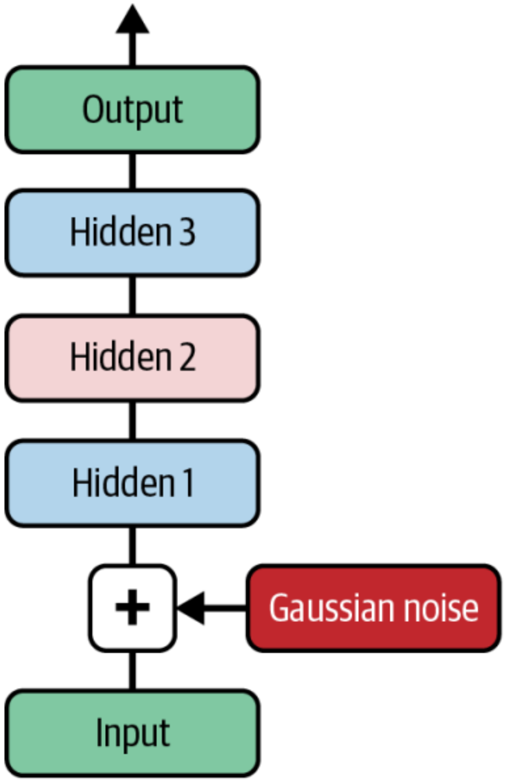
Denoising Autoencoders

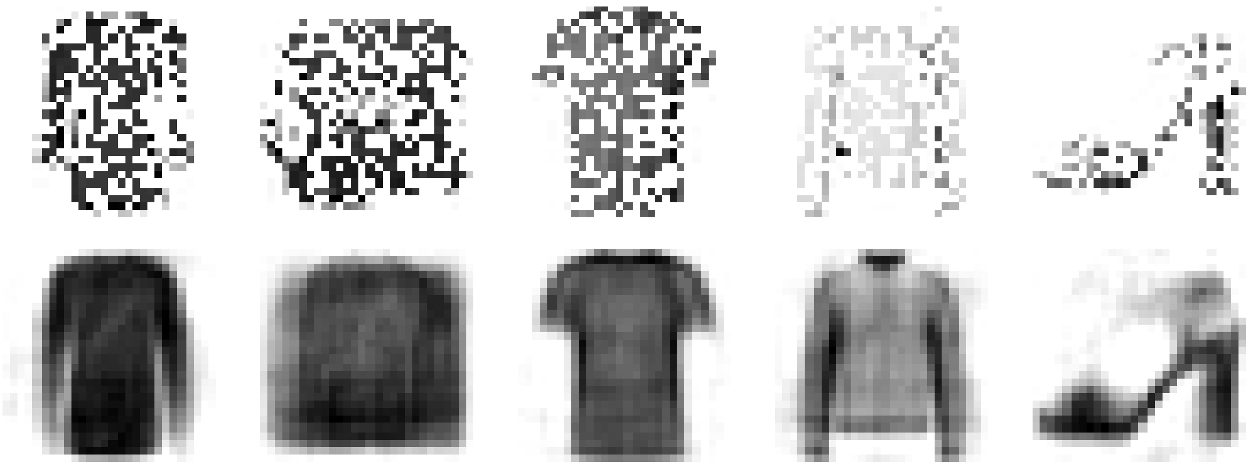


Denoising Autoencoders

```
dropout_encoder = tf.keras.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(30, activation="relu")
])
dropout_decoder = tf.keras.Sequential([
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(28 * 28),
    tf.keras.layers.Reshape([28, 28])
])
dropout_ae = tf.keras.Sequential([dropout_encoder, dropout_decoder])
```

- You can replace the Dropout layer with `tf.keras.layers.GaussianNoise(0.2)`.

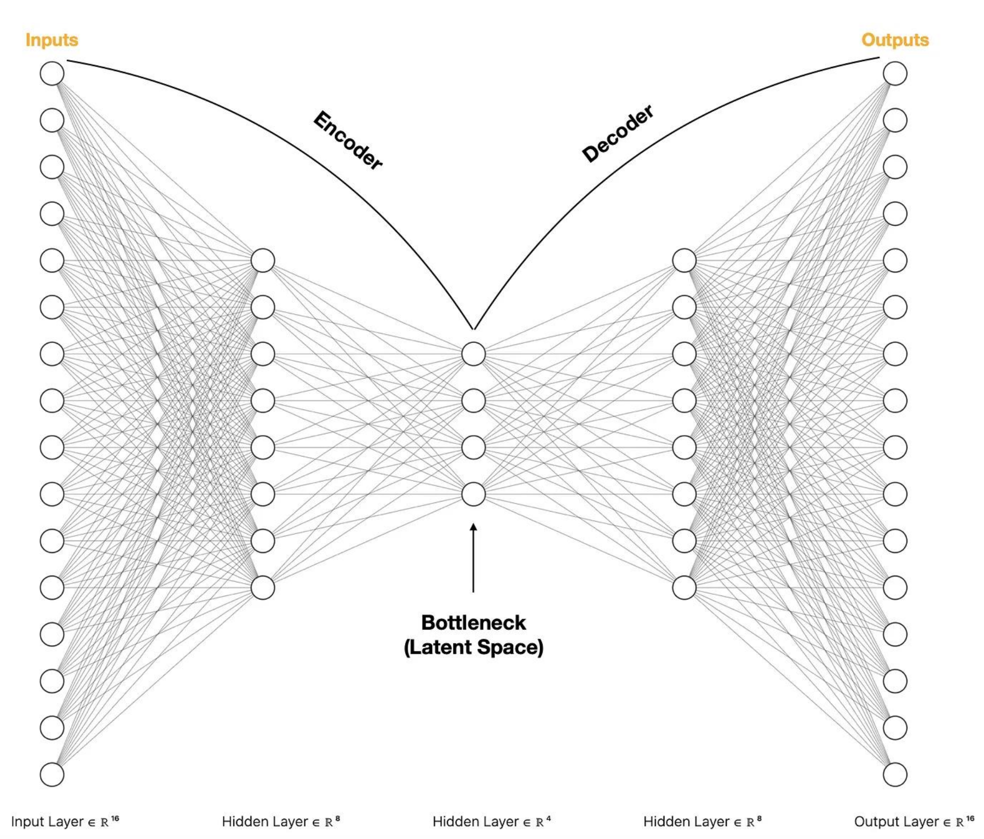
Denoising Autoencoders

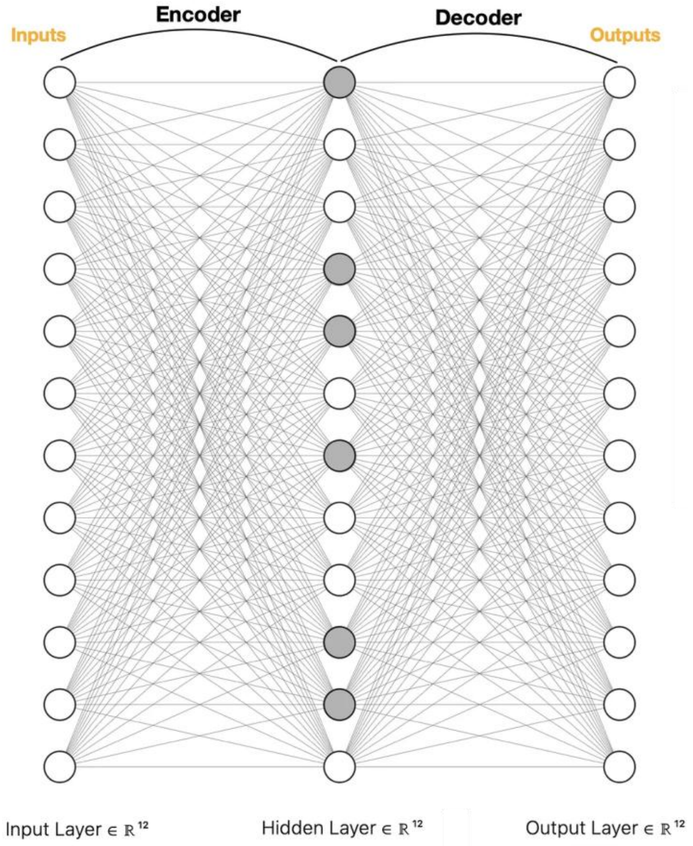


Sparse Autoencoders

- *Sparsity* constraint often leads to good feature extraction: by adding an appropriate term to the cost function, the autoencoder is pushed to reduce the number of active neurons in the coding layer.
  - For example, it may be pushed to have on average only 5% significantly active neurons in the coding layer.
- This forces the autoencoder to represent each input as a combination of a small number of activations.
  - Each neuron in the coding layer typically ends up representing a useful feature.
- Approach 1: use the sigmoid activation in the coding layer (to constrain the codings to values between 0 and 1), use a large coding layer, and add some $\ell 1$ regularization to the coding layer's activations.

Sparse Autoencoders





Sparse Autoencoders

```
sparse_l1_encoder = tf.keras.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(300, activation="sigmoid"),
    tf.keras.layers.ActivityRegularization(l1=1e-4)
])
sparse_l1_decoder = tf.keras.Sequential([
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(28 * 28),
    tf.keras.layers.Reshape([28, 28])
])
sparse_l1_ae = tf.keras.Sequential([sparse_l1_encoder, sparse_l1_decoder])
```

$$L(x, \hat{x}) + \lambda \sum_i |a_i^{(h)}|$$

Sparse Autoencoders

- Approach 2: measure the actual sparsity of the coding layer at each training iteration, and penalize the model when the measured sparsity differs from a target value.
  - We do so by computing the average activation of each neuron in the coding layer, over the whole training batch.
    - The batch size must not be too small, or else the mean will not be accurate.
  - Use the mean activation per neuron, to penalize the neurons that are too active, or not active enough, by adding a *sparsity loss* to the cost function.
  - Example: if a neuron has an average activation of 0.3, but the target sparsity is 0.1, it must be penalized to activate less.

Sparsity Loss

- The sparsity loss could be the squared error (e.g. $(0.3-0.1)^2$), but in practice a it is better is to use the Kullback–Leibler (KL) divergence.

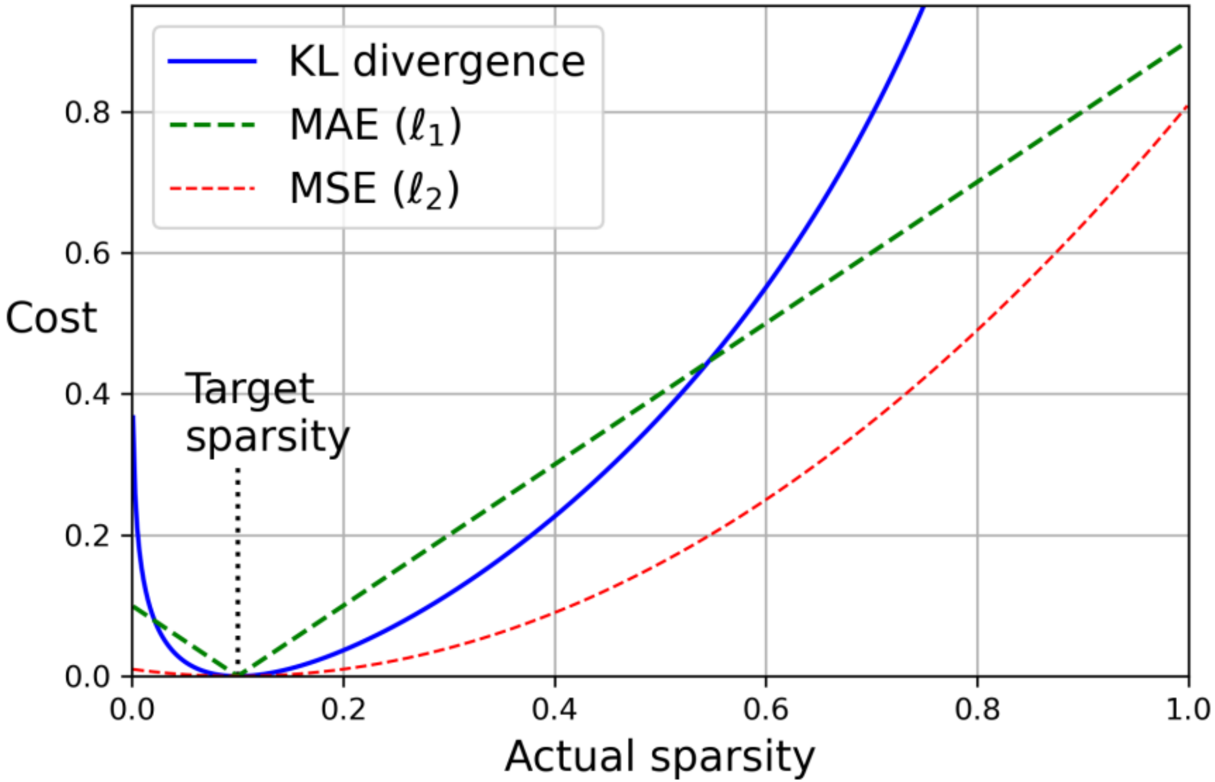
- The KL divergence between the target probability p that a neuron in the coding layer will activate and the actual probability q, estimated by measuring the mean activation over the training batch, is:

$$D_{KL}(p \parallel q) = p \log \frac{p}{q} + (1-p) \log \frac{1-p}{1-q}$$

- After computing the sparsity loss for each neuron in the coding layer, we sum up these losses and add the result to the cost function:

$$L(x, \hat{x}) + \sum_j D_{KL}(p \parallel q_j)$$

Sparsity Loss



KL Divergence Regularization

```
kl_divergence = tf.keras.losses.kullback_leibler_divergence

class KLDivergenceRegularizer(tf.keras.regularizers.Regularizer):
    def __init__(self, weight, target):
        self.weight = weight
        self.target = target

    def __call__(self, inputs):
        mean_activities = tf.reduce_mean(inputs, axis=0)
        return self.weight * kl_divergence(self.target, mean_activities)
```

Sparse Autoencoders

```
kld_reg = KLDivergenceRegularizer(weight=5e-3, target=0.1)
sparse_kl_encoder = tf.keras.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(300, activation="sigmoid",
                          activity_regularizer=kld_reg)
])
sparse_kl_decoder = tf.keras.Sequential([
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(28 * 28),
    tf.keras.layers.Reshape([28, 28])
])
sparse_kl_ae = tf.keras.Sequential([sparse_kl_encoder, sparse_kl_decoder])
```

Variational Autoencoders

- *Variational autoencoders* (VAEs) were introduced in 2013 by Kingma and Welling and are different from all the autoencoders in these particular ways:
  - They are *probabilistic autoencoders*, meaning that their outputs are partly determined by chance, even after training (as opposed to denoising autoencoders, which use randomness only during training).
  - They are *generative autoencoders*, meaning that they can generate new instances that look like they were sampled from the training set.

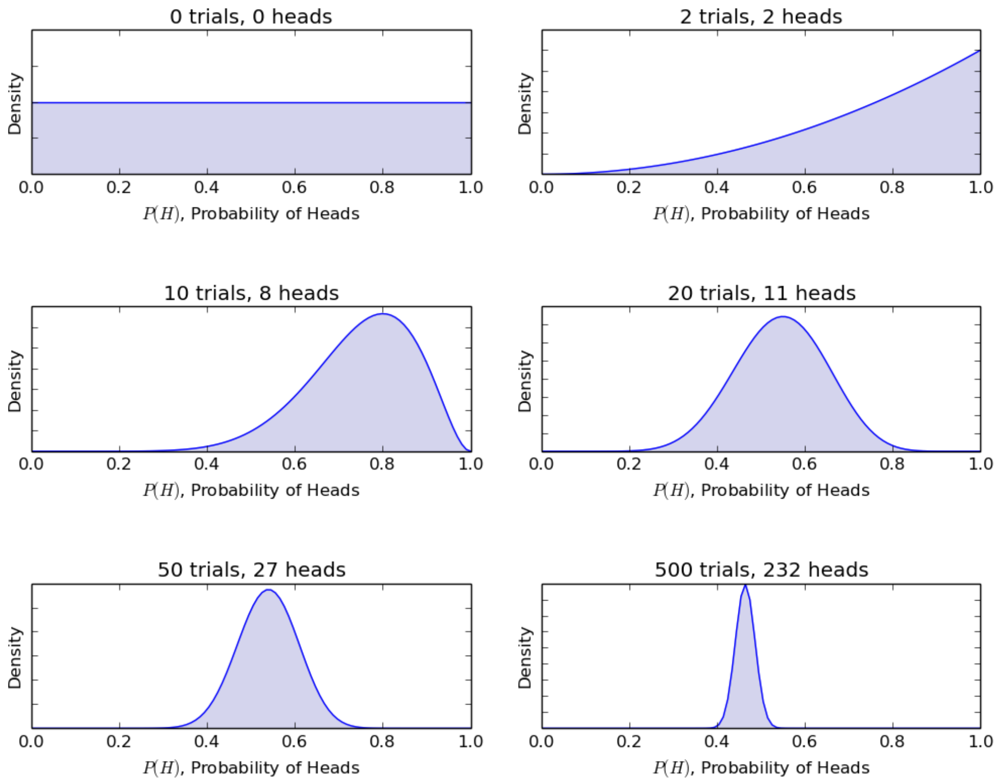
- These properties make VAEs rather similar to RBMs, but they are easier to train, and the sampling process is much faster.

Bayesian Inference

- Variational autoencoders perform variational Bayesian inference, which is an efficient way of carrying out approximate Bayesian inference.
- Bayesian inference means updating a probability distribution based on new data, using equations derived from Bayes' theorem.
- The original distribution is called the *prior*, while the updated distribution is called the *posterior*.
- In our case, we want to find a good approximation of the data distribution. Once we have that, we can sample from it.

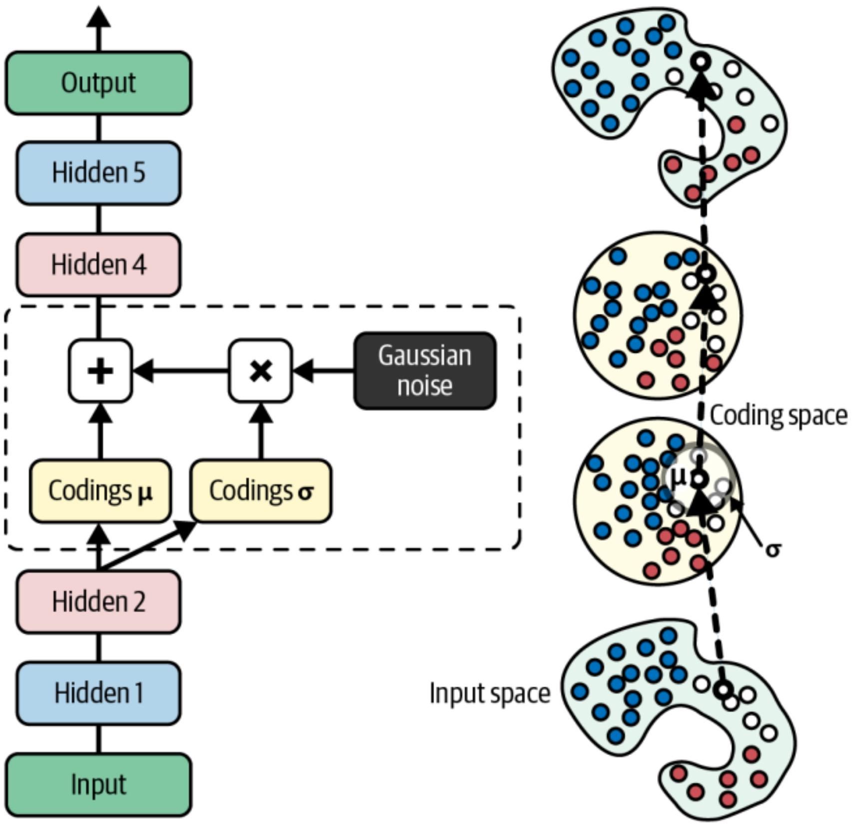
Bayesian Inference

$$P(H \mid E) = \frac{P(E \mid H) \cdot P(H)}{P(E)}$$



How VAEs Work

- Instead of directly producing a coding for a given input, the encoder creates a *mean coding* $\boldsymbol{\mu}$ and a standard deviation $\boldsymbol{\sigma}$.
- The actual coding is sampled randomly from a Gaussian distribution with mean $\boldsymbol{\mu}$ and standard deviation $\boldsymbol{\sigma}$.
- After that the decoder decodes the sampled coding normally.



Cost Function of VAEs

- The cost function is composed of two parts:
  1. The reconstruction loss: pushes the autoencoder to reproduce its inputs. We can use the MSE for this.
  2. The *latent loss:* pushes the autoencoder to have codings that look as if they were sampled from a Gaussian distribution. It is the KL divergence between the target distribution and the actual distribution of the codings:

$$\mathcal{L} = -\frac{1}{2}\sum_{i=1}^{n}\left[1 + \log\left(\sigma_i^2\right) - \sigma_i^2 - \mu_i^2\right]$$

  where μ_i and σ_i are the mean and standard deviation of the i-the component of the codings.

- A common tweak is using $\gamma = \log(\sigma^2)$ rather than σ:

$$\mathcal{L} = -\frac{1}{2}\sum_{i=1}^{n}\left[1 + \gamma_i - \exp(\gamma_i) - \mu_i^2\right]$$

Sampling Layer

- First, we need a custom layer to sample the codings, given $\boldsymbol{\mu}$ and $\boldsymbol{\gamma}$:

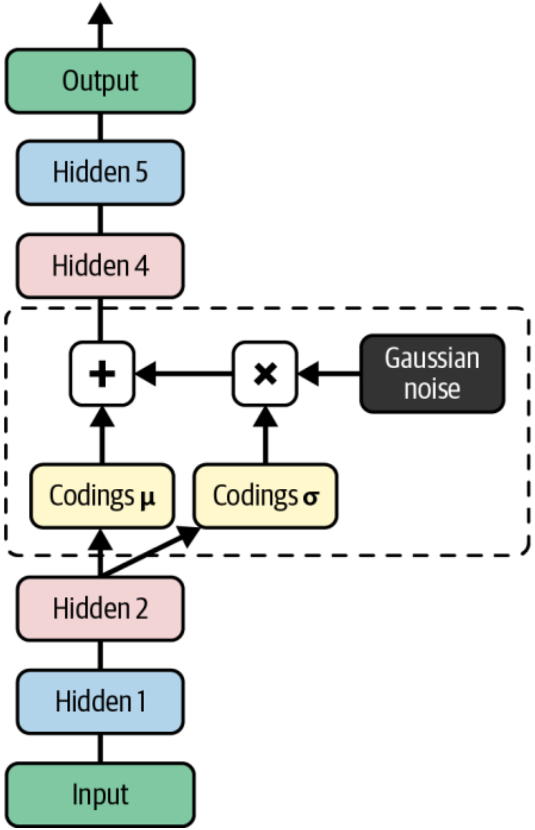
```
class Sampling(tf.keras.layers.Layer):
    def call(self, inputs):
        mean, log_var = inputs
        return tf.random.normal(tf.shape(log_var)) * tf.exp(log_var / 2) + mean
```

- This `Sampling` layer takes two inputs: `mean` ($\boldsymbol{\mu}$) and `log_var` ($\boldsymbol{\gamma}$).
  - It uses the function `tf.random.normal`() to sample a random vector (of the same shape as $\boldsymbol{\gamma}$) from the Gaussian distribution, with mean 0 and standard deviation 1.
  - Then it multiplies it by $\exp(\boldsymbol{\gamma}/2)$ (which is equal to $\boldsymbol{\sigma}$), and finally it adds $\boldsymbol{\mu}$ and returns the result.

The Encoder

```
codings_size = 10

inputs = tf.keras.layers.Input(shape=[28, 28])
Z = tf.keras.layers.Flatten()(inputs)
Z = tf.keras.layers.Dense(150, activation="relu")(Z)
Z = tf.keras.layers.Dense(100, activation="relu")(Z)
codings_mean = tf.keras.layers.Dense(codings_size)(Z)  # μ
codings_log_var = tf.keras.layers.Dense(codings_size)(Z)  # γ
codings = Sampling()([codings_mean, codings_log_var])
variational_encoder = tf.keras.Model(
    inputs=[inputs], outputs=[codings_mean, codings_log_var, codings])
```



The Decoder

```
decoder_inputs = tf.keras.layers.Input(shape=[codings_size])
x = tf.keras.layers.Dense(100, activation="relu")(decoder_inputs)
x = tf.keras.layers.Dense(150, activation="relu")(x)
x = tf.keras.layers.Dense(28 * 28)(x)
outputs = tf.keras.layers.Reshape([28, 28])(x)
variational_decoder = tf.keras.Model(inputs=[decoder_inputs], outputs=[outputs])
```

```
_, _, codings = variational_encoder(inputs)
reconstructions = variational_decoder(codings)
variational_ae = tf.keras.Model(inputs=[inputs], outputs=[reconstructions])
```

```
latent_loss = -0.5 * tf.reduce_sum(
    1 + codings_log_var - tf.exp(codings_log_var) - tf.square(codings_mean),
    axis=-1)
variational_ae.add_loss(tf.reduce_mean(latent_loss) / 784.)
```

Generating Fashion MNIST Images

```
codings = tf.random.normal(shape=[3 * 7, codings_size])
images = variational_decoder(codings).numpy()
```

Semantic Interpolation

- Variational autoencoders make it possible to perform *semantic interpolation*: instead of interpolating between two images at the pixel level, which would look as if the two images were just overlaid, we can interpolate at the codings level.

```
codings = np.zeros([7, codings_size])
codings[:, 3] = np.linspace(-0.8, 0.8, 7)  # axis 3 looks best in this case
images = variational_decoder(codings).numpy()
```