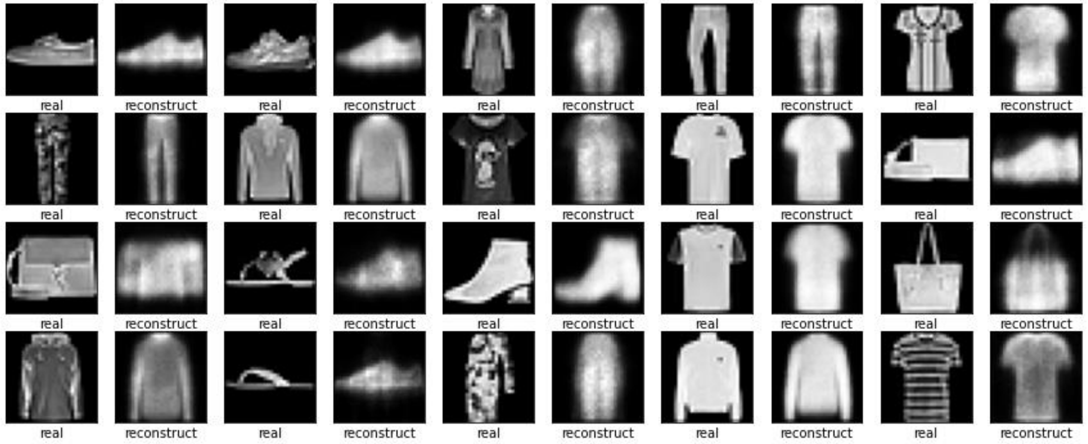
```
kl_sparse_ae = SparseAE(regularizer=KLDivergenceRegularizer(l=0.01))
kl_sparse_ae.build()
kl_sparse_ae.train(X_train, X_train, epochs=20, validation_data=(X_valid, X_valid))
```

```
kl_sparse_ae.show_reconstructions(X_valid[:20], n_cols=5)
```

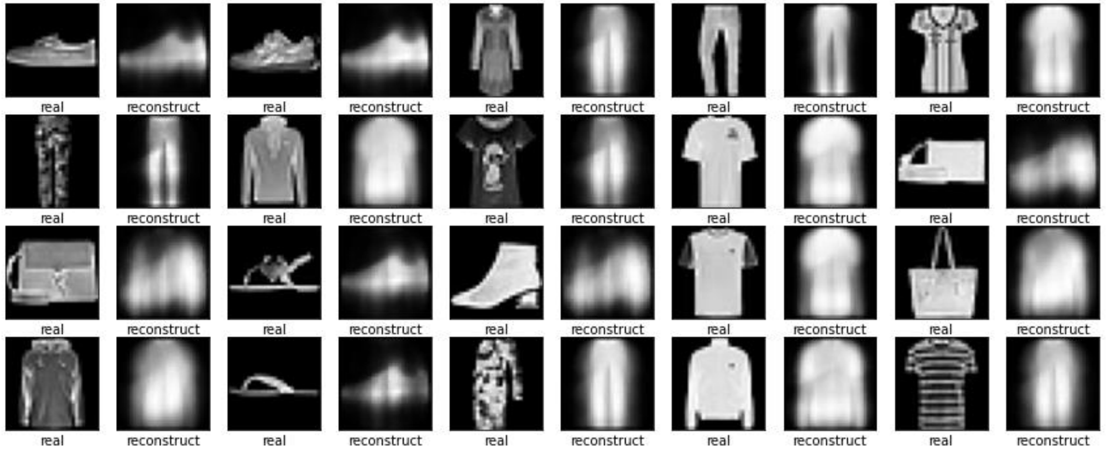


Sparse Auto-Encoder

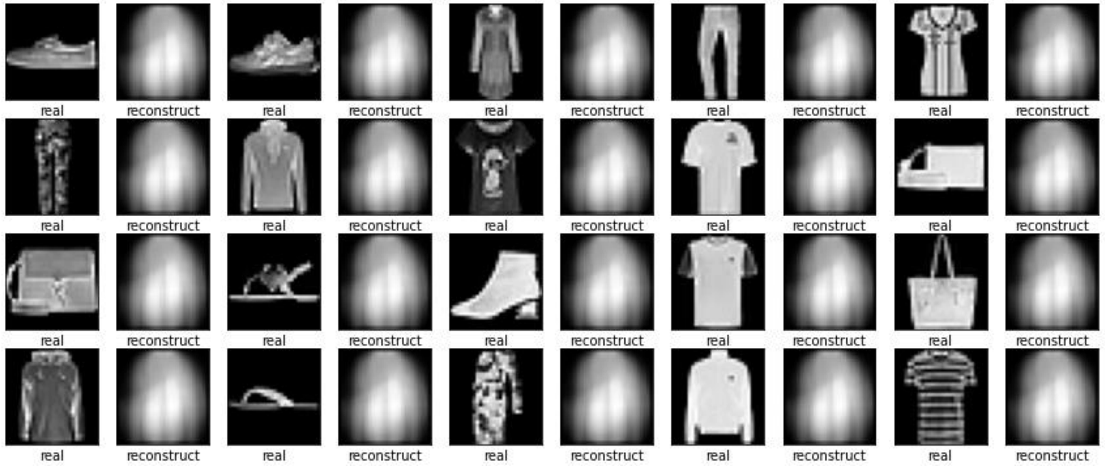
Effect of regularization parameter



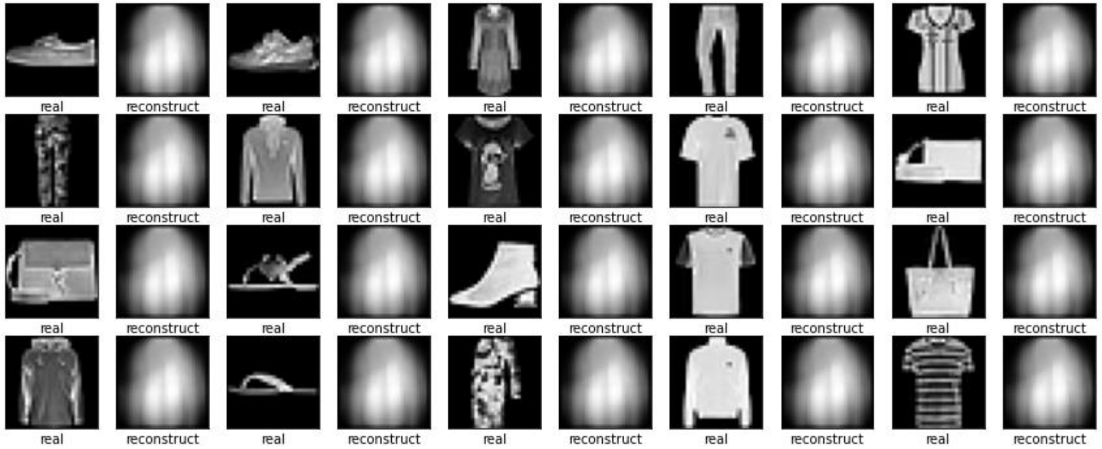
$\lambda = 0.001$



$\lambda = 0.01$



$\lambda = 0.05$



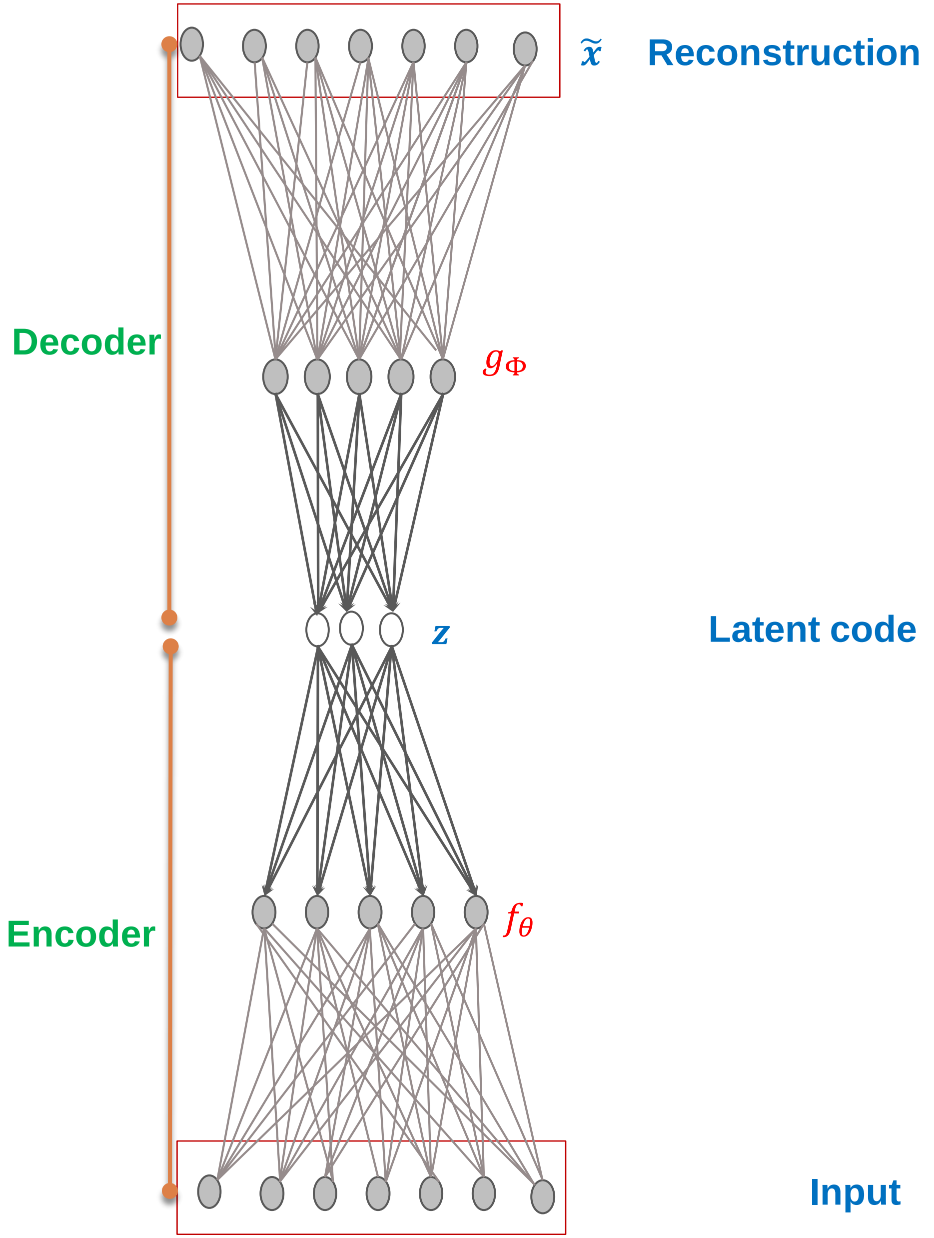
$\lambda = 10$

Contractive Auto-Encoder

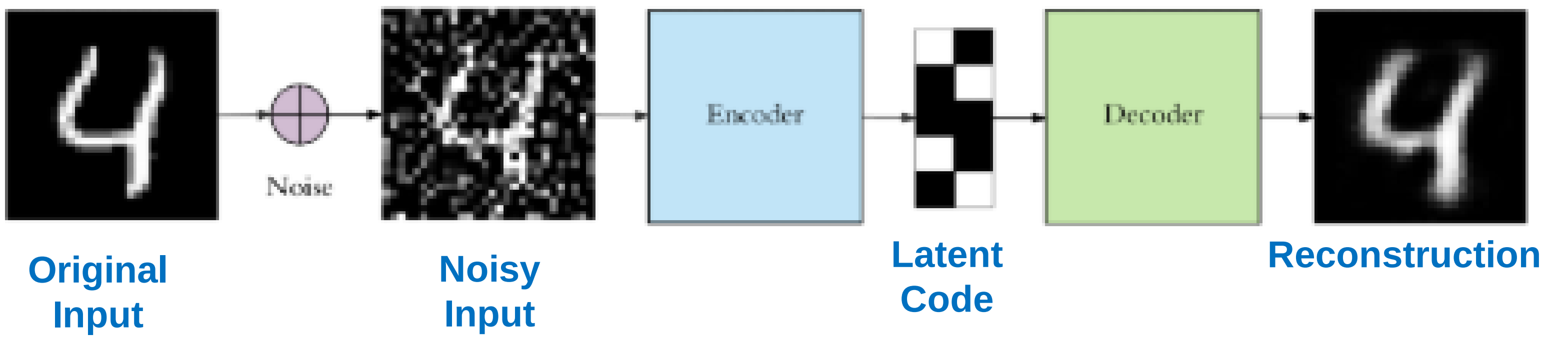
- **Regularized AE**:
  - $\min_{\theta,\Phi} \mathbb{E}_{\boldsymbol{x}\sim\mathbb{P}} [d(\boldsymbol{x}, g_\Phi(f_\theta(\boldsymbol{x}))] + \lambda\Omega(\boldsymbol{x},\boldsymbol{z})$

  where $\Omega(\boldsymbol{x},\boldsymbol{z}) = \sum_{i=1}^{m} \|\nabla_{\boldsymbol{x}} \boldsymbol{z}_i\|^2 = \left\|\frac{\partial f_\theta(x)}{\partial x}\right\|_F^2.$

- Hence, we train to resist the perturbations of the input by minimizing the magnitude of the gradient of the encoder f
  - This contracts the input neighbourhood to a smaller output neighbourhood, hence the name Contractive AE



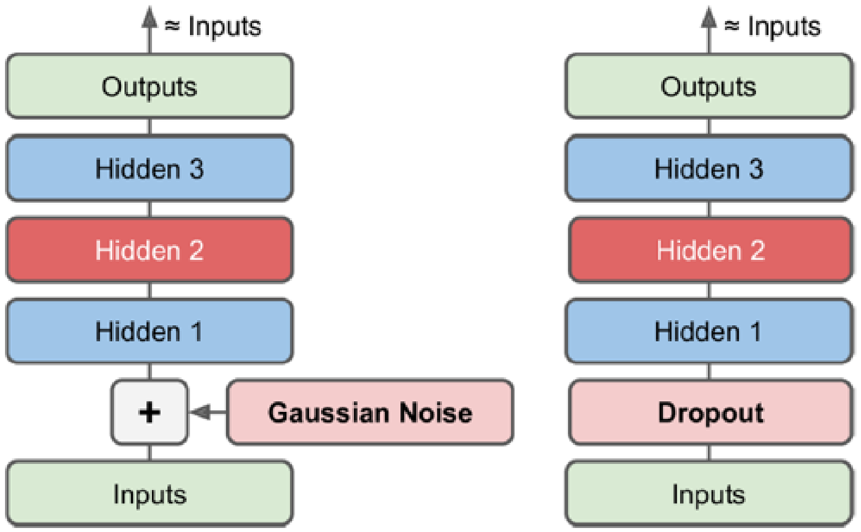
Denoising Auto-Encoder



- Add a small **Gaussian noise** to original input and require the auto-encoder to reconstruct the original input
  - $\boldsymbol{x}' = \boldsymbol{x} + \boldsymbol{\epsilon}$ where $\boldsymbol{\epsilon} \sim \boldsymbol{N}(\boldsymbol{0}, \eta \boldsymbol{I})$ and learn such that $\boldsymbol{g}_{\boldsymbol{\Phi}}\left(\boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{x}')\right) \approx \boldsymbol{x}$
- **Denoising auto-encoder**
  - $\min_{\theta, \Phi} \mathbb{E}_{\boldsymbol{x} \sim \mathbb{P}} \left[\mathbb{E}_{\boldsymbol{x}' \sim N(\boldsymbol{x}, \eta I)} [d(\boldsymbol{x}, g_{\Phi}(f_{\theta}(\boldsymbol{x}')))] \right]$

Implementation of Denoising Auto-Encoder

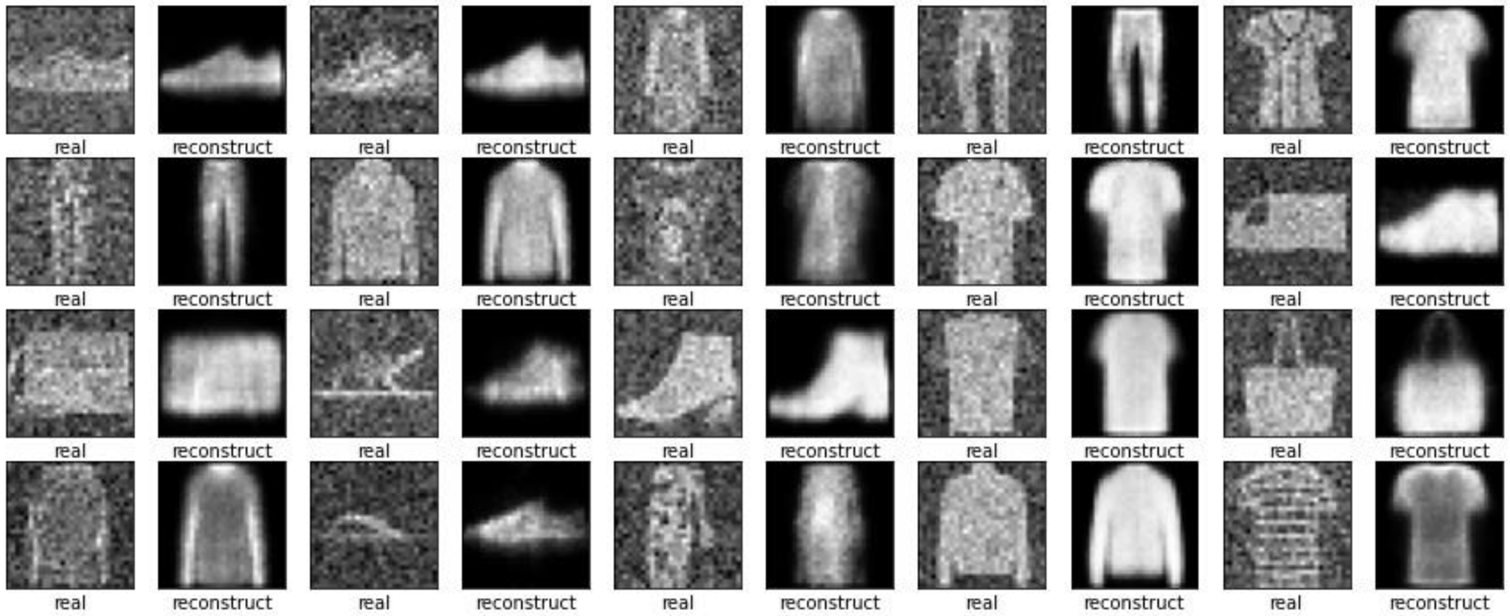
```
class DenoisingAE(GeneralAE):
    def __init__(self, optimizer = keras.optimizers.SGD(lr=0.1), noise= 0.2, seed= 6789):
        super(DenoisingAE, self).__init__(optimizer)
        self.noise = noise
        self.seed = seed
        tf.random.set_seed(self.seed)
        np.random.seed(self.seed)


    def build(self):
        self.encoder = keras.models.Sequential([keras.layers.Flatten(input_shape=[28, 28]),
                                                keras.layers.GaussianNoise(self.noise),
                                                keras.layers.Dense(100, activation="selu"),
                                                keras.layers.Dense(30, activation="selu")])

        self.decoder = keras.models.Sequential([keras.layers.Dense(100, activation="selu", input_shape=[30]),
                                                keras.layers.Dense(28 * 28, activation="sigmoid"),
                                                keras.layers.Reshape([28, 28])])

        self.auto_encoder = keras.models.Sequential([self.encoder, self.decoder])
        self.auto_encoder.compile(loss="binary_crossentropy", optimizer=self.optimizer, metrics=[GeneralAE.rounded_accuracy])
```



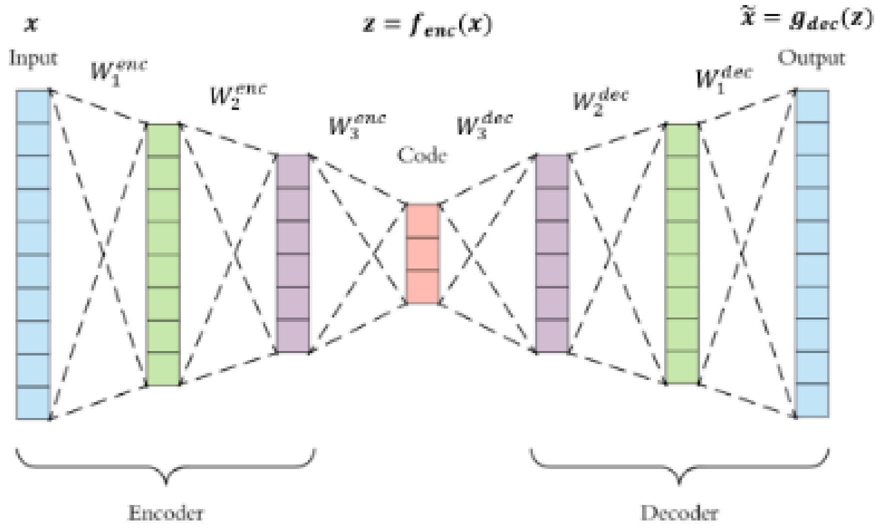
(Source: Hand-On, Ch15)

Implementation of Denoising Auto-Encoder

```
noise = keras.layers.GaussianNoise(0.2)
denoise_ae.show_reconstructions(noise(X_valid[0:20], training= True), n_cols=5)
```



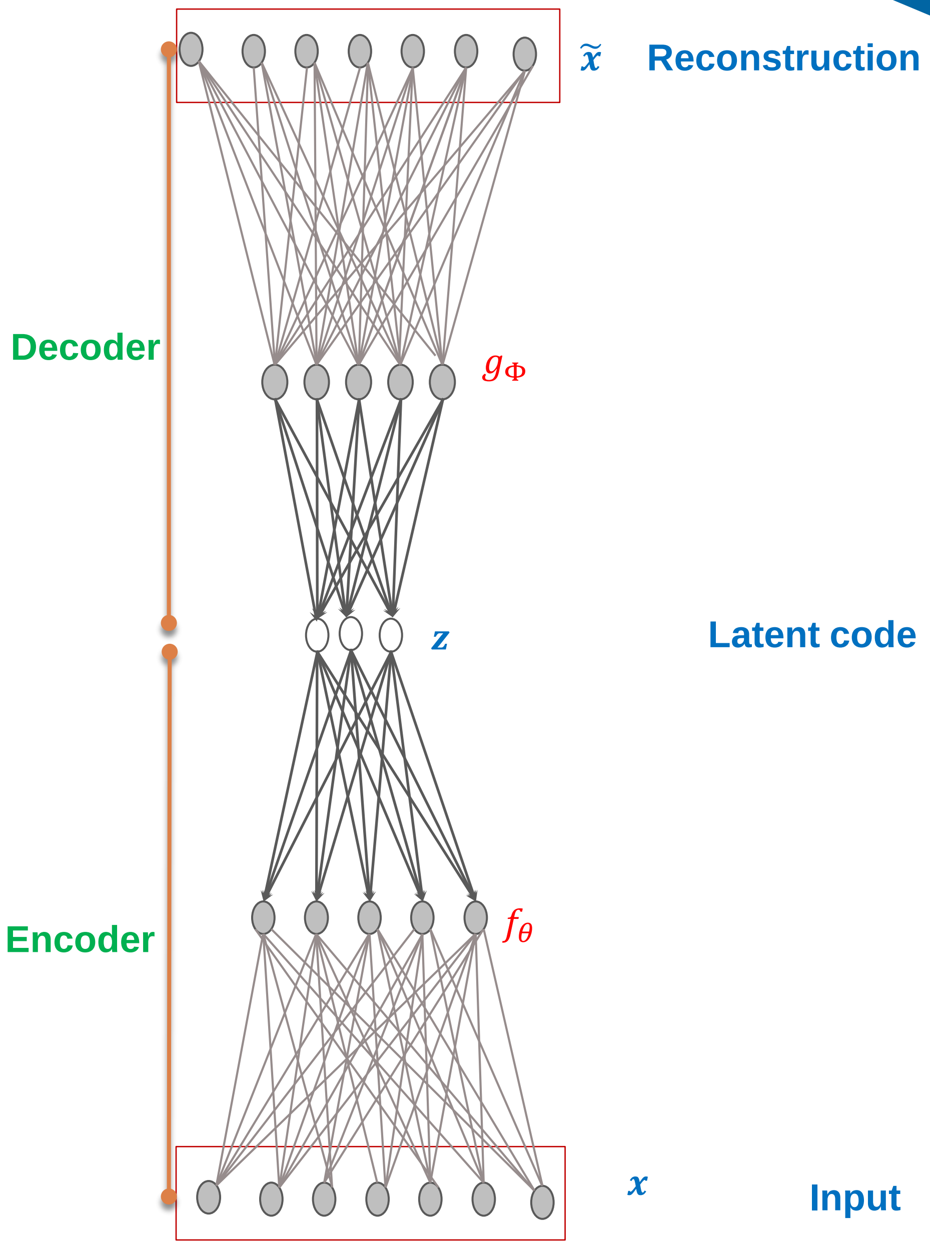
Tying encoders and decoders



- To reduce the **number of parameters** and put constraints on Auto-Encoder to avoid trivial models, we can tie encoders and decoders
  - $W_1^{enc} = \left(W_1^{dec}\right)^T$, $W_2^{enc} = \left(W_2^{dec}\right)^T$, and $W_3^{enc} = \left(W_3^{dec}\right)^T$

Representation power, size and depth

- We can build deep AE where both f and g are deep NNs
  - Hence, enjoying the power of deep NNs, especially the universal function approximation properties
  - i.e., even with single layer and one additional hidden layer for the encoder, AE can be very powerful if enough hidden units are given
  - However, without proper regularization, too powerful encoder and decoder are not necessarily good
- Depth can exponentially reduce computational cost and amount of training data in some cases
- Much better compression can be achieved with deep AE compared with shallow or linear AE (Hinton and Salakhutdinov, '06)




Q&A

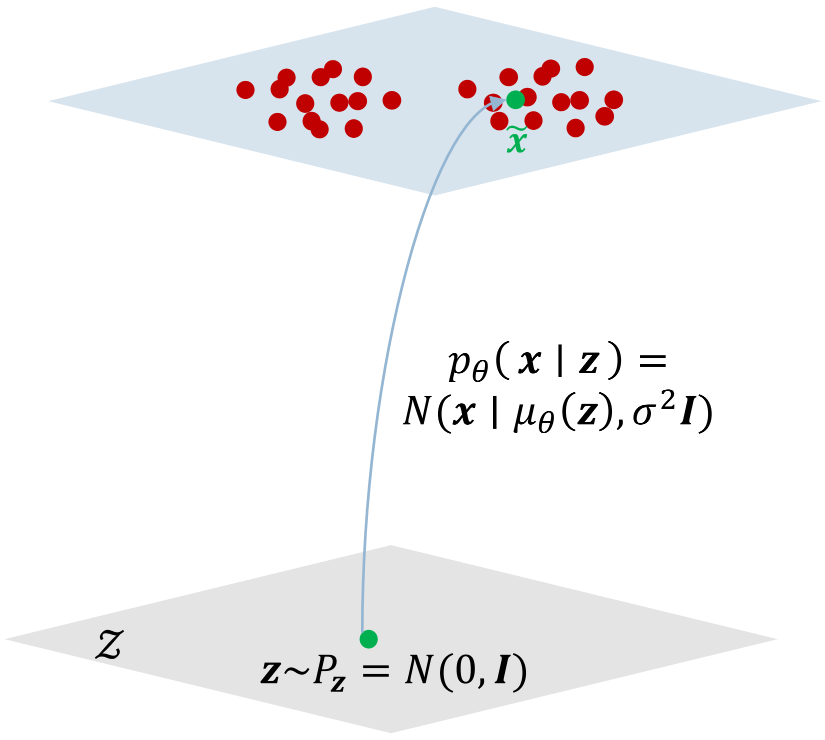
Stochastic Auto-Encoder (Variational Auto-Encoder)

Variational Auto-Encoder (VAE)

Generative model viewpoint

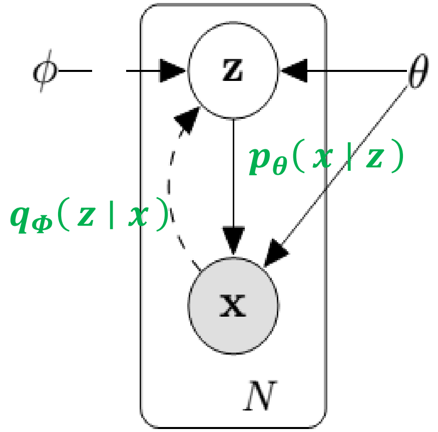
- Given a training set $D = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_N\}$ where each $\boldsymbol{x}_i \sim p_d(\boldsymbol{x})$.
  - $p_d(\boldsymbol{x})$ exists but unknown.

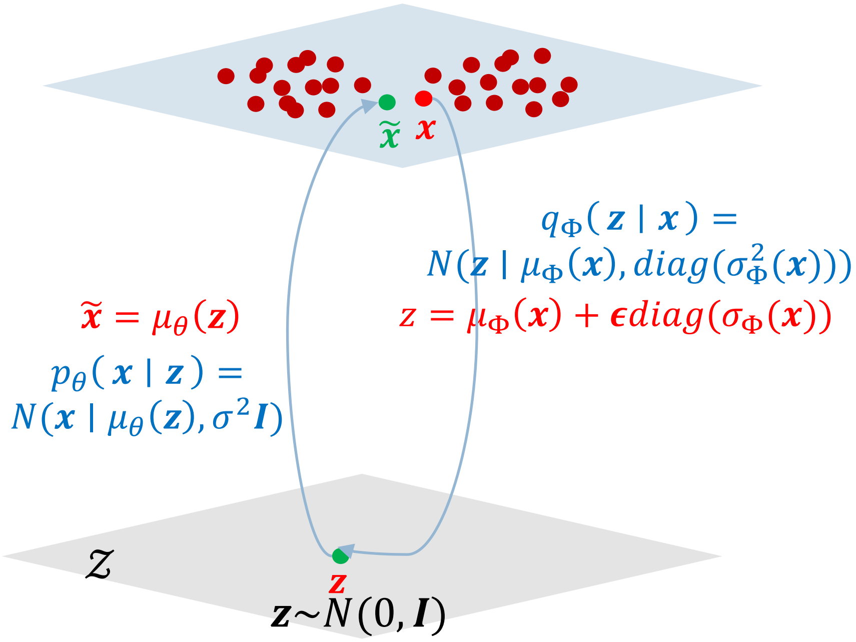
- Learn a stochastic decoder $p_\theta(\boldsymbol{x} \mid \boldsymbol{z}) = N(\boldsymbol{x} \mid \mu_\theta(\boldsymbol{z}), \sigma^2 \boldsymbol{I})$
  - $\boldsymbol{z} \sim p(\boldsymbol{z}) = N(0, \boldsymbol{I}) \rightarrow \widetilde{\boldsymbol{x}} \sim p_\theta(\boldsymbol{x} \mid \boldsymbol{z})$ or $\widetilde{\boldsymbol{x}} = \mu_\theta(\boldsymbol{z}) + \boldsymbol{\epsilon}\sigma\boldsymbol{I}$ with $\boldsymbol{\epsilon} \sim N(0, \boldsymbol{I})$
  - $\widetilde{\boldsymbol{x}}$ seems to **be sampled** from $p_d(\boldsymbol{x})$.
  - $\widetilde{\boldsymbol{x}}$ mimics true data samples in the training set D.



Variational Auto-Encoder (VAE)

- Use **stochastic encoder**
  - $q_{\Phi}(\boldsymbol{z} \mid \boldsymbol{x}) = N(\boldsymbol{z} \mid \mu_{\Phi}(\boldsymbol{x}), diag(\sigma_{\Phi}^2(\boldsymbol{x}))$
  - $z = \mu_{\Phi}(\boldsymbol{x}) + \boldsymbol{\epsilon} diag(\sigma_{\Phi}(\boldsymbol{x}))$ with $\boldsymbol{\epsilon} \sim N(0, \boldsymbol{I})$
- Use **stochastic decoder**
  - $p_{\theta}(\boldsymbol{x} \mid \boldsymbol{z}) = N(\boldsymbol{x} \mid \mu_{\theta}(\boldsymbol{z}), \sigma^2 \boldsymbol{I})$
  - $\widetilde{\boldsymbol{x}} = \mu_{\theta}(\boldsymbol{z})$



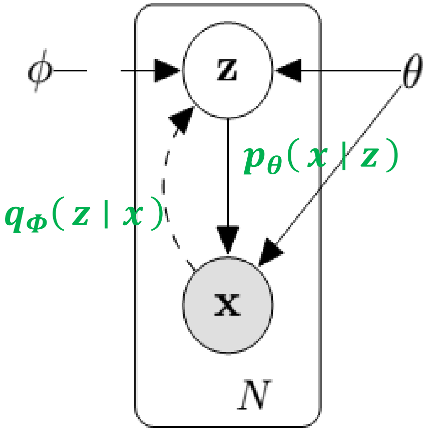


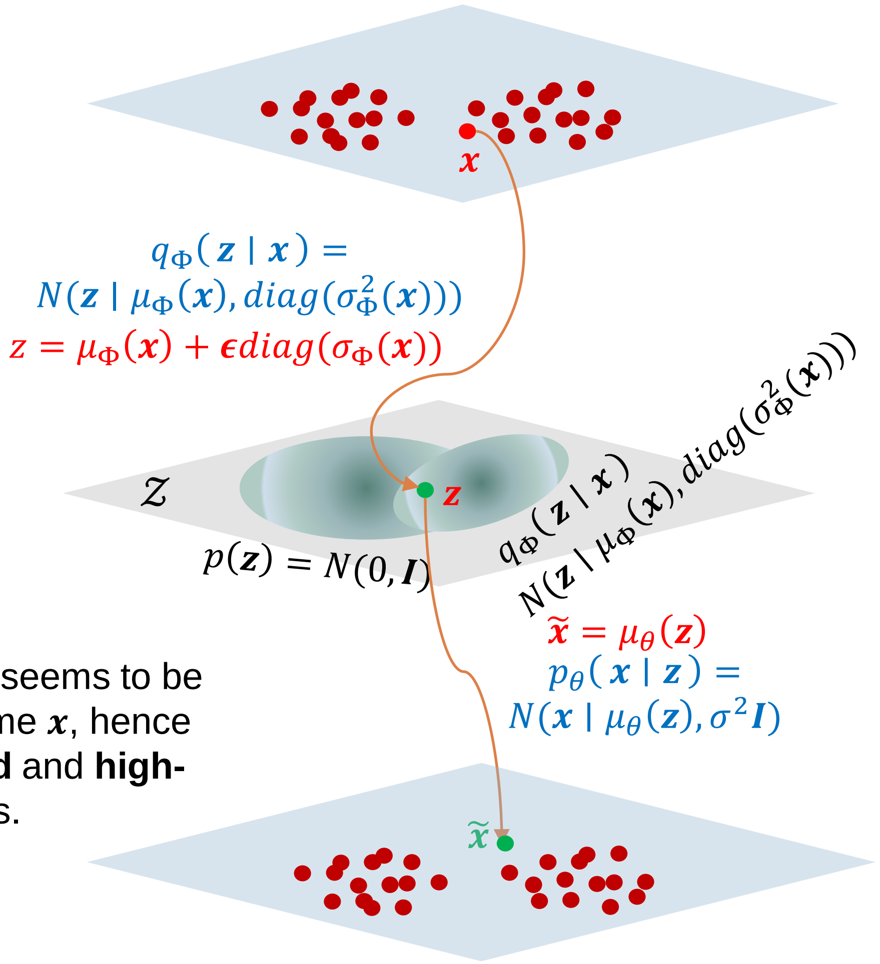
Variational Auto-Encoder (VAE)

- Use **stochastic encoder**
  - $q_\Phi(\boldsymbol{z} \mid \boldsymbol{x}) = N(\boldsymbol{z} \mid \mu_\Phi(\boldsymbol{x}), diag(\sigma_\Phi^2(\boldsymbol{x}))$
  - $z = \mu_\Phi(\boldsymbol{x}) + \boldsymbol{\epsilon} diag(\sigma_\Phi(\boldsymbol{x}))$ with $\boldsymbol{\epsilon} \sim N(0,\boldsymbol{I})$

- Use **stochastic decoder**
  - $p_\theta(\boldsymbol{x} \mid \boldsymbol{z}) = N(\boldsymbol{x} \mid \mu_\theta(\boldsymbol{z}), \sigma^2 \boldsymbol{I})$
  - $\tilde{\boldsymbol{x}} = \mu_\theta(\boldsymbol{z})$

- **Objective function**
  - Reconstruction:
    - $\max_{\theta,\Phi} \mathbb{E}_{\boldsymbol{x}}[\log p_\theta(\boldsymbol{x} \mid \boldsymbol{z})] \leftrightarrow \min_{\theta,\Phi} \frac{1}{2\sigma^2} \mathbb{E}_{\boldsymbol{x}}[d(\boldsymbol{x}, \mu_\theta(\boldsymbol{z}))]$ for $\boldsymbol{z} = \mu_\Phi(\boldsymbol{x}) + \boldsymbol{\epsilon} diag\left(\sigma_\Phi(\boldsymbol{x})\right)^{1/2}$.
  - Matching to the prior $p(z) = N(\boldsymbol{0}, \boldsymbol{I})$
    - $\min_{\theta,\Phi} \mathbb{E}_{\boldsymbol{x}}[KL(q_\Phi(\boldsymbol{z} \mid \boldsymbol{x}) \| N(\boldsymbol{0}, \boldsymbol{I}))] = \frac{1}{2} \min_{\theta,\Phi} \mathbb{E}_{\boldsymbol{x}}\left[-d + \|\mu_\phi(\boldsymbol{x})\|_2^2 + \|\sigma_\phi(\boldsymbol{x})\|_2^2 - sum(\log[\sigma_\phi(\boldsymbol{x})^2])\right]$

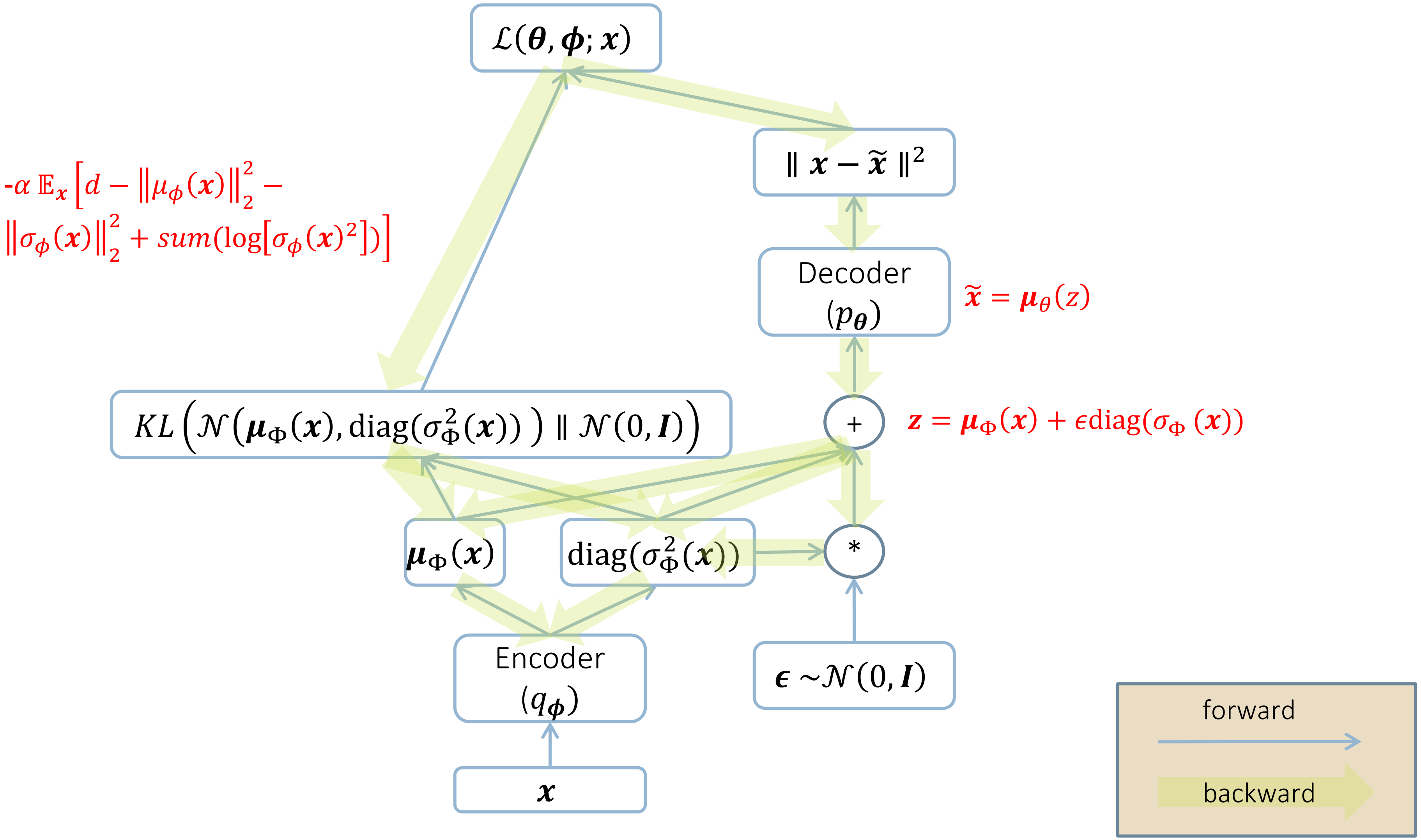




- **The final objective function**
  - $\min_{\theta,\Phi} \mathbb{E}_{\boldsymbol{x}}\left[\mathbb{E}_\epsilon\left[\frac{1}{\sigma^2} d(\boldsymbol{x}, \mu_\theta(\boldsymbol{z})) + \|\mu_\phi(\boldsymbol{x})\|^2 + \|\sigma_\phi(\boldsymbol{x})\|^2 - sum(\log[\sigma_\phi(\boldsymbol{x})^2])\right]\right]$ with $\boldsymbol{z} = \mu_\Phi(\boldsymbol{x}) + \boldsymbol{\epsilon} diag(\sigma_\Phi(\boldsymbol{x}))^{\frac{1}{2}}$ and $\boldsymbol{\epsilon} \sim N(0,\boldsymbol{I})$.

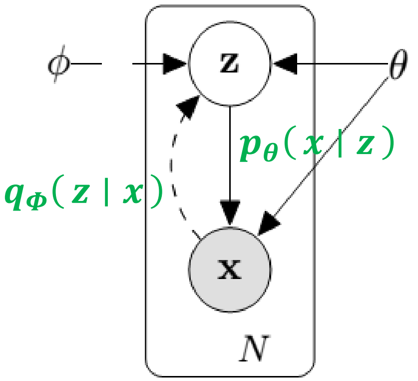
By **matching** $q_\Phi(\boldsymbol{z} \mid \boldsymbol{x})$ and the prior $N(0, \boldsymbol{I})$, $z \sim N(0, \boldsymbol{I})$ seems to be sampled from $q_\Phi(\boldsymbol{z} \mid \boldsymbol{x})$ for some $\boldsymbol{x}$, hence leading to $\tilde{\boldsymbol{x}} = \mu_\theta(\boldsymbol{z})$ to be **good** and **high-quality** reconstructed examples.

VAE computational graph



Not in assessment

VAE Formal Derivation



Auto-Encoding Variational Bayes

Diederik P. Kingma
Machine Learning Group
Universiteit van Amsterdam
dpkingma@gmail.com

Max Welling
Machine Learning Group
Universiteit van Amsterdam
welling.max@gmail.com

Paper: https://arxiv.org/pdf/1312.6114.pdf

- $\log p_\theta(x) = \log \frac{p_\theta(x,z)}{p_\theta(z|x)} = \log \frac{p_\theta(x,z)}{q_\Phi(z|x)} \frac{q_\Phi(z|x)}{p_\theta(z|x)} = \log \frac{p_\theta(x,z)}{q_\Phi(z|x)} + \log \frac{q_\Phi(z|x)}{p_\theta(z|x)}$
- $\mathbb{E}_{x\sim p_d(x)}[\log p_\theta(x)] = \mathbb{E}_{x\sim p_d(x), z\sim q_\Phi(z|x)}[\log p_\theta(x)] = \mathbb{E}_{x\sim p_d(x), z\sim q_\Phi(z|x)}\left[\log \frac{p_\theta(x,z)}{q_\Phi(z|x)}\right] + \mathbb{E}_{x\sim p_d(x), z\sim q_\Phi(z|x)}\left[\log \frac{q_\Phi(z|x)}{p_\theta(z|x)}\right] =$
  $\mathbb{E}_{x\sim p_d(x), z\sim q_\Phi(z|x)}\left[\log \frac{p_\theta(x,z)}{q_\Phi(z|x)}\right] + \mathbb{E}_x\left[\underbrace{\mathbb{E}_{z\sim q_\Phi(z|x)}\left[\log \frac{q_\Phi(z|x)}{p_\theta(z|x)}\right]}_{KL(q_\Phi(z \mid x) \| p_\theta(z \mid x)) \geq 0}\right] \geq \mathbb{E}_{x\sim p_d(x), z\sim q_\Phi(z|x)}\left[\log \frac{p_\theta(x,z)}{q_\Phi(z|x)}\right].$
- $\mathbb{E}_{x\sim p_d(x)}[\log p_\theta(x)] \geq \mathbb{E}_{x\sim p_d(x), z\sim q_\Phi(z|x)}\left[\log \frac{p_\theta(x,z)}{q_\Phi(z|x)}\right] = \mathbb{E}_x\left[\mathbb{E}_z\left[\log \frac{p_\theta(x|z)p(z)}{q_\Phi(z|x)}\right]\right] = \mathbb{E}_x[\mathbb{E}_z[\log p_\theta(x \mid z)]] + \mathbb{E}_x\left[\mathbb{E}_z\left[\log \frac{p(z)}{q_\Phi(z|x)}\right]\right] =$
  $\mathbb{E}_x[\mathbb{E}_z[\log p_\theta(x \mid z)]] - \mathbb{E}_x[KL(q_\Phi(z \mid x) \| N(\mathbf{0}, \boldsymbol{I}))] = \mathbb{E}_x\left[\mathbb{E}_{\epsilon\sim N(0,I)}[\log p_\theta(x \mid z)]\right] - \mathbb{E}_x[KL(q_\Phi(z \mid x) \| N(\mathbf{0}, \boldsymbol{I}))].$
  - Use reparameterization trick: $\mathbb{E}_x[\mathbb{E}_z[\log p_\theta(x \mid z)]] = \mathbb{E}_x\left[\mathbb{E}_{\epsilon\sim N(0,I)}[\log p_\theta(x \mid z)]\right]$ where $z = \mu_\Phi(\boldsymbol{x}) + \boldsymbol{\epsilon} diag(\sigma_\Phi(\boldsymbol{x}))^{\frac{1}{2}}$.
- The final objective function

$$\max_{\theta,\Phi}\left\{\mathbb{E}_x\left[\mathbb{E}_{\epsilon\sim N(0,I)}[\log p_\theta(x \mid z)]\right] - \mathbb{E}_x[KL(q_\Phi(z \mid x) \| N(\mathbf{0}, \boldsymbol{I}))]\right\}$$

$$\max_{\theta,\Phi}\left\{\mathbb{E}_x\left[\mathbb{E}_{\epsilon\sim N(0,I)}\left[\frac{-\|x - \mu_\Phi(z)\|^2}{2\sigma^2}\right]\right] - \mathbb{E}_x[KL(q_\Phi(z \mid x) \| N(\mathbf{0}, \boldsymbol{I}))]\right\}$$

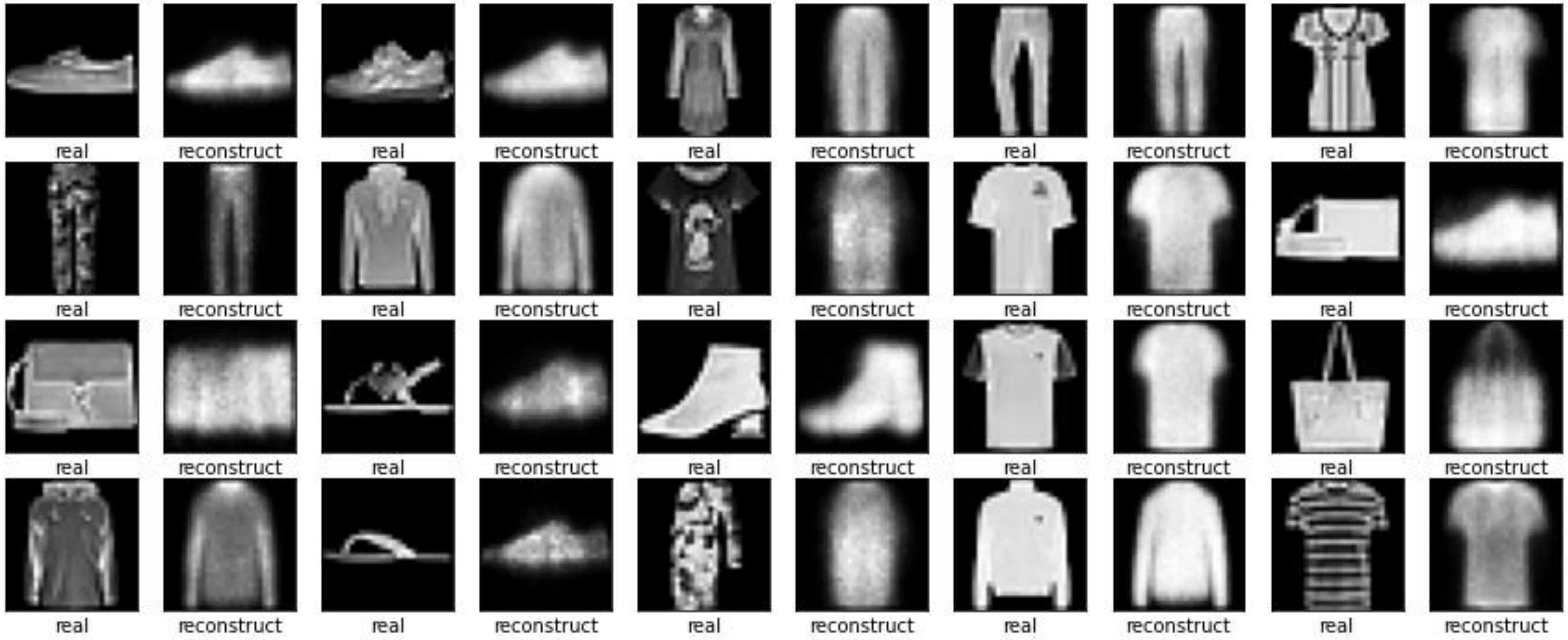
Implementation of VAE

```
class VariationalAE(GeneralAE):
    def __init__(self, optimizer = keras.optimizers.SGD(lr=0.1), alpha= 0.5, seed= 6789):
        super(VariationalAE, self).__init__(optimizer)
        self.alpha = alpha
        self.seed = seed
        tf.random.set_seed(self.seed)
        np.random.seed(self.seed)

    def build(self):
        codings_size = 10
        inputs = keras.layers.Input(shape=[28, 28])
        z = keras.layers.Flatten()(inputs)
        z = keras.layers.Dense(150, activation="selu")(z)
        z = keras.layers.Dense(100, activation="selu")(z)
        codings_mean = keras.layers.Dense(codings_size)(z)
        codings_log_var = keras.layers.Dense(codings_size)(z)
        codings = Sampling()([codings_mean, codings_log_var])
        self.encoder = keras.models.Model(inputs=[inputs], outputs=[codings_mean, codings_log_var, codings])

        decoder_inputs = keras.layers.Input(shape=[codings_size])
        x = keras.layers.Dense(100, activation="selu")(decoder_inputs)
        x = keras.layers.Dense(150, activation="selu")(x)
        x = keras.layers.Dense(28 * 28, activation="sigmoid")(x)
        outputs = keras.layers.Reshape([28, 28])(x)
        self.decoder = keras.models.Model(inputs=[decoder_inputs], outputs=[outputs])

        _, _, codings = self.encoder(inputs)
        reconstructions = self.decoder(codings)
        self.auto_encoder = keras.models.Model(inputs=[inputs], outputs=[reconstructions])

        latent_loss = -self.alpha * tf.reduce_sum(1 + codings_log_var - tf.math.exp(codings_log_var) - tf.math.square(codings_mean),axis=-1)
        self.auto_encoder.add_loss(tf.math.reduce_mean(latent_loss) / 784.)
        self.auto_encoder.compile(loss="binary_crossentropy", optimizer=self.optimizer, metrics=[GeneralAE.rounded_accuracy])
```

```
class Sampling(keras.layers.Layer):
    def call(self, inputs):
        mean, log_var = inputs
        return tf.random.normal(tf.shape(log_var)) * tf.math.exp(log_var / 2) + mean
```

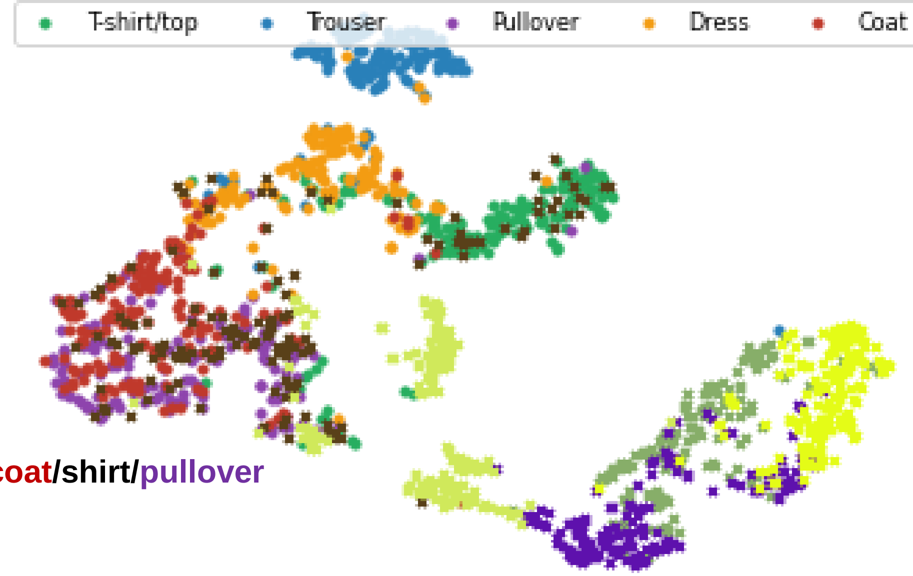
Implementation of VAE

```
vae.show_reconstructions(X_valid[:20], n_cols=5)
```



VAE latent codes with TSNE

```
visualize_latent_codes(X_valid[0:1500], y_valid[0:1500], vae)
```

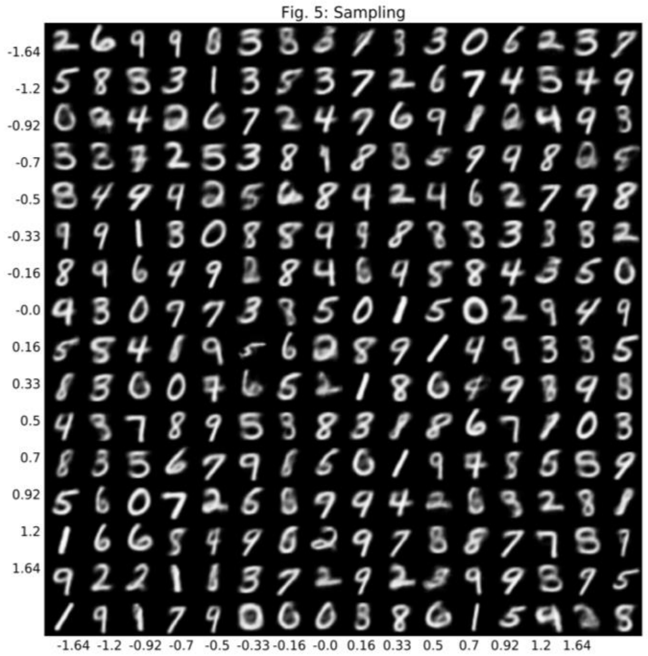


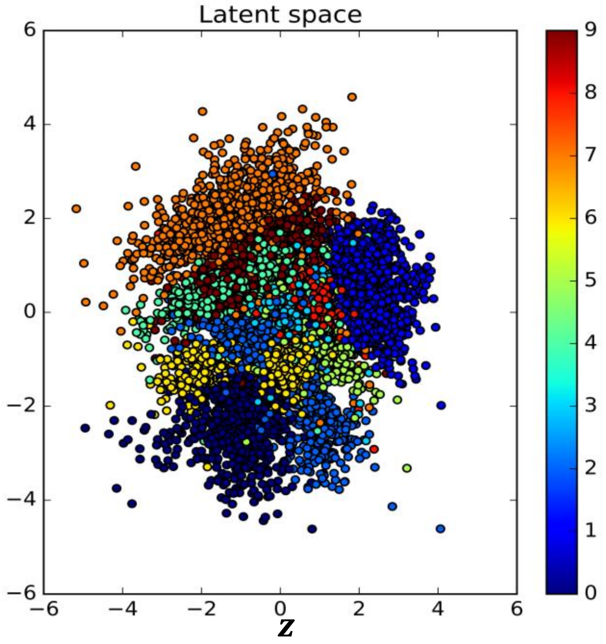
coat/shirt/pullover

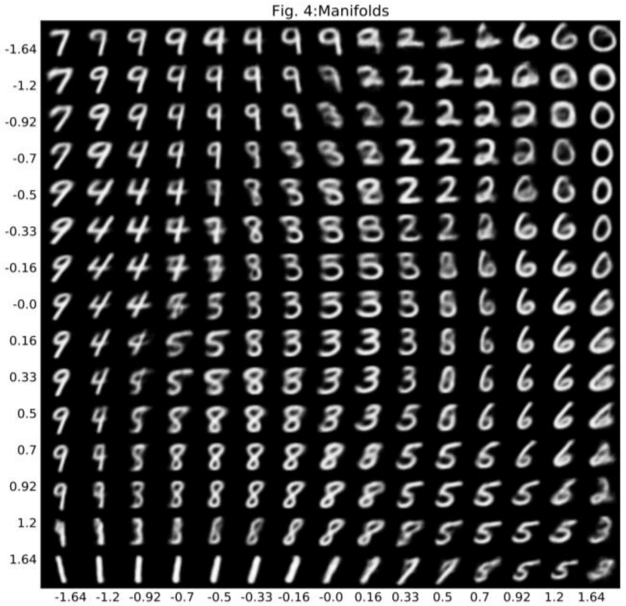
sandal/sneaker/ankle boot

Variational Auto-encoders (VAE)

- Run VAE on MNIST data with 2-dimensional latent variable $\boldsymbol{z}$







sampling

$p(x \mid z)$

$q(z \mid x)$

sampling on manifold

Variational Auto-encoders (VAE)

- VAE weakness: tend to generate blurry images on complex images
- Why? It captures global structures well, but has difficult with modelling the local information – GAN will be able to address this!

Summary

- Revision of some basic knowledge
- Learning efficient representations
- Auto-Encoder
  - Standard Auto-Encoder, Sparse, Contractive, Denoising Auto-Encoders
- Stochastic Auto-Encoder
  - Variational Auto-Encoder

Thanks for your attention!

Question time