# Deep Learning

## Week 9: Variational autoencoder

## Contents



## Introduction

In the last week of the module we saw how normalising flow probabilistic deep learning models can be used to model data distributions. In particular, you learned about the NICE, RealNVP and Glow models, and saw how these flows can be built and trained using custom layers in Keras.

In this week of the module, we will look at another important deep learning algorithm: the variational autoencoder, or VAE. The VAE is an algorithm for inference and learning in a latent variable generative model. It has been successfully applied in a variety of application domains, such as neuroimaging (Benou et al 2016), drug discovery (Jin et al 2018), anomaly detection (Xu et al 2018), image generation (Vahdat & Kautz 2020) and music generation (Dhariwal et al 2020).

In its simplest form, the VAE is an unsupervised learning algorithm, and like normalising flows, the generative model can be used to create new examples similar to the dataset. However, unlike normalising flows, the generative model is not invertible, and so it's not as straightforward to train the model using maximum likelihood.

The VAE uses the principle of variational inference to approximate the posterior distribution, by defining a parameterised family of distributions conditioned on a data example, and then maximising a lower bound on the marginal likelihood. This is the evidence lower bound, or ELBO.

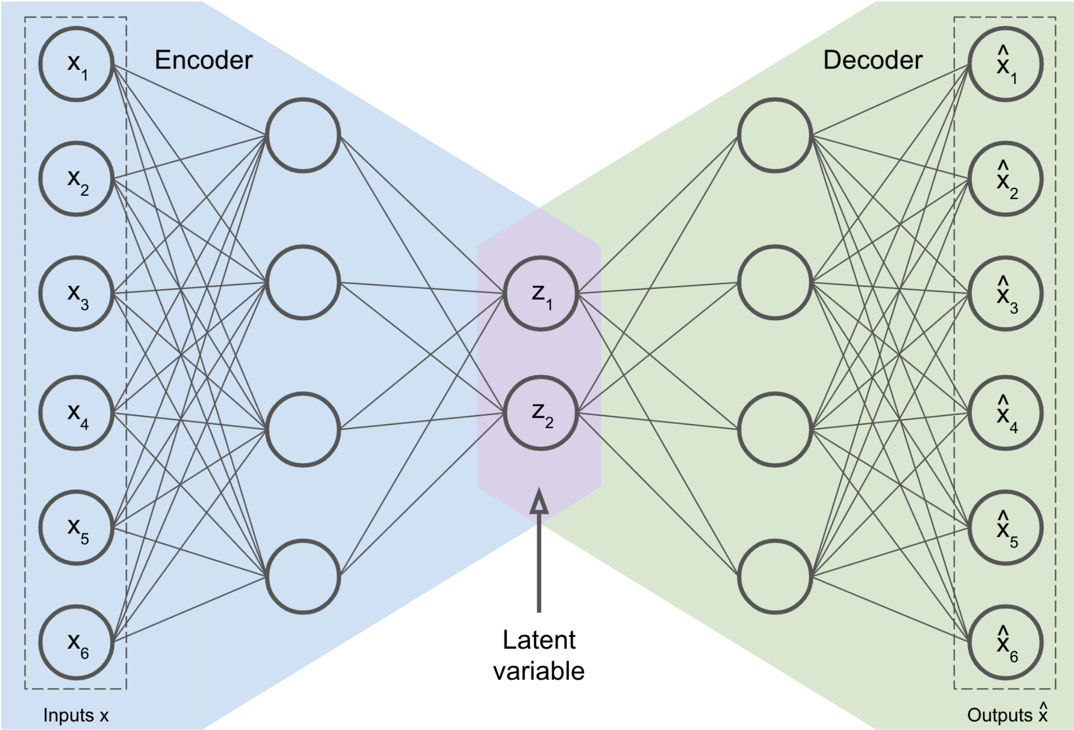
In this week, we'll build up the pieces we need to implement a variational autoencoder with Keras, starting with looking at building regular autoencoder architectures.

## Autoencoders

In this section, we'll look at how to implement a standard autoencoder architecture.

An autoencoder can be viewed as a compression algorithm, similar to a VAE, although it's not a probabilistic model, and it is not a model of the underlying data distribution.

The aim of an autoencoder is to learn an efficient data encoding. The network is normally trained in an unsupervised manner, and the task of the network is to reproduce its input as its output.




An autoencoder network architecture, with bottleneck latent variable $\mathbf{z}$

The autoencoder has a bottleneck architecture as in the above figure, and can be broken into two parts: the **encoder** network and the **decoder** network. In the middle of the bottleneck is the latent variable $\mathbf{z}$, which captures the encoding of the data. The dimensionality of $\mathbf{z}$ is typically much lower than the data $\mathbf{x}$, and so the network is trained to perform nonlinear dimension reduction (Kramer 1991). The job of the encoder is to learn an efficient representation of the data in a much lower dimensional encoding space, whilst the decoder is required to decompress the latent code to reconstruct the data input $\mathbf{x}$.

For an autoencoder network $f_\theta$, the model is trained to minimise the loss

$$L(\theta; \mathcal{D}_{train}) = \frac{1}{|\mathcal{D}_{train}|} \sum_{x_i \in \mathcal{D}_{train}} l(x_i, f_\theta(x_i)),$$

where $l : \mathbb{R}^D \times \mathbb{R}^D \mapsto \mathbb{R}$ is a suitable loss function, such as mean squared error. In practice, the model is trained using minibatches of data as usual.

There are several variants of the autoencoder model, one notable example being the **denoising autoencoder** (Vincent & Larochelle 2010). In this model, the input $\mathbf{x}$ is corrupted with noise to produce the input $\tilde{\mathbf{x}}$, and the model is trained to minimise the loss

$$L(\theta; \mathcal{D}_{train}) = \frac{1}{|\mathcal{D}_{train}|} \sum_{x_i \in \mathcal{D}_{train}} l(x_i, f_\theta(\tilde{x}_i)).$$

In other words, the model is tasked to clean the corrupted input by encoding it into a suitable representation. Intuitively, this is motivated by the idea that good representations should be robust to the corruption of the input $\mathbf{x}$, and that to denoise the input successfully, the model needs to extract features that capture useful structure in the distribution of the input, and ignore features in the data that are unimportant.

The noise is typically injected stochastically during the training run, according to a prescribed distribution $q(\tilde{\mathbf{x}} \mid \mathbf{x})$, so that the noise is different on each epoch.

## CNN autoencoder example

In this section, we will implement a CNN autoencoder for the Fashion-MNIST dataset, and examine the learned encodings.

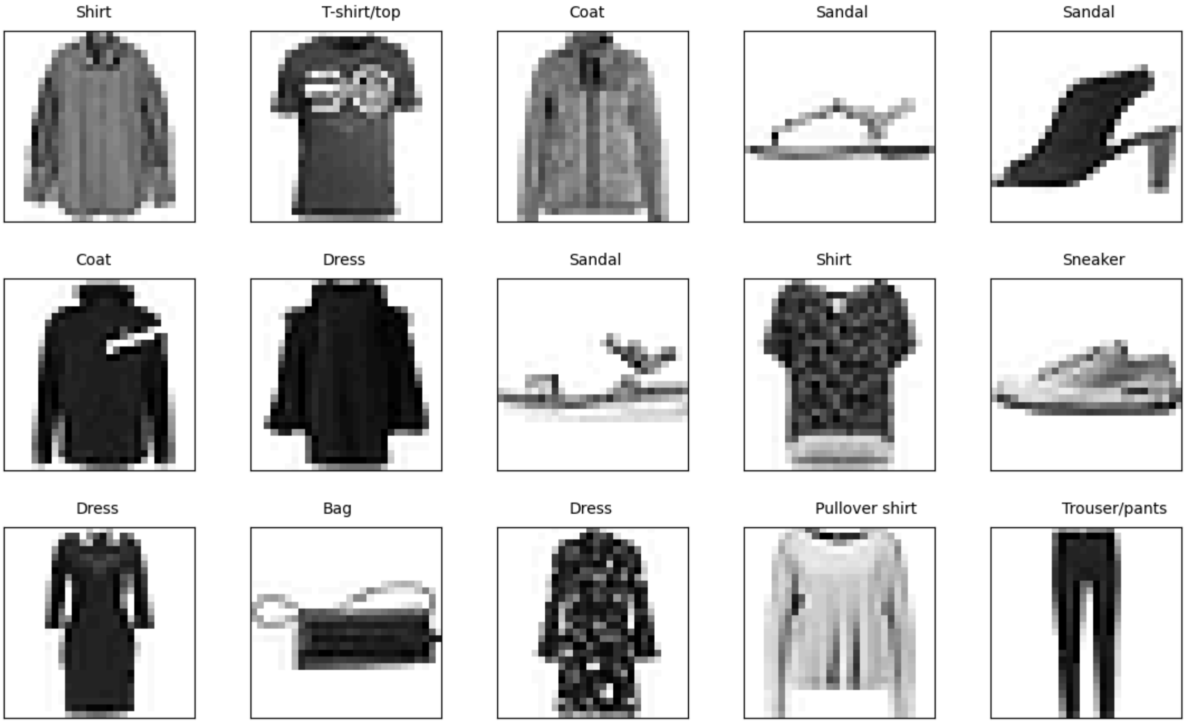
In [2]:
```
import keras
from keras import ops
```

The Fashion-MNIST dataset can be loaded with the Keras API.

In [3]:
```
# Load the dataset

import numpy as np

(x_train, y_train), (x_test, y_test) = keras.datasets.fashion_mnist.load_data()
x_train = (x_train / 255.).astype(np.float32)
x_test = (x_test / 255.).astype(np.float32)
```

In [4]:
```
# Store the class names

class_names = np.array(['T-shirt/top', 'Trouser/pants', 'Pullover shirt', 'Dress',
                        'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag','Ankle boot'])
```

In [5]:
```
# Display a few examples

import matplotlib.pyplot as plt

n_rows, n_cols = 3, 5
fig, axes = plt.subplots(n_rows, n_cols, figsize=(14, 8))
inx = np.random.choice(x_train.shape[0], n_rows*n_cols, replace=False)
fig.subplots_adjust(hspace=0.3, wspace=0.1)

for n, (image, label) in enumerate(zip(x_train[inx], y_train[inx])):
    row = n // n_cols
    col = n % n_cols
    axes[row, col].imshow(image, cmap='binary')
    axes[row, col].get_xaxis().set_visible(False)
    axes[row, col].get_yaxis().set_visible(False)
    axes[row, col].text(10., -2.5, f'{class_names[label]}')
plt.show()
```




## Build the CNN autoencoder model

We define the encoder and decoder networks separately.

In [7]:
```
# Build a CNN encoder

from keras.models import Sequential
from keras.layers import Input, Conv2D, MaxPool2D, Flatten, Dense

encoded_dim = 2

cnn_encoder = Sequential([
    Input(shape=(28, 28, 1)),
    Conv2D(16, 5, activation='relu'),
    MaxPool2D(2),
    Conv2D(8, 5, activation='relu'),
    Flatten(),
    Dense(64, activation='relu'),
    Dense(encoded_dim)
])
cnn_encoder.summary()
```

**Model: "sequential_1"**

| **Layer (type)** | **Output Shape** | **Param #** |
|---|---|---|
| conv2d_2 (Conv2D) | (None, 24, 24, 16) | 416 |
| max_pooling2d_1 (MaxPooling2D) | (None, 12, 12, 16) | 0 |
| conv2d_3 (Conv2D) | (None, 8, 8, 8) | 3,208 |
| flatten_1 (Flatten) | (None, 512) | 0 |
| dense_2 (Dense) | (None, 64) | 32,832 |
| dense_3 (Dense) | (None, 2) | 130 |

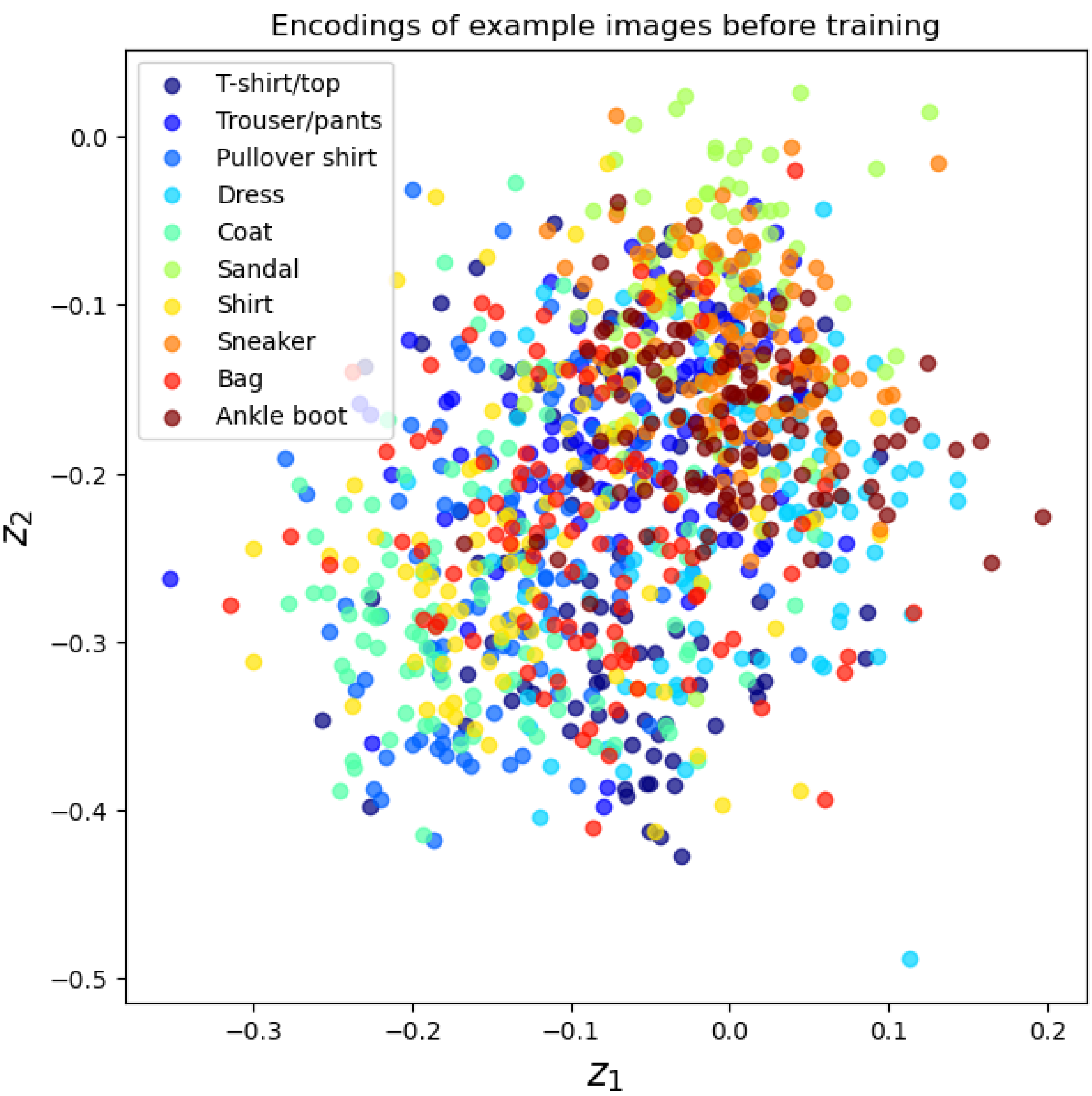
**Total params:** 36,586 (142.91 KB)

**Trainable params:** 36,586 (142.91 KB)

**Non-trainable params:** 0 (0.00 B)

In [9]:
```
# Compute encodings before training

inx = np.random.choice(x_test.shape[0], 1000, replace=False)
untrained_encodings = ops.convert_to_numpy(cnn_encoder(x_test[inx]))
untrained_encoding_labels = y_test[inx]
```

In [10]:
```
# Plot untrained encodings

plt.figure(figsize=(7, 7))
cmap = plt.get_cmap('jet', 10)
for i, class_label in enumerate(class_names):
    inx = np.where(untrained_encoding_labels == i)[0]
    plt.scatter(untrained_encodings[inx, 0], untrained_encodings[inx, 1],
                color=cmap(i), label=class_label, alpha=0.7)
plt.xlabel('$z_1$', fontsize=16)
plt.ylabel('$z_2$', fontsize=16)
plt.title('Encodings of example images before training')
plt.legend()
plt.show()
```




In [11]:
```
# Build a CNN decoder

from keras.layers import Reshape, UpSampling2D, Conv2DTranspose

cnn_decoder = Sequential([
    Input(shape=(encoded_dim,)),
```

```
    Dense(64, activation='relu'),
    Dense(512, activation='relu'),
    Reshape((8, 8, 8)),
    Conv2DTranspose(16, 5, activation='relu'),
    UpSampling2D((2, 2)),
    Conv2DTranspose(1, 5, activation='sigmoid')
])
cnn_decoder.summary()
```

**Model: "sequential_2"**

| Layer (type) | Output Shape | Param # |
|---|---|---|
| dense_4 (Dense) | (None, 64) | 192 |
| dense_5 (Dense) | (None, 512) | 33,280 |
| reshape (Reshape) | (None, 8, 8, 8) | 0 |
| conv2d_transpose (Conv2DTranspose) | (None, 12, 12, 16) | 3,216 |
| up_sampling2d (UpSampling2D) | (None, 24, 24, 16) | 0 |
| conv2d_transpose_1 (Conv2DTranspose) | (None, 28, 28, 1) | 401 |

**Total params:** 37,089 (144.88 KB)

**Trainable params:** 37,089 (144.88 KB)

**Non-trainable params:** 0 (0.00 B)

In [16]:
```
# Define the autoencoder

from keras.models import Model

cnn_autoencoder = Model(inputs=cnn_encoder.inputs,
                        outputs=cnn_decoder(cnn_encoder.outputs))
```

Make train and test Datasets

In [17]:
```
# Create Dataset objects for train and test sets

import tensorflow as tf

train_dataset = tf.data.Dataset.from_tensor_slices((x_train, x_train))
test_dataset = tf.data.Dataset.from_tensor_slices((x_test, x_test))
```

In [18]:
```
# Process the datasets

train_dataset = train_dataset.shuffle(1000)

train_dataset = train_dataset.batch(64).prefetch(tf.data.AUTOTUNE)
test_dataset = test_dataset.batch(64).prefetch(tf.data.AUTOTUNE)
```

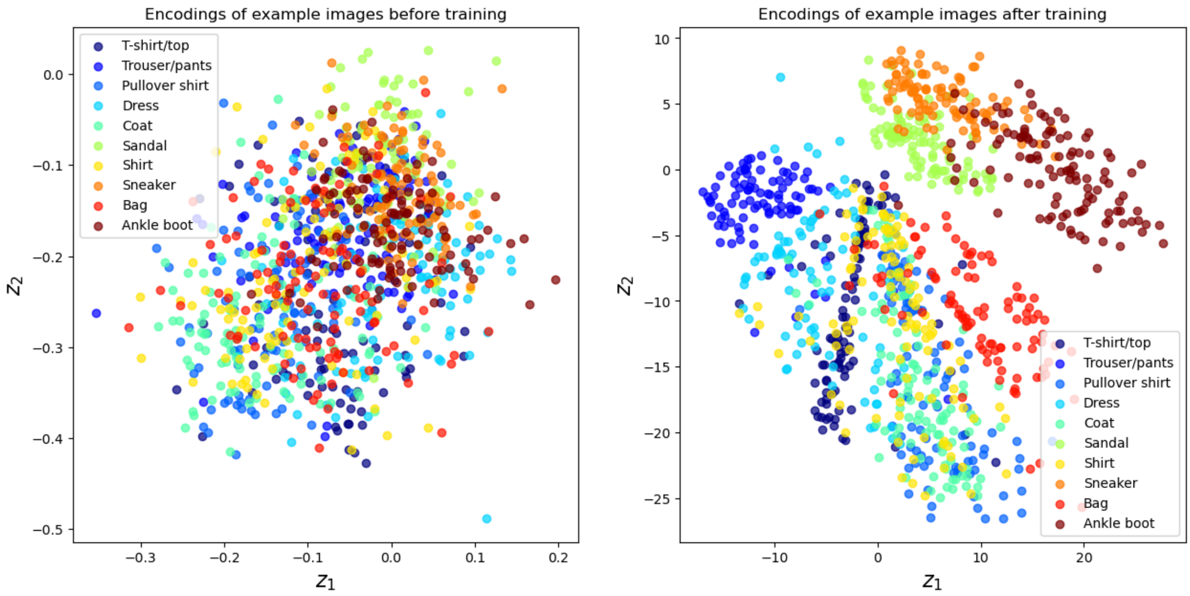
In [19]:
```
# Compile and fit the model

cnn_autoencoder.compile(loss='binary_crossentropy')
cnn_autoencoder.fit(train_dataset, epochs=10)
```

```
Epoch 1/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - loss: 0.3352
Epoch 2/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 0.3331
Epoch 3/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 0.3315
Epoch 4/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 0.3302
Epoch 5/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 0.3296
Epoch 6/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 0.3289
Epoch 7/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 0.3283
Epoch 8/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 0.3276
Epoch 9/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 0.3273
Epoch 10/10
938/938 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - loss: 0.3268
```

Out[19]:
```
<keras.src.callbacks.history.History at 0x704e27faeb90>
```

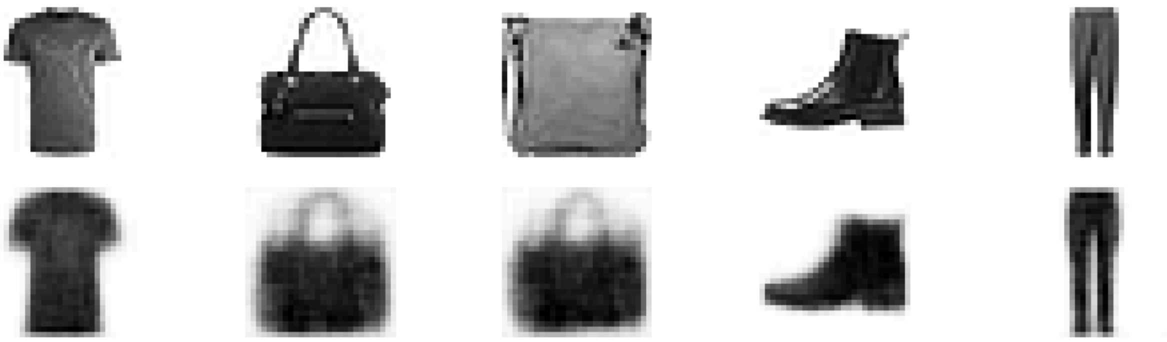
In [20]:
```python
# Compute encodings after training

inx = np.random.choice(x_test.shape[0], 1000, replace=False)
trained_encodings = ops.convert_to_numpy(cnn_encoder(x_test[inx]))
trained_encoding_labels = y_test[inx]
```

In [21]:
```python
# Plot untrained and trained encodings

plt.figure(figsize=(15, 7))
cmap = plt.get_cmap('jet', 10)

plt.subplot(1, 2, 1)
for i, class_label in enumerate(class_names):
    inx = np.where(untrained_encoding_labels == i)[0]
    plt.scatter(untrained_encodings[inx, 0], untrained_encodings[inx, 1],
                color=cmap(i), label=class_label, alpha=0.7)
plt.xlabel('$z_1$', fontsize=16)
plt.ylabel('$z_2$', fontsize=16)
plt.title('Encodings of example images before training')
plt.legend()

plt.subplot(1, 2, 2)
for i, class_label in enumerate(class_names):
    inx = np.where(trained_encoding_labels == i)[0]
    plt.scatter(trained_encodings[inx, 0], trained_encodings[inx, 1],
                color=cmap(i), label=class_label, alpha=0.7)
plt.xlabel('$z_1$', fontsize=16)
plt.ylabel('$z_2$', fontsize=16)
plt.title('Encodings of example images after training')
plt.legend()

plt.show()
```




In [23]:
```python
# Compute the autoencoder's reconstructions

inx = np.random.choice(x_test.shape[0], 5, replace=False)
reconstructed_example_images = ops.convert_to_numpy(cnn_autoencoder(x_test[inx]))
```

In [24]:
```python
# Evaluate the autoencoder's reconstructions

f, axs = plt.subplots(2, 5, figsize=(15, 4))
for j in range(5):
    axs[0, j].imshow(x_test[inx][j], cmap='binary')
    axs[1, j].imshow(reconstructed_example_images[j].squeeze(), cmap='binary')
    axs[0, j].axis('off')
    axs[1, j].axis('off')
```



*Exercise*. Redesign the CNN autoencoder above using strides $\geq 2$ for the encoder, and design the decoder to be the reverse architecture.

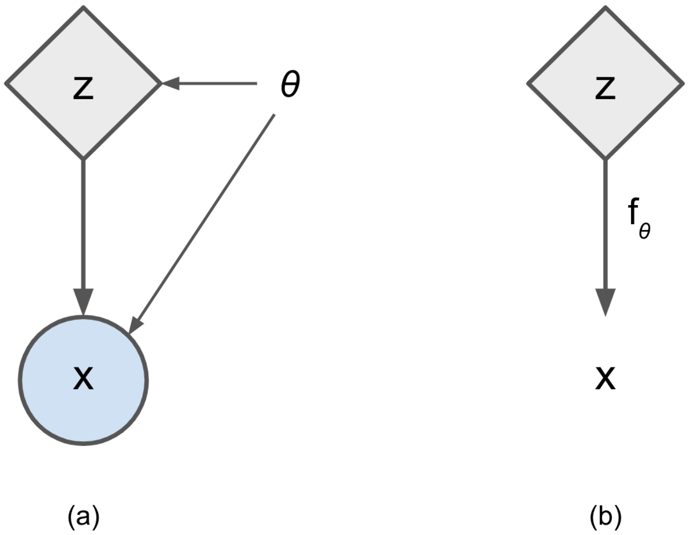
## The variational autoencoder

We will now review the variational autoencoder (VAE) algorithm, its derivation from applying the principle of variational inference to a prescribed generative model, and its connection to standard autoencoders. The VAE was developed independently by Kingma & Welling 2014 and Rezende et al 2014 at about the same time. For a general reference on variational inference, see Blei et al 2017.

First, we describe the generative model behind the variational autoencoder. This is a **latent variable generative model**, where we introduce a latent (unobserved) random variable that is intended to capture hidden causes or explanations of the data.

Furthermore, it is a **prescribed model** in the sense that we prescribe a noise model for the observations. Given a latent variable $z \in \mathbb{R}^l$, this determines a distribution over possible observations $p_\theta(x \mid z)$, with $x \in \mathbb{R}^D$. This class of generative model is also called a **likelihood-based model**, since the observations have an associated likelihood function.

This is in contrast to an **implicit model**, where there is no likelihood function on the observations, and instead a realisation of the latent variable $z$ implicitly defines the observation $x$ (note that this is the case with normalising flows, although there we have the additional special structure that the generative model is invertible, and so the observation likelihood can still be explicitly computed). This is illustrated in the following figure.




Latent variable directed graphical models; (a) a prescribed generative model that defines a likelihood for each observation, and (b) an implicit generative model. The VAE is based on the prescribed model

The generative model under consideration can be written as $p_\theta(z)p_\theta(x \mid z)$, where the conditional distribution $p_\theta(x \mid z)$ is defined by a neural network. The **marginal likelihood** (or **model evidence**) of an individual datapoint $x \in \mathbb{R}^D$ is given by

$$p_\theta(x) = \int p_\theta(z)p_\theta(x \mid z)dz, \tag{1}$$

where $\theta$ are the model parameters.

Note that under the usual i.i.d. assumption of our dataset $\mathbf{x} = (x_i)_{i=1}^N$, the full data log-likelihood is given by

$$\log p_\theta(\mathbf{x}) = \sum_{i=1}^N \log p_\theta(x_i).$$

In the following we will continue to consider the likelihood of a single datapoint $x_i$, and drop the subscript $i$ for notational convenience.

Now, we would like to choose the parameters $\theta$ that maximise the marginal likelihood. Unfortunately, the integral above is intractable in general (as is the true posterior $p(z \mid x)$), so we need to approximate it.

The approximation that we will use is the **evidence lower bound** (ELBO), or **variational free energy**, which is a lower bound on the true marginal log-likelihood:

$$
\begin{aligned}
\log p_\theta(x) &\geq \mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x \mid z)\right] - D_{KL}\left(q_\phi(z \mid x) || p_\theta(z)\right) && (2)\\
&=: \mathcal{L}(\theta, \phi; x), && (1)
\end{aligned}
$$

where $q_\phi(z \mid x)$ is a parameterised distribution of our choosing, and $D_{KL}$ denotes the Kullback-Leibler divergence, given by

$$
D_{KL}\left(q_\phi(z \mid x) || p_\theta(z)\right) = \int q_\phi(z \mid x) \log\left(\frac{q_\phi(z \mid x)}{p_\theta(z)}\right) dz.
$$

The two terms in $(2)$ are often interpreted as a reconstruction loss term and a regularisation term:

$$
\mathcal{L}(\theta, \phi; x) = \underbrace{\mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x \mid z)\right]}_{\text{reconstruction loss}} - \underbrace{D_{KL}\left(q_\phi(z \mid x) || p_\theta(z)\right)}_{\text{regulariser}}.
$$

This decomposition shows the connection to autoencoders: if $q_\phi(z \mid x)$ is a parameterised neural network, then we can view this as the encoder and $p_\theta(x \mid z)$ as the decoder. Then the reconstruction loss is the probabilistic version of the autoencoder reconstruction loss (where we could consider $q_\phi(z \mid x)$ as a delta distribution). The second term regularises the encoder, and ensures it doesn't stray too far from the prior distribution $p_\theta(z)$.

The ELBO is also sometimes written as $\mathcal{L}(\theta, \phi; x) = \mathbb{E}_{q_\phi(z|x)}\left[-\log q_\phi(z \mid x) + \log p_\theta(x, z)\right]$.

## Derivation of the ELBO

We will derive the evidence lower bound in two different ways. The first is a simple derivation using Jensen's inequality, and the second will help to shed some light on the optimal choice for the distribution $q_\phi(z \mid x)$.

*Derivation 1.* The marginal log-likelihood is given by (cf. $(1)$)

$$
\begin{aligned}
\log p_\theta(x) &= \log \int p_\theta(x \mid z) p_\theta(z) dz && (2)\\
&= \log \int p_\theta(x \mid z) \frac{p_\theta(z)}{q_\phi(z \mid x)} q_\phi(z \mid x) dz && (3)\\
&\geq \int \log\left(p_\theta(x \mid z) \frac{p_\theta(z)}{q_\phi(z \mid x)}\right) q_\phi(z \mid x) dz && (4)\\
&= \int q_\phi(z \mid x) \log p_\theta(x \mid z) dz - \int q_\phi(z \mid x) \log\left(\frac{q_\phi(z \mid x)}{p_\theta(z)}\right) dz && (5)\\
&= \mathcal{L}(\theta, \phi; x), && (6)
\end{aligned}
$$

where the third line in the above uses Jensen's inequality.

*Derivation 2.* Let $q_\phi(z \mid x)$ be a parameterised family of distributions that we use to approximate the true posterior $p_\theta(z \mid x)$. We define the objective function that we wish to minimise as the KL-divergence $D_{KL}(q_\phi(z \mid x) || p_\theta(z \mid x))$. Then we have

$$D_{KL}(q_\phi(z \mid x)||p_\theta(z \mid x)) = \int q_\phi(z \mid x) \log\left(\frac{q_\phi(z \mid x)}{p_\theta(z \mid x)}\right) dz \tag{7}$$

$$= \int q_\phi(z \mid x) \log\left(\frac{q_\phi(z \mid x) p_\theta(x)}{p_\theta(x \mid z) p_\theta(z)}\right) dz \tag{8}$$

$$= \int q_\phi(z \mid x) \log p_\theta(x) dz - \int q_\phi(z \mid x) \log p_\theta(x \mid z) dz \tag{9}$$

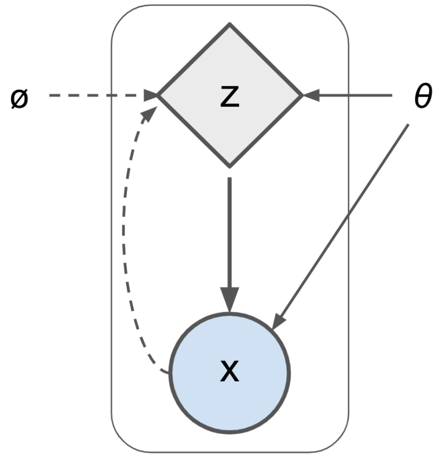
$$+ \int q_\phi(z \mid x) \log\left(\frac{q_\phi(z \mid x)}{p_\theta(z)}\right) dz \tag{10}$$

$$= \log p_\theta(x) - \mathcal{L}(\theta, \phi; x) \tag{11}$$

Since the KL-divergence is always non-negative, the above shows that $\mathcal{L}(\theta, \phi; x)$ is indeed a lower bound on the marginal log-likelihood $\log p_\theta(x)$. Furthermore, it shows that the gap in the bound is given by $D_{KL}(q_\phi(z \mid x)||p_\theta(z \mid x))$.

We see that to maximise the ELBO, the distribution $q_\phi(z \mid x)$ should approximate the true posterior $p_\theta(z \mid x))$. And the better the approximation, the tighter the bound.

The following figure illustrates that the variational autoencoder adds the variational approximation $q_\phi(z \mid x)$ to the intractable true posterior $p_\theta(z \mid x))$.




The prescribed generative model underlying the variational autoencoder, and the variational approximation $q_\phi(z \mid x)$ with variational parameters $\phi$ depicted with dashed lines

Note that the generative model parameters $\theta$ and the variational parameters $\phi$ are **global variables**, whereas the latent random variable $z$ is a **local variable**. The variational parameters $\phi$ are shared across all data points, and are not specific to individual data points, in contrast to traditional mean-field variational inference. This strategy is known as **amortized inference** (Hoffman et al 2013).

## The reparameterization trick

We have now defined our ELBO objective function that we wish to maximise, which is a lower bound on the marginal log-likelihood:

$$\mathcal{L}(\theta, \phi; x) = \mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x \mid z)\right] - D_{KL}\left(q_\phi(z \mid x)||p_\theta(z)\right)$$

Note that we are able to evaluate the densities $q_\phi(z \mid x)$, $p_\theta(x \mid z)$, $p_\theta(z)$ as well as sample from the approximating distribution $q_\phi(z \mid x)$, so the ELBO can be evaluated using Monte Carlo samples $\{z^{(j)}\}_{j=1}^{L}$, with $z^{(j)}$ sampled from $q_\phi(z \mid x)$

$$\mathcal{L}(\theta, \phi; x) \approx \frac{1}{L} \sum_{j=1}^{L} \log p_\theta(x \mid z^{(j)}) + \log p_\theta(z^{(j)}) - \log q_\phi(z^{(j)} \mid x)$$

The question remains how to optimise the ELBO with respect to the parameters $\theta$ and $\phi$. Note that taking gradients with respect to $\phi$ is not straightforward, as the $z^{(j)}$ are samples.

A typical **score-function estimator** (Glynn 1990, Kleijnen & Rubinstein 1996) for the general type of problem of taking a gradient of an expectation of some function $f(z)$ is given by

$$\begin{aligned}
\nabla_\phi \mathbb{E}_{q_\phi(z)}\left[f(z)\right] &= \mathbb{E}_{q_\phi(z)}\left[f(z) \nabla_\phi \log q_\phi(z)\right] && (12)\\
&\approx \frac{1}{L} \sum_{j=1}^{L} \left[f(z^{(j)}) \nabla_\phi \log q_\phi(z^{(j)})\right]. && (13)
\end{aligned}$$

This estimator is also used in reinforcement learning for policy gradients, where it is often referred to as the REINFORCE algorithm (Williams 1992). However, this estimator typically has high variance (Blei et al 2012), and in our case we can do better, in particular since our function $f(z)$ $\left(= \log p_\theta(x, z^{(j)}) - \log q_\phi(z^{(j)} \mid x)\right)$ is differentiable.

The 'reparameterization trick' simply reparameterises the random latent variable $z \sim q_\phi(z \mid x)$ using a differentiable transformation $g_\phi(\epsilon, x)$ of an auxiliary noise variable $\epsilon \sim p(\epsilon)$:

$$\begin{aligned}
z &\sim q_\phi(z \mid x)\\
z &= g_\phi(\epsilon, x), \quad \epsilon \sim p(\epsilon)
\end{aligned}$$

An example is if $q_\phi(z \mid x)$ a multivariate Gaussian distribution $N(\mu_\phi(x), \Sigma_\phi(x))$. Then we could reparameterise the distribution as

$$p(\epsilon) = N(\mathbf{0}, \mathbf{I}), \quad g_\phi(\epsilon, x) = \mu_\phi(x) + L_\phi(x)\epsilon, \quad \text{where } L_\phi(x) L_\phi(x)^T = \Sigma_\phi(x).$$

The encoder network would then output the distribution parameters $\mu_\phi(x)$ and $L_\phi(x)$, which are both fully differentiable with respect to $\phi$. Note that the noise variable distribution does not depend on any parameters.

If the transformation $g_\phi(\cdot, x) : \mathbb{R}^d \mapsto \mathbb{R}^l$ is invertible, this is nothing more than a change of variables, so the change of variables formula applies to give

$$\begin{aligned}
q_\phi(z \mid x) &= \left|\det J_{g_\phi}(\epsilon)\right|^{-1} \cdot p(\epsilon) && (14)\\
&= \left|\frac{d\epsilon}{dz}\right| \cdot p(\epsilon) && (15)
\end{aligned}$$

This leads to the **pathwise estimator** (Devroye 1996), which for a general function $f(z)$ and reparameterization $g_\phi(\epsilon)$ is given by

$$\begin{aligned}
\nabla_\phi \mathbb{E}_{q_\phi(z)}\left[f(z)\right] &= \nabla_\phi \int q_\phi(z) f(z) dz && (16)\\
&= \nabla_\phi \int \left|\frac{d\epsilon}{dz}\right| \cdot p(\epsilon) f(g_\phi(\epsilon)) \left|\frac{dz}{d\epsilon}\right| d\epsilon && (17)\\
&= \nabla_\phi \mathbb{E}_{p(\epsilon)}\left[f(g_\phi(\epsilon))\right] && (18)\\
&= \mathbb{E}_{p(\epsilon)}\left[\nabla_\phi f(g_\phi(\epsilon))\right] && (19)
\end{aligned}$$

In our case, the reparameterization $g_\phi(\epsilon, x)$ and noise variable $\epsilon \sim p(\epsilon)$ leads to the **Stochastic Gradient Variational Bayes** (SGVB) estimator $\hat{\mathcal{L}}^A(\theta, \phi; x) \approx \mathcal{L}(\theta, \phi; x)$:

$$\hat{\mathcal{L}}^A(\theta,\phi;x) := \frac{1}{L}\sum_{j=1}^{L} \log p_\theta(x, z^{(j)}) - \log q_\phi(z^{(j)}|x) \tag{3}$$
$$\text{where } z^{(j)} = g_\phi(\epsilon^{(j)}, x) \quad \text{and} \quad \epsilon^{(j)} \sim p(\epsilon)$$

We can now use this estimator to approximate the ELBO objective, and take its gradients with respect to the parameters $(\theta, \phi)$ on minibatches of data to optimise them:

$$\begin{aligned}
\mathbb{E}_{x\sim p_{data}}[\log p_\theta(x)] &\approx \frac{1}{|\mathcal{D}_{train}|}\sum_{i\in\mathcal{D}_{train}} \log p_\theta(x_i) && (20)\\
&\geq \frac{1}{|\mathcal{D}_{train}|}\sum_{i\in\mathcal{D}_{train}} \mathcal{L}(\theta,\phi;x_i) && (21)\\
&\approx \frac{1}{|\mathcal{D}_{train}|}\sum_{i\in\mathcal{D}_{train}} \hat{\mathcal{L}}^A(\theta,\phi;x) && (22)\\
&\approx \frac{1}{M}\sum_{i\in\mathcal{D}_m} \hat{\mathcal{L}}^A(\theta,\phi;x), && (23)
\end{aligned}$$

where as usual $\mathcal{D}_m$ is a randomly sampled minibatch of training data points, and $M = |\mathcal{D}_m|$.

Note that we wish to maximise the above objective, so in practice we will take the negative of the quantity above to minimise.

Finally, depending on the choice of prior $p_\theta(z)$ and variational posterior $q_\phi(z \mid x)$, it may be possible to analytically evaluate the KL-divergence term $D_{KL}(q_\phi(z \mid x)||p_\theta(z))$. This is true, for example, in the case where both distributions are Gaussian. In this case, there is no need to approximate this term in the ELBO with Monte Carlo samples, and we can instead use the alternate version of the SGVB estimator:

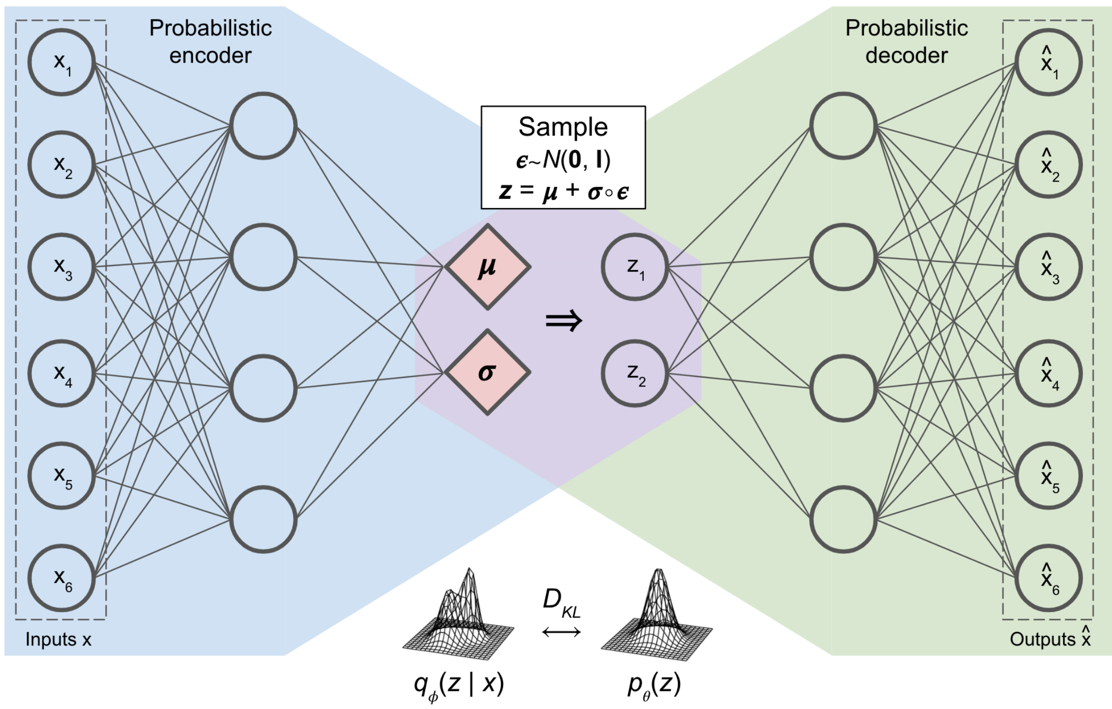
$$\hat{\mathcal{L}}^B(\theta,\phi;x) := \frac{1}{L}\sum_{j=1}^{L} \log p_\theta(x \mid z^{(j)}) - D_{KL}(q_\phi(z \mid x)||p_\theta(z)), \tag{4}$$

where as before $z^{(j)} = g_\phi(\epsilon^{(j)}, x)$ and $\epsilon^{(j)} \sim p(\epsilon)$.

It is worth noting that it is common in practice to take a single Monte Carlo sample ($L = 1$) in the SGVB estimator $(3)$ or $(4)$, particularly for larger minibatch sizes.

We summarise the Auto-Encoding Variational Bayes (variational autoencoder) algorithm as follows. The algorithm inputs are the encoder and decoder networks $q_\phi(z \mid x)$, $p_\theta(x \mid z)$, prior distribution $p_\theta(z)$, minibatch size $M$ and number of Monte Carlo samples $L$.

> Initialise $\phi$, $\theta$ randomly
> **while** not converged:
>     sample minibatch $\mathcal{D}_m$ of $M$ data examples
>     sample $M \times L$ noise variables $\epsilon^{(i,j)}$ for each $x_i \in \mathcal{D}_m$ and $j = 1, \ldots, L$
>     compute gradient $\frac{1}{M}\sum_{x_i\in\mathcal{D}_m} \nabla_{\phi,\theta}\hat{\mathcal{L}}(\theta,\phi;x_i)$, where $\hat{\mathcal{L}}$ is $\hat{\mathcal{L}}^A$ or $\hat{\mathcal{L}}^B$
>     update parameters by applying gradient with a NN optimiser (e.g. SGD, Adam)




The variational encoder. The encoder/inference network defines the latent variable distribution via the reparameterization trick, the decoder/generative network reconstructs the original input by defining a likelihood $p_\theta(x \mid z)$. The variational posterior $q_\phi(z \mid x)$ is penalised for varying too much from the prior $p_\theta(z)$

## VAE implementation

In this section we will develop a full implementation of the variational autoencoder. This implementation will involve model subclassing, which is a fully flexible way to build models in Keras.

In [2]:
```
import keras
from keras import ops
```

### Load the Frey Face dataset

We will use the Frey Face dataset to demonstrate the VAE, as in the original paper by Kingma & Welling.

In [3]:
```
# Load the data

import numpy as np

faces_data = np.load('./data/frey_faces.npy')
faces_data.shape
```

Out[3]:
```
(1965, 28, 20)
```

In [4]:
```
# Split data into train and validation sets

from sklearn.model_selection import train_test_split

x_train, x_val = train_test_split(faces_data, test_size=0.1)
x_train.shape
```

Out[4]:
```
(1768, 28, 20)
```

In [5]:
```python
# View a sample of the data

import matplotlib.pyplot as plt

n_rows, n_cols = 4, 10
fig, axes = plt.subplots(n_rows, n_cols, figsize=(14, 8))
inx = np.random.choice(x_train.shape[0], n_rows*n_cols, replace=False)
fig.subplots_adjust(hspace=0., wspace=0.)

for n, image in enumerate(x_train[inx]):
    row = n // n_cols
    col = n % n_cols
    axes[row, col].imshow(image, cmap='gray')
    axes[row, col].get_xaxis().set_visible(False)
    axes[row, col].get_yaxis().set_visible(False)
plt.show()
```

In [6]:
```python
# Load the data into DataLoaders

import torch

class FreyFaceDataset(torch.utils.data.Dataset):

    def __init__(self, images):
        self.images = (images / 255.).astype(np.float32)

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        image = self.images[idx]
        return (image,)


train_dataset = FreyFaceDataset(x_train)
val_dataset = FreyFaceDataset(x_val)

train_dataloader = torch.utils.data.DataLoader(train_dataset,
                                               shuffle=True,batch_size=100)
val_dataloader = torch.utils.data.DataLoader(val_dataset,
                                             shuffle=False, batch_size=20)
```

### Generative model

Recall that the generative model $p_\theta(z)p_\theta(x \mid z)$ is defined by the prior $p_\theta(z)$ and decoder $p_\theta(x \mid z)$. We will choose a standard isotropic Gaussian distribution for the prior.

For the decoder, we follow Kingma & Welling and use a Gaussian likelihood, but constrain the mean to $[0, 1]$.

It is worth mentioning that it is also common practice to use an independent Bernoulli likelihood per pixel in the decoder for similar image datasets (Kingma & Welling uses this for MNIST), despite this being incorrect as the data is not binary.

In [7]:

```
# Define the decoder

from keras.models import Model
from keras.layers import Input, Dense, Reshape

img_h, img_w = 28, 20
latent_dim = 2

inputs = Input(shape=(latent_dim,))
h = Dense(200, activation='relu')(inputs)
h = Dense(img_h * img_w * 2)(h)
h = Reshape((img_h, img_w, 2))(h)
h1, h2 = ops.unstack(h, axis=-1)
x_mean = ops.sigmoid(h1)
x_log_std = h2

decoder = Model(inputs=inputs, outputs=[x_mean, x_log_std], name='decoder')
decoder.summary()
```

**Model: "decoder"**

| Layer (type) | Output Shape | Param # |
|---|---|---|
| input_layer (InputLayer) | (None, 2) | 0 |
| dense (Dense) | (None, 200) | 600 |
| dense_1 (Dense) | (None, 1120) | 225,120 |
| reshape (Reshape) | (None, 28, 20, 2) | 0 |
| unstack (Unstack) | [(None, 28, 20), (None, 28, 20)] | 0 |
| sigmoid (Sigmoid) | (None, 28, 20) | 0 |

**Total params:** 225,720 (881.72 KB)
**Trainable params:** 225,720 (881.72 KB)
**Non-trainable params:** 0 (0.00 B)

### Inference model

We now define the encoder, or inference model $q_\phi(z \mid x)$. We will use a diagonal Gaussian for the approximate posterior, where the mean and diagonal covariance matrix are predicted by the encoder.

In [8]:

```
# Define the encoder

from keras.layers import Flatten
```

```
inputs = Input(shape=(img_h, img_w))
h = Flatten()(inputs)
h = Dense(200, activation='relu')(h)
h = Dense(2 * latent_dim)(h)
z_mean, z_log_var = ops.split(h, 2, axis=-1)

encoder = Model(inputs=inputs, outputs=[z_mean, z_log_var], name='encoder')
encoder.summary()
```

**Model: "encoder"**

| Layer (type) | Output Shape | Param # |
|---|---|---|
| input_layer_1 (InputLayer) | (None, 28, 20) | 0 |
| flatten (Flatten) | (None, 560) | 0 |
| dense_2 (Dense) | (None, 200) | 112,200 |
| dense_3 (Dense) | (None, 4) | 804 |
| split (Split) | [(None, 2), (None, 2)] | 0 |

**Total params:** 113,004 (441.42 KB)
**Trainable params:** 113,004 (441.42 KB)
**Non-trainable params:** 0 (0.00 B)

## Training the encoder and decoder

We now compile and fit the encoder and decoder networks. Recall the ELBO objective

$$\mathcal{L}(\theta, \phi; x) = \mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x \mid z)\right] - D_{KL}\left(q_\phi(z \mid x) || p_\theta(z)\right).$$

Since the prior and approximate posterior are both Gaussian, we will use the second form of the SGVB estimator (we will set $L = 1$):

$$\hat{\mathcal{L}}^B(\theta, \phi; x) := \frac{1}{L}\sum_{j=1}^{L} \log p_\theta(x \mid z^{(j)}) - D_{KL}(q_\phi(z \mid x)||p_\theta(z)),$$

We have chosen the prior $p_\theta(z)$ to be $N(\mathbf{0}, \mathbf{I})$, and the approximate posterior can be written as $N(\mu_q, \mathrm{diag}(\sigma_q))$. In this case, we can write the KL divergence as

$$D_{KL}(q_\phi(z \mid x)||p_\theta(z)) = \frac{1}{2}\left[\mu_q^T \mu_q + \sum_{i=1}^{l}(\sigma_q)_i - l - \log\prod_{i=1}^{l}(\sigma_q)_i\right], \tag{24}$$

where $l$ is the dimension of the latent space.

To implement the VAE, we will use model subclassing to override the in-built `train_step` method (see the TensorFlow and PyTorch guides). This gives us control over what happens when we call the `.fit()` method.

In [9]:
```
# Build the VAE Model object

from keras.metrics import Mean
import tensorflow as tf


class VAE(Model):

    def __init__(self, encoder, decoder, **kwargs):
        super().__init__(**kwargs)
```

```
        self.encoder = encoder
        self.decoder = decoder
        self.loss_metric = Mean(name='loss')
        self.nll_metric = Mean(name='nll')
        self.kl_metric = Mean(name='kl')
        self.pi = ops.array(np.pi)

    def _get_losses(self, data):
        z_mean, z_log_var = self.encoder(data[0])
        kl_loss = 0.5 * ops.sum((ops.square(z_mean)
                            + ops.exp(z_log_var) - 1 - z_log_var), axis=-1)
        kl_loss = ops.mean(kl_loss)

        epsilon = keras.random.normal(ops.shape(z_mean))
        z_std = ops.exp(0.5 * z_log_var)
        z_sample = z_mean + (z_std * epsilon)

        x_mean, x_log_std = self.decoder(z_sample)  # (B, 28, 20)
        log_Z = 0.5 * ops.log(2 * self.pi)
        nll_loss = 0.5 * ops.square((data - x_mean) / ops.exp(x_log_std))
        + x_log_std + log_Z
        nll_loss = ops.mean(ops.sum(nll_loss, axis=[-1, -2]))

        loss = kl_loss + nll_loss
        return loss, kl_loss, nll_loss

    def call(self, inputs):
        z_mean, z_log_var = self.encoder(inputs)
        epsilon = keras.random.normal(ops.shape(z_mean))
        z_std = ops.exp(0.5 * z_log_var)
        z_sample = z_mean + (z_std * epsilon)
        return self.decoder(z_sample)

    def train_step(self, data):
        backend = keras.config.backend()
        if backend == 'tensorflow':
            with tf.GradientTape() as tape:
                loss, kl_loss, nll_loss = self._get_losses(data)
            grads = tape.gradient(loss, self.trainable_weights)
            self.optimizer.apply_gradients(zip(grads, self.trainable_weights))
        elif backend == 'torch':
            self.zero_grad()
            loss, kl_loss, nll_loss = self._get_losses(data)

            loss.backward()

            gradients = [v.value.grad for v in self.trainable_weights]
            with torch.no_grad():
                self.optimizer.apply(gradients, self.trainable_weights)
        else:
            raise NotImplementedError(f"Unsupported backend: {backend}")

        self.loss_metric.update_state(loss)
        self.nll_metric.update_state(nll_loss)
        self.kl_metric.update_state(kl_loss)
        return {m.name: m.result() for m in self.metrics}

    def test_step(self, data):
        loss, kl_loss, nll_loss = self._get_losses(data)
        self.loss_metric.update_state(loss)
        self.nll_metric.update_state(nll_loss)
        self.kl_metric.update_state(kl_loss)
        return {m.name: m.result() for m in self.metrics}

    @property
```

```python
    def metrics(self):
        return [self.loss_metric, self.nll_metric, self.kl_metric]
```

In [10]:
```python
# Instantiate the Model

vae = VAE(encoder, decoder, name='vae')
```

In [ ]:
```python
# Compile and fit the Model

from keras.callbacks import EarlyStopping

early_stopping = EarlyStopping(patience=10)

vae.compile(optimizer='adam')
history = vae.fit(train_dataloader, validation_data=val_dataloader,
                  epochs=200, callbacks=[early_stopping])
```

```
Epoch 1/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 2s 27ms/step - kl: 54.5660 - loss: 295.6676 - nll: 241.10
16 - val_kl: 92.9325 - val_loss: -392.6727 - val_nll: -485.6052
Epoch 2/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step - kl: 77.1622 - loss: -447.5816 - nll: -524.
7438 - val_kl: 46.5437 - val_loss: -533.4911 - val_nll: -580.0348
Epoch 3/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 29ms/step - kl: 42.6011 - loss: -551.5690 - nll: -594.
1702 - val_kl: 33.3567 - val_loss: -571.9655 - val_nll: -605.3221
Epoch 4/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - kl: 32.0961 - loss: -591.4537 - nll: -623.
5499 - val_kl: 27.2290 - val_loss: -584.3499 - val_nll: -611.5788
Epoch 5/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - kl: 26.5386 - loss: -598.4175 - nll: -624.
9561 - val_kl: 25.3625 - val_loss: -577.7645 - val_nll: -603.1269
Epoch 6/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - kl: 23.1810 - loss: -584.0470 - nll: -607.
2280 - val_kl: 20.9089 - val_loss: -588.4152 - val_nll: -609.3241
Epoch 7/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - kl: 21.7657 - loss: -607.1662 - nll: -628.
9319 - val_kl: 19.8475 - val_loss: -600.1986 - val_nll: -620.0461
Epoch 8/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - kl: 20.4701 - loss: -622.4835 - nll: -642.
9536 - val_kl: 19.0233 - val_loss: -612.3876 - val_nll: -631.4109
Epoch 9/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - kl: 19.5551 - loss: -630.3776 - nll: -649.
9327 - val_kl: 20.3582 - val_loss: -621.5499 - val_nll: -641.9083
Epoch 10/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - kl: 19.3845 - loss: -647.2494 - nll: -666.
6339 - val_kl: 18.8008 - val_loss: -649.7042 - val_nll: -668.5049
Epoch 11/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 32ms/step - kl: 18.5915 - loss: -665.9258 - nll: -684.
5173 - val_kl: 17.8761 - val_loss: -662.1248 - val_nll: -680.0010
Epoch 12/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - kl: 17.7757 - loss: -676.2123 - nll: -693.
9880 - val_kl: 18.1330 - val_loss: -656.0035 - val_nll: -674.1366
Epoch 13/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - kl: 17.1271 - loss: -676.8470 - nll: -693.
9741 - val_kl: 17.2919 - val_loss: -668.1827 - val_nll: -685.4745
Epoch 14/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - kl: 16.6053 - loss: -691.6306 - nll: -708.
2359 - val_kl: 16.9079 - val_loss: -676.5692 - val_nll: -693.4771
Epoch 15/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - kl: 16.3898 - loss: -701.5421 - nll: -717.
9319 - val_kl: 15.3904 - val_loss: -691.3827 - val_nll: -706.7731
Epoch 16/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - kl: 15.8592 - loss: -719.2132 - nll: -735.
0724 - val_kl: 15.2328 - val_loss: -698.7263 - val_nll: -713.9592
Epoch 17/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - kl: 15.3298 - loss: -725.6563 - nll: -740.
9860 - val_kl: 14.2831 - val_loss: -702.2245 - val_nll: -716.5076
Epoch 18/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - kl: 14.7039 - loss: -717.1664 - nll: -731.
8704 - val_kl: 14.7673 - val_loss: -707.3248 - val_nll: -722.0922
Epoch 19/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - kl: 14.5581 - loss: -726.3088 - nll: -740.
8670 - val_kl: 14.5427 - val_loss: -702.0098 - val_nll: -716.5525
Epoch 20/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - kl: 13.8592 - loss: -716.1331 - nll: -729.
9923 - val_kl: 13.8332 - val_loss: -703.8502 - val_nll: -717.6834
Epoch 21/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - kl: 13.9117 - loss: -724.3967 - nll: -738.
3083 - val_kl: 13.4566 - val_loss: -718.3166 - val_nll: -731.7732
Epoch 22/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - kl: 13.5363 - loss: -732.7628 - nll: -746.
2991 - val_kl: 13.3069 - val_loss: -722.5202 - val_nll: -735.8270
Epoch 23/200
```

```
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - kl: 13.5034 - loss: -737.2853 - nll: -750.
7888 - val_kl: 12.8192 - val_loss: -724.6711 - val_nll: -737.4904
Epoch 24/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - kl: 13.1229 - loss: -735.1896 - nll: -748.
3124 - val_kl: 12.7345 - val_loss: -724.5156 - val_nll: -737.2502
Epoch 25/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - kl: 12.8833 - loss: -738.8208 - nll: -751.
7041 - val_kl: 12.2381 - val_loss: -727.9149 - val_nll: -740.1531
Epoch 26/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - kl: 12.5965 - loss: -744.9024 - nll: -757.
4990 - val_kl: 12.2817 - val_loss: -730.6133 - val_nll: -742.8951
Epoch 27/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - kl: 12.4851 - loss: -746.4224 - nll: -758.
9075 - val_kl: 12.3861 - val_loss: -734.6251 - val_nll: -747.0112
Epoch 28/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - kl: 12.5243 - loss: -753.2223 - nll: -765.
7465 - val_kl: 12.6983 - val_loss: -719.9800 - val_nll: -732.6783
Epoch 29/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 29ms/step - kl: 12.3299 - loss: -748.8930 - nll: -761.
2230 - val_kl: 11.6475 - val_loss: -729.3101 - val_nll: -740.9576
Epoch 30/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - kl: 12.0809 - loss: -751.4343 - nll: -763.
5153 - val_kl: 12.4502 - val_loss: -720.9672 - val_nll: -733.4172
Epoch 31/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - kl: 12.0861 - loss: -746.5848 - nll: -758.
6709 - val_kl: 11.6610 - val_loss: -737.7368 - val_nll: -749.3978
Epoch 32/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - kl: 12.0127 - loss: -756.8195 - nll: -768.
8322 - val_kl: 11.7769 - val_loss: -741.7223 - val_nll: -753.4991
Epoch 33/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - kl: 11.9954 - loss: -765.7440 - nll: -777.
7393 - val_kl: 11.6515 - val_loss: -743.1998 - val_nll: -754.8513
Epoch 34/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - kl: 11.8301 - loss: -758.6342 - nll: -770.
4644 - val_kl: 11.5139 - val_loss: -738.7375 - val_nll: -750.2513
Epoch 35/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - kl: 11.7782 - loss: -760.4554 - nll: -772.
2335 - val_kl: 11.4057 - val_loss: -745.9750 - val_nll: -757.3807
Epoch 36/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - kl: 11.6614 - loss: -768.4497 - nll: -780.
1112 - val_kl: 11.3183 - val_loss: -747.2803 - val_nll: -758.5986
Epoch 37/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - kl: 11.5143 - loss: -768.8934 - nll: -780.
4077 - val_kl: 11.1536 - val_loss: -741.8081 - val_nll: -752.9617
Epoch 38/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step - kl: 11.4459 - loss: -767.6762 - nll: -779.
1221 - val_kl: 11.2587 - val_loss: -729.6229 - val_nll: -740.8816
Epoch 39/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - kl: 11.5458 - loss: -764.6865 - nll: -776.
2323 - val_kl: 11.4317 - val_loss: -739.2997 - val_nll: -750.7313
Epoch 40/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - kl: 11.3954 - loss: -765.4913 - nll: -776.
8867 - val_kl: 11.1125 - val_loss: -752.6779 - val_nll: -763.7904
Epoch 41/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - kl: 11.2509 - loss: -764.4271 - nll: -775.
6780 - val_kl: 10.8315 - val_loss: -751.3973 - val_nll: -762.2288
Epoch 42/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - kl: 11.0681 - loss: -778.8685 - nll: -789.
9366 - val_kl: 11.0474 - val_loss: -751.0557 - val_nll: -762.1031
Epoch 43/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - kl: 11.1320 - loss: -781.8817 - nll: -793.
0137 - val_kl: 10.7007 - val_loss: -749.0486 - val_nll: -759.7493
Epoch 44/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - kl: 11.1345 - loss: -777.4308 - nll: -788.
5653 - val_kl: 10.7763 - val_loss: -752.3547 - val_nll: -763.1311
Epoch 45/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - kl: 10.9617 - loss: -778.2737 - nll: -789.
```

```
2354 - val_kl: 10.8709 - val_loss: -758.5714 - val_nll: -769.4422
Epoch 46/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - kl: 10.8848 - loss: -781.4102 - nll: -792.
2949 - val_kl: 10.4668 - val_loss: -757.9376 - val_nll: -768.4043
Epoch 47/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - kl: 10.8193 - loss: -784.1684 - nll: -794.
9877 - val_kl: 10.1987 - val_loss: -747.6705 - val_nll: -757.8691
Epoch 48/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - kl: 10.5925 - loss: -775.3235 - nll: -785.
9160 - val_kl: 10.3628 - val_loss: -757.1893 - val_nll: -767.5521
Epoch 49/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - kl: 10.7023 - loss: -780.2882 - nll: -790.
9905 - val_kl: 10.3819 - val_loss: -758.8394 - val_nll: -769.2212
Epoch 50/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - kl: 10.6878 - loss: -779.8488 - nll: -790.
5367 - val_kl: 10.7772 - val_loss: -759.6715 - val_nll: -770.4486
Epoch 51/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - kl: 10.6500 - loss: -778.7781 - nll: -789.
4282 - val_kl: 10.6284 - val_loss: -760.3243 - val_nll: -770.9527
Epoch 52/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - kl: 10.5825 - loss: -782.9460 - nll: -793.
5285 - val_kl: 10.7892 - val_loss: -756.4574 - val_nll: -767.2465
Epoch 53/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - kl: 10.6430 - loss: -779.3554 - nll: -789.
9983 - val_kl: 10.1091 - val_loss: -756.4230 - val_nll: -766.5321
Epoch 54/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - kl: 10.5684 - loss: -773.9346 - nll: -784.
5030 - val_kl: 10.5815 - val_loss: -767.6227 - val_nll: -778.2041
Epoch 55/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - kl: 10.6393 - loss: -793.1954 - nll: -803.
8348 - val_kl: 10.6673 - val_loss: -766.2513 - val_nll: -776.9186
Epoch 56/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - kl: 10.6701 - loss: -788.6266 - nll: -799.
2966 - val_kl: 10.3836 - val_loss: -769.7987 - val_nll: -780.1823
Epoch 57/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - kl: 10.5720 - loss: -788.7788 - nll: -799.
3507 - val_kl: 10.4613 - val_loss: -772.5193 - val_nll: -782.9805
Epoch 58/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - kl: 10.5870 - loss: -795.3962 - nll: -805.
9832 - val_kl: 10.2609 - val_loss: -771.6398 - val_nll: -781.9007
Epoch 59/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - kl: 10.3914 - loss: -792.1815 - nll: -802.
5728 - val_kl: 10.3098 - val_loss: -761.8081 - val_nll: -772.1179
Epoch 60/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - kl: 10.1931 - loss: -780.7876 - nll: -790.
9807 - val_kl: 9.9375 - val_loss: -773.2677 - val_nll: -783.2052
Epoch 61/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - kl: 10.0782 - loss: -780.8503 - nll: -790.
9285 - val_kl: 9.6190 - val_loss: -751.7928 - val_nll: -761.4119
Epoch 62/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - kl: 10.1235 - loss: -762.9006 - nll: -773.
0240 - val_kl: 10.3801 - val_loss: -772.2001 - val_nll: -782.5802
Epoch 63/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - kl: 10.4697 - loss: -803.1576 - nll: -813.
6273 - val_kl: 10.1986 - val_loss: -770.2116 - val_nll: -780.4103
Epoch 64/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - kl: 10.2120 - loss: -803.8167 - nll: -814.
0287 - val_kl: 9.8290 - val_loss: -773.8790 - val_nll: -783.7079
Epoch 65/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - kl: 10.1675 - loss: -797.5135 - nll: -807.
6810 - val_kl: 10.2830 - val_loss: -776.2853 - val_nll: -786.5682
Epoch 66/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - kl: 10.3077 - loss: -787.8779 - nll: -798.
1855 - val_kl: 10.2649 - val_loss: -774.7249 - val_nll: -784.9899
Epoch 67/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - kl: 10.2700 - loss: -798.0064 - nll: -808.
2764 - val_kl: 10.1084 - val_loss: -776.2153 - val_nll: -786.3237
```

```
Epoch 68/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - kl: 10.3277 - loss: -799.6561 - nll: -809.
9838 - val_kl: 9.8556 - val_loss: -767.0026 - val_nll: -776.8582
Epoch 69/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - kl: 10.1793 - loss: -802.3119 - nll: -812.
4911 - val_kl: 10.0749 - val_loss: -780.1770 - val_nll: -790.2520
Epoch 70/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - kl: 10.2012 - loss: -808.7274 - nll: -818.
9285 - val_kl: 9.9825 - val_loss: -783.3392 - val_nll: -793.3217
Epoch 71/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - kl: 10.0680 - loss: -808.9020 - nll: -818.
9700 - val_kl: 10.0016 - val_loss: -778.4155 - val_nll: -788.4172
Epoch 72/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - kl: 9.9695 - loss: -806.8411 - nll: -816.8
105 - val_kl: 9.7090 - val_loss: -780.6629 - val_nll: -790.3719
Epoch 73/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - kl: 9.9707 - loss: -790.8351 - nll: -800.8
057 - val_kl: 9.9377 - val_loss: -780.3218 - val_nll: -790.2596
Epoch 74/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - kl: 10.0911 - loss: -800.9391 - nll: -811.
0303 - val_kl: 9.7348 - val_loss: -784.8478 - val_nll: -794.5825
Epoch 75/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - kl: 9.8420 - loss: -800.1721 - nll: -810.0
140 - val_kl: 9.3694 - val_loss: -768.0602 - val_nll: -777.4296
Epoch 76/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - kl: 9.6625 - loss: -787.0721 - nll: -796.7
347 - val_kl: 9.3039 - val_loss: -768.4344 - val_nll: -777.7383
Epoch 77/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - kl: 9.5776 - loss: -796.4427 - nll: -806.0
203 - val_kl: 9.5101 - val_loss: -784.4468 - val_nll: -793.9569
Epoch 78/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 28ms/step - kl: 9.8602 - loss: -811.1672 - nll: -821.0
274 - val_kl: 10.0442 - val_loss: -781.6166 - val_nll: -791.6608
Epoch 79/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - kl: 9.9959 - loss: -812.8063 - nll: -822.8
022 - val_kl: 9.7850 - val_loss: -772.8030 - val_nll: -782.5880
Epoch 80/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - kl: 9.7770 - loss: -806.7320 - nll: -816.5
090 - val_kl: 9.6886 - val_loss: -789.6354 - val_nll: -799.3240
Epoch 81/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - kl: 9.8141 - loss: -811.6374 - nll: -821.4
514 - val_kl: 9.7835 - val_loss: -788.5890 - val_nll: -798.3724
Epoch 82/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - kl: 9.8310 - loss: -810.3112 - nll: -820.1
422 - val_kl: 9.5103 - val_loss: -792.5886 - val_nll: -802.0989
Epoch 83/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - kl: 9.7302 - loss: -816.3682 - nll: -826.0
984 - val_kl: 9.6603 - val_loss: -788.3881 - val_nll: -798.0484
Epoch 84/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - kl: 9.7586 - loss: -817.6969 - nll: -827.4
556 - val_kl: 9.6160 - val_loss: -791.1253 - val_nll: -800.7413
Epoch 85/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - kl: 9.7075 - loss: -815.5018 - nll: -825.2
094 - val_kl: 9.5316 - val_loss: -792.3756 - val_nll: -801.9071
Epoch 86/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - kl: 9.6196 - loss: -805.0212 - nll: -814.6
408 - val_kl: 9.7754 - val_loss: -782.6927 - val_nll: -792.4681
Epoch 87/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - kl: 9.8170 - loss: -812.1644 - nll: -821.9
815 - val_kl: 9.8602 - val_loss: -786.2432 - val_nll: -796.1034
Epoch 88/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - kl: 9.8065 - loss: -814.5376 - nll: -824.3
441 - val_kl: 9.6441 - val_loss: -797.3018 - val_nll: -806.9459
Epoch 89/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - kl: 9.6987 - loss: -821.1002 - nll: -830.7
988 - val_kl: 9.3623 - val_loss: -796.3478 - val_nll: -805.7101
Epoch 90/200
```

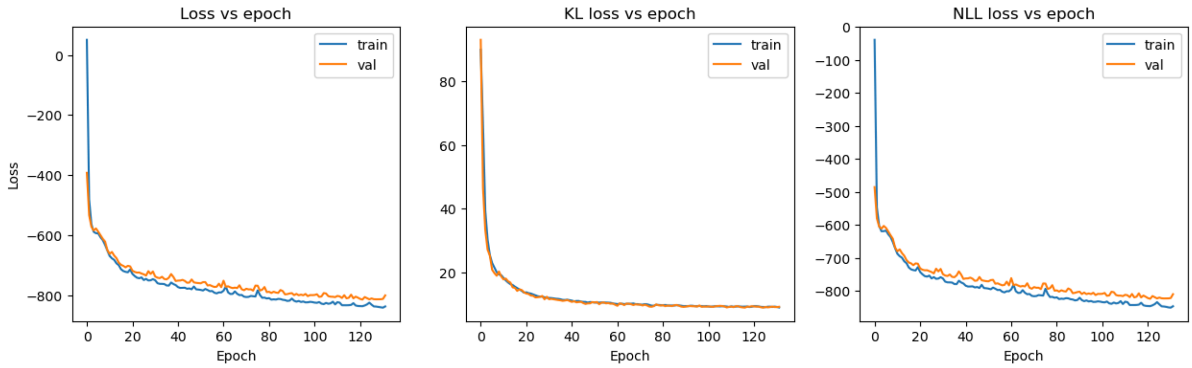
```
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - kl: 9.5656 - loss: -824.2831 - nll: -833.8
487 - val_kl: 9.2619 - val_loss: -794.2492 - val_nll: -803.5111
Epoch 91/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - kl: 9.4976 - loss: -812.9114 - nll: -822.4
091 - val_kl: 9.3437 - val_loss: -793.0164 - val_nll: -802.3600
Epoch 92/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - kl: 9.4857 - loss: -817.8033 - nll: -827.2
890 - val_kl: 9.5047 - val_loss: -799.8885 - val_nll: -809.3931
Epoch 93/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - kl: 9.6282 - loss: -825.7262 - nll: -835.3
544 - val_kl: 9.5272 - val_loss: -795.5555 - val_nll: -805.0827
Epoch 94/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - kl: 9.6589 - loss: -823.1824 - nll: -832.8
413 - val_kl: 9.4766 - val_loss: -800.2495 - val_nll: -809.7262
Epoch 95/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - kl: 9.4960 - loss: -823.5439 - nll: -833.0
399 - val_kl: 9.4700 - val_loss: -798.1188 - val_nll: -807.5888
Epoch 96/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - kl: 9.4788 - loss: -820.9158 - nll: -830.3
947 - val_kl: 9.3604 - val_loss: -800.3260 - val_nll: -809.6863
Epoch 97/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - kl: 9.5538 - loss: -827.2442 - nll: -836.7
980 - val_kl: 9.6967 - val_loss: -795.8373 - val_nll: -805.5339
Epoch 98/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - kl: 9.5708 - loss: -821.3774 - nll: -830.9
482 - val_kl: 9.2928 - val_loss: -803.1716 - val_nll: -812.4645
Epoch 99/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - kl: 9.4655 - loss: -822.4689 - nll: -831.9
344 - val_kl: 9.3770 - val_loss: -799.6027 - val_nll: -808.9798
Epoch 100/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - kl: 9.5466 - loss: -828.8375 - nll: -838.3
840 - val_kl: 9.3376 - val_loss: -799.8305 - val_nll: -809.1681
Epoch 101/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - kl: 9.4045 - loss: -826.8464 - nll: -836.2
508 - val_kl: 9.4119 - val_loss: -799.2919 - val_nll: -808.7040
Epoch 102/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - kl: 9.3704 - loss: -831.2337 - nll: -840.6
041 - val_kl: 9.3567 - val_loss: -796.8973 - val_nll: -806.2540
Epoch 103/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - kl: 9.3577 - loss: -821.0459 - nll: -830.4
037 - val_kl: 9.2779 - val_loss: -805.5562 - val_nll: -814.8340
Epoch 104/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - kl: 9.5228 - loss: -835.5372 - nll: -845.0
599 - val_kl: 9.4387 - val_loss: -804.3253 - val_nll: -813.7639
Epoch 105/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - kl: 9.4810 - loss: -825.3038 - nll: -834.7
849 - val_kl: 9.5230 - val_loss: -792.9158 - val_nll: -802.4389
Epoch 106/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - kl: 9.4141 - loss: -827.0316 - nll: -836.4
457 - val_kl: 9.5914 - val_loss: -794.3239 - val_nll: -803.9152
Epoch 107/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - kl: 9.4433 - loss: -828.2502 - nll: -837.6
935 - val_kl: 9.5316 - val_loss: -803.0966 - val_nll: -812.6283
Epoch 108/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - kl: 9.4660 - loss: -825.1511 - nll: -834.6
171 - val_kl: 9.4126 - val_loss: -804.9681 - val_nll: -814.3807
Epoch 109/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - kl: 9.4652 - loss: -823.7717 - nll: -833.2
369 - val_kl: 9.0969 - val_loss: -805.2094 - val_nll: -814.3063
Epoch 110/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - kl: 9.2710 - loss: -827.3738 - nll: -836.6
448 - val_kl: 9.2581 - val_loss: -806.5124 - val_nll: -815.7706
Epoch 111/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - kl: 9.3778 - loss: -823.6774 - nll: -833.0
552 - val_kl: 9.3647 - val_loss: -802.4485 - val_nll: -811.8133
Epoch 112/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - kl: 9.3662 - loss: -821.6652 - nll: -831.0
```

```
315 - val_kl: 9.1988 - val_loss: -803.9826 - val_nll: -813.1814
Epoch 113/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - kl: 9.3111 - loss: -828.6627 - nll: -837.9
739 - val_kl: 9.4771 - val_loss: -809.4242 - val_nll: -818.9012
Epoch 114/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - kl: 9.4746 - loss: -830.7944 - nll: -840.2
689 - val_kl: 9.4643 - val_loss: -799.8790 - val_nll: -809.3434
Epoch 115/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - kl: 9.3897 - loss: -831.3996 - nll: -840.7
893 - val_kl: 9.2081 - val_loss: -810.4515 - val_nll: -819.6597
Epoch 116/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - kl: 9.3147 - loss: -837.1895 - nll: -846.5
042 - val_kl: 9.1060 - val_loss: -808.5238 - val_nll: -817.6298
Epoch 117/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - kl: 9.3753 - loss: -836.0165 - nll: -845.3
918 - val_kl: 9.0375 - val_loss: -798.2362 - val_nll: -807.2737
Epoch 118/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - kl: 9.4081 - loss: -828.3523 - nll: -837.7
603 - val_kl: 9.3380 - val_loss: -810.4990 - val_nll: -819.8370
Epoch 119/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - kl: 9.2772 - loss: -842.4493 - nll: -851.7
264 - val_kl: 9.3076 - val_loss: -804.7904 - val_nll: -814.0980
Epoch 120/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - kl: 9.2532 - loss: -833.3672 - nll: -842.6
204 - val_kl: 9.4805 - val_loss: -808.2543 - val_nll: -817.7349
Epoch 121/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - kl: 9.4678 - loss: -830.4432 - nll: -839.9
109 - val_kl: 9.4099 - val_loss: -812.4645 - val_nll: -821.8745
Epoch 122/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - kl: 9.5259 - loss: -843.0267 - nll: -852.5
526 - val_kl: 9.3369 - val_loss: -814.5245 - val_nll: -823.8613
Epoch 123/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - kl: 9.4729 - loss: -841.4292 - nll: -850.9
021 - val_kl: 9.3514 - val_loss: -806.1848 - val_nll: -815.5361
Epoch 124/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step - kl: 9.2935 - loss: -833.5746 - nll: -842.8
682 - val_kl: 9.1475 - val_loss: -810.0343 - val_nll: -819.1818
Epoch 125/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - kl: 9.2145 - loss: -819.3718 - nll: -828.5
863 - val_kl: 9.0292 - val_loss: -813.7303 - val_nll: -822.7595
Epoch 126/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - kl: 9.1484 - loss: -829.1292 - nll: -838.2
775 - val_kl: 9.2539 - val_loss: -810.9033 - val_nll: -820.1572
Epoch 127/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - kl: 9.3182 - loss: -835.7309 - nll: -845.0
491 - val_kl: 9.1182 - val_loss: -813.3168 - val_nll: -822.4349
Epoch 128/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - kl: 9.2954 - loss: -831.2030 - nll: -840.4
984 - val_kl: 9.2065 - val_loss: -813.6806 - val_nll: -822.8871
Epoch 129/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - kl: 9.2659 - loss: -842.8499 - nll: -852.1
158 - val_kl: 9.3582 - val_loss: -813.4840 - val_nll: -822.8423
Epoch 130/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - kl: 9.4276 - loss: -841.3474 - nll: -850.7
750 - val_kl: 9.2242 - val_loss: -813.4524 - val_nll: -822.6766
Epoch 131/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - kl: 9.2754 - loss: -842.6168 - nll: -851.8
922 - val_kl: 9.2061 - val_loss: -812.5172 - val_nll: -821.7233
Epoch 132/200
18/18 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - kl: 9.1347 - loss: -841.4671 - nll: -850.6
017 - val_kl: 9.2920 - val_loss: -800.6252 - val_nll: -809.9171
```

In [12]:

```
# Plot the learning curves

fig = plt.figure(figsize=(15, 4))
fig.add_subplot(1, 3, 1)
plt.plot(history.history['loss'], label='train')
```

```python
plt.plot(history.history['val_loss'], label='val')
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Loss vs epoch")
plt.legend()

fig.add_subplot(1, 3, 2)
plt.plot(history.history['kl'], label='train')
plt.plot(history.history['val_kl'], label='val')
plt.xlabel("Epoch")
plt.title("KL loss vs epoch")
plt.legend()

fig.add_subplot(1, 3, 3)
plt.plot(history.history['nll'], label='train')
plt.plot(history.history['val_nll'], label='val')
plt.xlabel("Epoch")
plt.title("NLL loss vs epoch")
plt.legend()

plt.show()
```




In [13]:
```python
# Evaluate performance on the validation set

vae.evaluate(val_dataloader, return_dict=True)
```

```
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - kl: 9.2344 - loss: -800.0823 - nll: -809.31
67
```

Out[13]:
```
{'kl': 9.291972160339355,
 'loss': -798.3985595703125,
 'nll': -807.6905517578125}
```

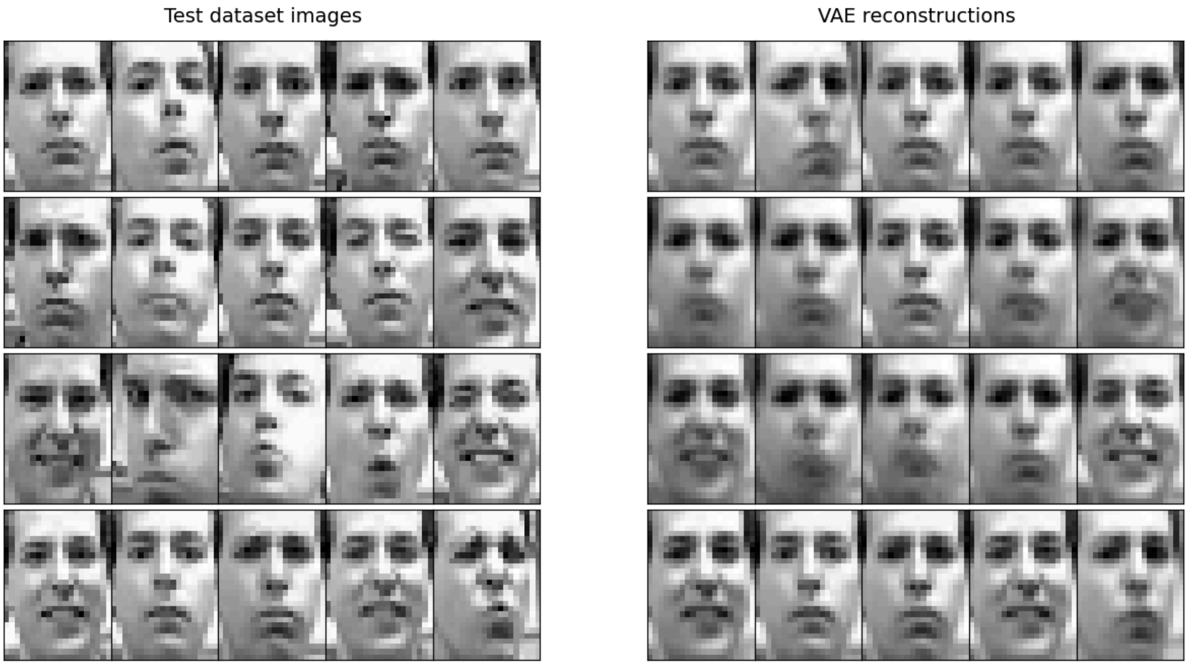
## View samples and reconstructions

In [14]:
```python
# Sample from the generative model

samples = ops.convert_to_numpy(vae.decoder(keras.random.normal(shape=(40, 2)))[0])
```

In [15]:
```python
# View the samples

n_rows, n_cols = 4, 10
fig, axes = plt.subplots(n_rows, n_cols, figsize=(14, 8))
fig.subplots_adjust(hspace=0., wspace=0.)

for n, image in enumerate(samples):
    row = n // n_cols
    col = n % n_cols
    axes[row, col].imshow(image, cmap='gray')
    axes[row, col].get_xaxis().set_visible(False)
    axes[row, col].get_yaxis().set_visible(False)
plt.show()
```

In [16]:
```
# Compute reconstructions from the validation dataset

images = next(iter(val_dataloader))[0]
reconstructions = ops.convert_to_numpy(vae(images)[0])
```

In [17]:
```
# Plot some reconstructions from the test dataset

import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

fig = plt.figure(figsize=(15, 8))
outer = gridspec.GridSpec(1, 2, hspace=0.2)
n_rows, n_cols = 4, 5
fig.text(0.23, 0.9, "Test dataset images", fontsize=14)
fig.text(0.66, 0.9, "VAE reconstructions", fontsize=14)
for i in range(2):
    inner = gridspec.GridSpecFromSubplotSpec(n_rows, n_cols,
                    subplot_spec=outer[i], wspace=0., hspace=0.)
    display_images = [images, reconstructions][i]
    for j in range(n_rows * n_cols):
        row = j // n_cols
        col = j % n_cols
        ax = plt.Subplot(fig, inner[j])
        ax.imshow(display_images[j], cmap='gray')
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
        fig.add_subplot(ax)
```




*Exercise.* Rewrite the loss function above so the KL divergence is approximated with Monte Carlo samples, so the SGVB estimator $\hat{\mathcal{L}}^{A}(\theta, \phi; x)$ is used instead. Also try modifying the posterior to be a full covariance Gaussian. Does this improve the model performance?

## References


- Benou, A., Veksler, R., Friedman, A. & Raviv, T.R. (2016), "De-noising of contrast-enhanced MRI sequences by an ensemble of expert deep neural networks", in *International Workshop on Deep Learning in Medical Image Analysis*, Athens, Greece, 21 October 2016.
- Blei, D. M., Kucukelbir, A. & McAuliffe, J. D. (2017), "Variational Inference: A Review for Statisticians", *Journal of the American Statistical Association*, **112** (518), 859-877.
- Blei, D. M., Jordan, M. I. & Paisley, J. W. (2012), "Variational bayesian inference with stochastic search", in *Proceedings of the 29th International Conference on Machine Learning (ICML)*, 1367–1374.
- Devroye, L. (1996), "Random Variate Generation in One Line of Code", in *Proceedings of the 28th Conference on Winter Simulation*, Coronado, California, USA, 265-272.
- Dhariwal, P., Heewoo, J., Payne, C., Kim, J. W., Radford, A. & Sutskever, I. (2020), "Jukebox: A Generative Model for Music", arXiv preprint, abs/2005.00341.
- Glynn, P. W. (1990), "Likelihood Ratio Gradient Estimation for Stochastic Systems", *Communications of the ACM*, **33** (10), 75-84.
- Hoffman, M. D., Blei, D. M., Wang, C, & Paisley, J. (2013), "Stochastic Variational Inference", *Journal of Machine Learning Research*, **14** (1), 1532-4435.
- Jin, W., Barzilay, R. & Jaakkola, T. (2018), "Junction Tree Variational Autoencoder for Molecular Graph Generation", in *Proceedings of Machine Learning Research*, **80**, 2323-2332.
- Kingma, D. P. & Welling, M., "Auto-Encoding Variational Bayes" (2014), in *Proceedings of the 2nd International Conference on Learning Representations (ICLR)*, Banff, AB, Canada, April 14-16, 2014.
- Kleijnen, J. P. C. & Rubinstein, R. Y. (1996), "Optimization and sensitivity analysis of computer simulation models by the score function method", *European Journal of Operational Research*, **88**, 413-427 .
- Kramer, M. A. (1991), "Nonlinear principal component analysis using autoassociative neural networks", *AIChE Journal*, **37** (2), 233–243.

- Rezende, D. J., Mohamed, S. & Wierstra, D. (2014), "Stochastic Backpropagation and Approximate Inference in Deep Generative Models", in *Proceedings of the 31st International Conference on Machine Learning, PMLR*, **32** (2), 1278-1286.
- Salakhutdinov, R. and Murray, I. (2008), "On the quantitative analysis of deep belief networks", in *Proceedings of the 25th international conference on Machine learning*, 892-879.
- Vahdat, A. and Kautz, J. (2020), "NVAE: A Deep Hierarchical Variational Autoencoder", in *Proceedings of the 34th International Conference on Neural Information Processing Systems*.
- Vincent, P. & Larochelle, H. (2010), "Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion", *Journal of Machine Learning Research*, **11**, 3371–3408.
- Williams, R. J. (1992), "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning", *Machine Learning*, **8** (3-4), 229-256.
- Xu, H., Chen, W., Zhao, N., Li, Z., Bu, J., Li, Z., Liu, Y., Zhao, Y., Pei, D., Feng, Y., Chen, J. J., Wanb, Z. & Qiao, H. (2018), "Unsupervised Anomaly Detection via Variational Auto-Encoder for Seasonal KPIs in Web Applications", *Proceedings of the 2018 World Wide Web Conference*, Palais des congrès de Lyon, Lyon, France, 23-27 April 2018.