Capstone Project

Probabilistic generative models

Instructions

In this notebook, you will practice working with generative models, using both normalising flow networks and the variational autoencoder algorithm. You will create a synthetic dataset with a normalising flow with randomised parameters. This dataset will then be used to train a variational autoencoder, and you will used the trained model to interpolate between the generated images. You will use concepts from throughout this course, including Distribution objects, probabilistic layers, bijectors, ELBO optimisation and KL divergence regularisers.

This project is peer-assessed. Within this notebook you will find instructions in each section for how to complete the project. Pay close attention to the instructions as the peer review will be carried out according to a grading rubric that checks key parts of the project instructions. Feel free to add extra cells into the notebook as required.

How to submit

When you have completed the Capstone project notebook, you will submit a pdf of the notebook for peer review. First ensure that the notebook has been fully executed from beginning to end, and all of the cell outputs are visible. This is important, as the grading rubric depends on the reviewer being able to view the outputs of your notebook. Save the notebook as a pdf (File -> Download as -> PDF via LaTeX). You should then submit this pdf for review.

Let's get started!

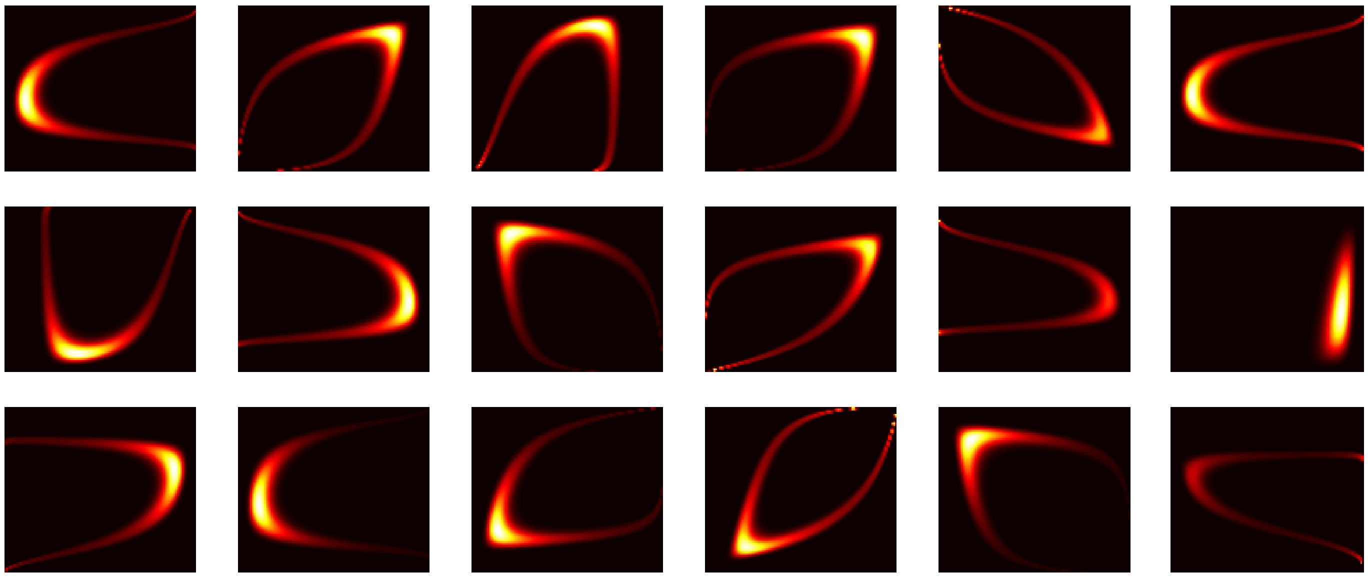
We'll start by running some imports below. For this project you are free to make further imports throughout the notebook as you wish.

In [1]:

```
#!pip install --upgrade --user tensorflow
#!pip install --upgrade --user tensorflow_probability
#!pip install matplotlib==3.2.2
```

In [1]:

```
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions
tfb = tfp.bijectors
tfpl = tfp.layers

import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```



For the capstone project, you will create your own image dataset from contour plots of a transformed distribution using a random normalising flow network. You will then use the variational autoencoder algorithm to train generative and inference networks, and synthesise new images by interpolating in the latent space.

The normalising flow

- To construct the image dataset, you will build a normalising flow to transform the 2-D Gaussian random variable $z = (z_1, z_2)$, which has mean $\mathbf{0}$ and covariance matrix $\Sigma = \sigma^2 \mathbf{I}_2$, with $\sigma = 0.3$.
- This normalising flow uses bijectors that are parameterised by the following random variables:
  - $\theta \sim U[0, 2\pi)$
  - $a \sim N(3, 1)$

The complete normalising flow is given by the following chain of transformations:

- $f_1(z) = (z_1, z_2 - 2)$,
- $f_2(z) = (z_1, \frac{z_2}{2})$,
- $f_3(z) = (z_1, z_2 + az_1^2)$,
- $f_4(z) = Rz$, where R is a rotation matrix with angle θ,
- $f_5(z) = \tanh(z)$, where the $\tanh$ function is applied elementwise.

The transformed random variable x is given by $x = f_5(f_4(f_3(f_2(f_1(z)))))$.

- You should use or construct bijectors for each of the transformations f_i, $i = 1, \ldots, 5$, and use `tfb.Chain` and `tfb.TransformedDistribution` to construct the final transformed distribution.
- Ensure to implement the `log_det_jacobian` methods for any subclassed bijectors that you write.
- Display a scatter plot of samples from the base distribution.
- Display 4 scatter plot images of the transformed distribution from your random normalising flow, using samples of θ and a. Fix the axes of these 4 plots to the range $[-1, 1]$.

In [3]:

```python
def plot_distribution(samples, ax, title, col='red'):
    ax.set_facecolor("black")
    ax.scatter(samples[:, 0], samples[:, 1], marker='.', c=col, alpha=0.5) #edgecolor='k',
    ax.set_xlim([-1,1])
    ax.set_ylim([-1,1])
    ax.set_title(title, size=15)
```

In [4]:

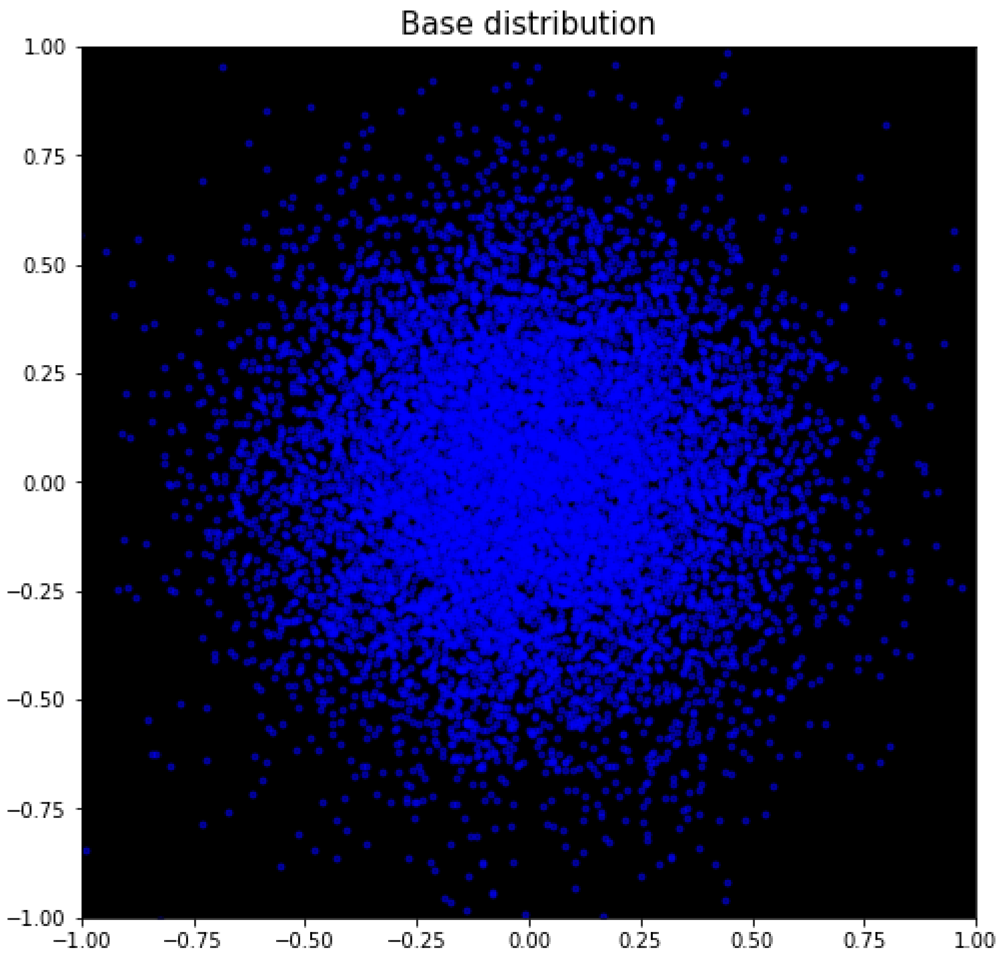
```
# f3(z)=(z1,z2+az1^2)
class Degree2Polynomial(tfb.Bijector):

    def __init__(self, a):
        self.a = a
        super(Degree2Polynomial, self).__init__(forward_min_event_ndims=1, is_constant_jaco

    def _forward(self, x):
        return tf.concat([x[..., :1], x[..., 1:] + self.a * tf.square(x[..., :1])], axis=-1

    def _inverse(self, y):
        return tf.concat([y[..., :1], y[..., 1:] - self.a * tf.square(y[..., :1])], axis=-1

    def _forward_log_det_jacobian(self, x):
        return tf.constant(0., dtype=x.dtype)


# f4(z)=Rz
class Rotation(tfb.Bijector):

    def __init__(self, theta):
        self.R = tf.constant([[np.cos(theta), -np.sin(theta)],
                              [np.sin(theta), np.cos(theta)]], dtype=tf.float32)
        super(Rotation, self).__init__(forward_min_event_ndims=1, is_constant_jacobian=True

    def _forward(self, x):
        return tf.linalg.matvec(self.R, x)

    def _inverse(self, y):
        return tf.linalg.matvec(tf.transpose(self.R), y)

    def _forward_log_det_jacobian(self, x):
        return tf.constant(0., x.dtype)
```

In [5]:

```
def get_normalizing_flow_dist(a, theta):
    bijectors = [
                    tfb.Shift([0.,-2]), # f1
                    tfb.Scale([1,1/2]), # f2
                    Degree2Polynomial(a),      # f3
                    Rotation(theta),    # f4
                    tfb.Tanh()          # f5
               ]
    flow_bijector = tfb.Chain(list(reversed(bijectors)))
    return tfd.TransformedDistribution(distribution=base_distribution,
                                       bijector=flow_bijector)
```

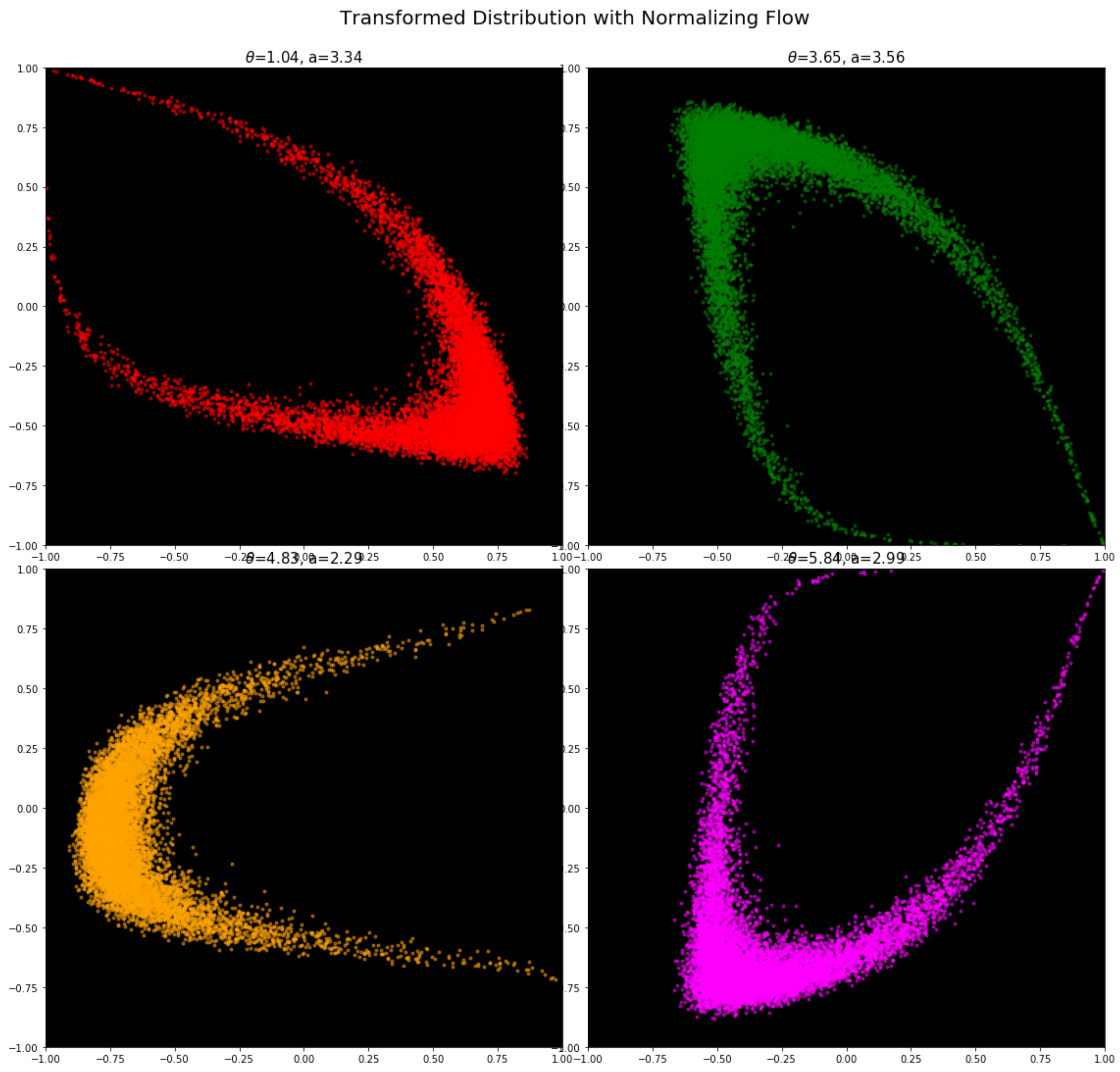
In [6]:

```
nsamples= 10000
sigma = 0.3
base_distribution = tfd.MultivariateNormalDiag(loc=tf.zeros(2), scale_diag=sigma*tf.ones(2)
samples = base_distribution.sample(nsamples)
fig, ax = plt.subplots(figsize=(8,8))
plot_distribution(samples, ax, 'Base distribution', 'blue')
plt.show()
```



In [16]:

```
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(15,15))
axes = axes.flatten()
plt.subplots_adjust(0, 0, 1, 0.925, 0.05, 0.05)
colors = ['red', 'green', 'orange', 'magenta']
for i in range(4):
    a = tfd.Normal(loc=3, scale=1).sample(1)[0].numpy()
    theta = tfd.Uniform(low = 0, high = 2*np.pi).sample(1)[0].numpy()
    transformed_distribution = get_normalizing_flow_dist(a, theta)
    samples = transformed_distribution.sample(nsamples)
    plot_distribution(samples, axes[i], r'$\theta$={:.02f}, a={:.02f}'.format(theta, a), co
plt.suptitle('Transformed Distribution with Normalizing Flow', size=20)
plt.show()
```



2. Create the image dataset

- You should now use your random normalising flow to generate an image dataset of contour plots from your random normalising flow network.
  - Feel free to get creative and experiment with different architectures to produce different sets of images!
- First, display a sample of 4 contour plot images from your normalising flow network using 4 independently sampled sets of parameters.
  - You may find the following `get_densities` function useful: this calculates density values for a (batched) Distribution for use in a contour plot.
- Your dataset should consist of at least 1000 images, stored in a numpy array of shape `(N, 36, 36, 3)`. Each image in the dataset should correspond to a contour plot of a transformed distribution from a normalising flow with an independently sampled set of parameters s, T, S, b. It will take a few minutes to create the dataset.
- As well as the `get_densities` function, the `get_image_array_from_density_values` function will help you to generate the dataset.
  - This function creates a numpy array for an image of the contour plot for a given set of density values Z. Feel free to choose your own options for the contour plots.
- Display a sample of 20 images from your generated dataset in a figure.

In [7]:

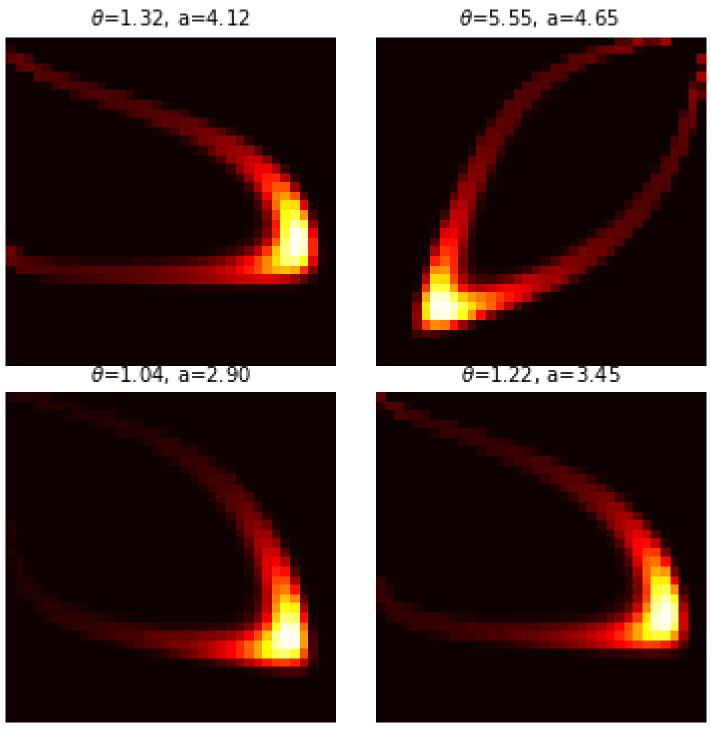
```
# Helper function to compute transformed distribution densities

X, Y = np.meshgrid(np.linspace(-1, 1, 100), np.linspace(-1, 1, 100))
inputs = np.transpose(np.stack((X, Y)), [1, 2, 0])

def get_densities(transformed_distribution):
    """
    This function takes a (batched) Distribution object as an argument, and returns a numpy
    array Z of shape (batch_shape, 100, 100) of density values, that can be used to make a
    contour plot with:
    plt.contourf(X, Y, Z[b, ...], cmap='hot', levels=100)
    where b is an index into the batch shape.
    """
    batch_shape = transformed_distribution.batch_shape
    Z = transformed_distribution.prob(np.expand_dims(inputs, 2))
    Z = np.transpose(Z, list(range(2, 2+len(batch_shape))) + [0, 1])
    return Z
```

In [8]:

```
# Helper function to convert contour plots to numpy arrays

import numpy as np
from matplotlib.backends.backend_agg import FigureCanvasAgg as FigureCanvas
from matplotlib.figure import Figure

def get_image_array_from_density_values(Z):
    """
    This function takes a numpy array Z of density values of shape (100, 100)
    and returns an integer numpy array of shape (36, 36, 3) of pixel values for an image.
    """
    assert Z.shape == (100, 100)
    fig = Figure(figsize=(0.5, 0.5))
    canvas = FigureCanvas(fig)
    ax = fig.gca()
    ax.contourf(X, Y, Z, cmap='hot', levels=100)
    ax.axis('off')
    fig.tight_layout(pad=0)

    ax.margins(0)
    fig.canvas.draw()
    image_from_plot = np.frombuffer(fig.canvas.tostring_rgb(), dtype=np.uint8)
    image_from_plot = image_from_plot.reshape(fig.canvas.get_width_height()[::-1] + (3,))
    return image_from_plot
```

In [11]:

```
plt.figure(figsize=(5,5))
plt.subplots_adjust(0, 0, 1, 0.95, 0.05, 0.08)
for i in range(4):
    a = tfd.Normal(loc=3, scale=1).sample(1)[0].numpy()
    theta = tfd.Uniform(low = 0, high = 2*np.pi).sample(1)[0].numpy()
    transformed_distribution = get_normalizing_flow_dist(a, theta)
    transformed_distribution = tfd.BatchReshape(transformed_distribution, [1])
    Z = get_densities(transformed_distribution)
    image = get_image_array_from_density_values(Z.squeeze())
    plt.subplot(2,2,i+1), plt.imshow(image), plt.axis('off')
    plt.title(r'$\theta$={:.02f}, a={:.02f}'.format(theta, a), size=10)
plt.show()
```



In [12]:

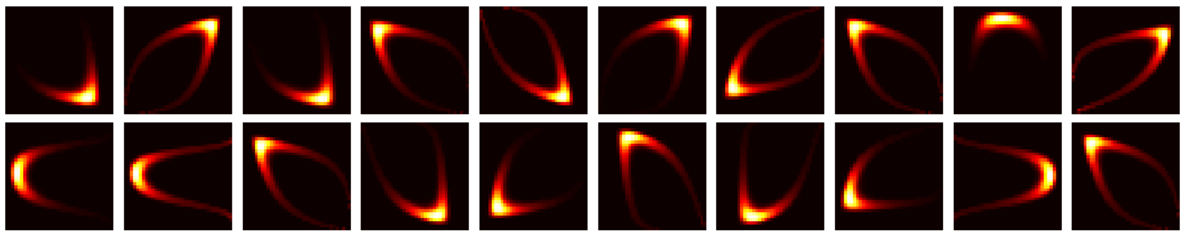
```
N = 1000
image_dataset = np.zeros((N, 36, 36, 3))
for i in range(N):
    a = tfd.Normal(loc=3, scale=1).sample(1)[0].numpy()
    theta = tfd.Uniform(low = 0, high = 2*np.pi).sample(1)[0].numpy()
    transformed_distribution = tfd.BatchReshape(get_normalizing_flow_dist(a, theta), [1])
    image_dataset[i,...] = get_image_array_from_density_values(get_densities(transformed_di
image_dataset = tf.convert_to_tensor(image_dataset, dtype=tf.float32)
image_dataset.shape
```

Out[12]:

```
TensorShape([1000, 36, 36, 3])
```

In [14]:

```
plt.figure(figsize=(20,4))
plt.subplots_adjust(0, 0, 1, 0.95, 0.05, 0.08)
indices = np.random.choice(N, 20)
for i in range(20):
    image = image_dataset[indices[i]].numpy()
    image = image / image.max()
    plt.subplot(2,10,i+1), plt.imshow(image), plt.axis('off')
plt.show()
```



3. Make `tf.data.Dataset` objects

- You should now split your dataset to create `tf.data.Dataset` objects for training and validation data.
- Using the `map` method, normalise the pixel values so that they lie between 0 and 1.
- These Datasets will be used to train a variational autoencoder (VAE). Use the `map` method to return a tuple of input and output Tensors where the image is duplicated as both input and output.
- Randomly shuffle the training Dataset.
- Batch both datasets with a batch size of 20, setting `drop_remainder=True` .
- Print the `element_spec` property for one of the Dataset objects.

In [15]:

```
n = len(image_dataset)
tf_image_dataset = tf.data.Dataset.from_tensor_slices(image_dataset)
tf_image_dataset = tf_image_dataset.shuffle(3)
tf_image_dataset = tf_image_dataset.map(lambda x : x / tf.reduce_max(x))
tf_image_dataset = tf_image_dataset.map(lambda x: (x, x))
```

In [16]:

```
train_sz = int(0.8*n)
training = tf_image_dataset.take(train_sz)
validation = tf_image_dataset.skip(train_sz)
```

In [17]:

```
training = training.batch(batch_size=20, drop_remainder=True)
validation = validation.batch(batch_size=20, drop_remainder=True)
training.element_spec
```

Out[17]:

```
(TensorSpec(shape=(20, 36, 36, 3), dtype=tf.float32, name=None),
 TensorSpec(shape=(20, 36, 36, 3), dtype=tf.float32, name=None))
```

4. Build the encoder and decoder networks

- You should now create the encoder and decoder for the variational autoencoder algorithm.

- You should design these networks yourself, subject to the following constraints:
  - The encoder and decoder networks should be built using the `Sequential` class.
  - The encoder and decoder networks should use probabilistic layers where necessary to represent distributions.
  - The prior distribution should be a zero-mean, isotropic Gaussian (identity covariance matrix).
  - The encoder network should add the KL divergence loss to the model.
- Print the model summary for the encoder and decoder networks.

In [2]:

```
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import (Dense, Flatten, Reshape, Concatenate, Conv2D, UpSampli
```

In [3]:

```
latent_dim = 2 #50
prior = tfd.MultivariateNormalDiag(loc=tf.zeros(latent_dim))

def get_kl_regularizer(prior_distribution):
    return tfpl.KLDivergenceRegularizer(prior_distribution,
                                        weight=1.0,
                                        use_exact_kl=False,
                                        test_points_fn=lambda q: q.sample(3),
                                        test_points_reduce_axis=(0,1))

kl_regularizer = get_kl_regularizer(prior)

def get_encoder(latent_dim, kl_regularizer):
    return Sequential([
            Conv2D(filters=32, kernel_size=3, activation='relu', strides=2, padding='same',
            BatchNormalization(),
            Conv2D(filters=64, kernel_size=3, activation='relu', strides=2, padding='same')
            BatchNormalization(),
            Conv2D(filters=128, kernel_size=3, activation='relu', strides=3, padding='same'
            BatchNormalization(),
            Flatten(),
            Dense(tfpl.MultivariateNormalTriL.params_size(latent_dim)),
            tfpl.MultivariateNormalTriL(latent_dim, activity_regularizer=kl_regularizer)
        ], name='encoder')

def get_decoder(latent_dim):
    return Sequential([
        Dense(1152, activation='relu', input_shape=(latent_dim,)),
        Reshape((3,3,128)),
        UpSampling2D(size=(3,3)),
        Conv2D(filters=64, kernel_size=3, activation='relu', padding='same'),
        UpSampling2D(size=(2,2)),
        Conv2D(filters=32, kernel_size=2, activation='relu', padding='same'),
        UpSampling2D(size=(2,2)),
        Conv2D(filters=128, kernel_size=2, activation='relu', padding='same'),
        Conv2D(filters=3, kernel_size=2, activation=None, padding='same'),
        Flatten(),
        tfpl.IndependentBernoulli(event_shape=(36,36,3))
    ], name='decoder')
```

In [4]:

```
encoder = get_encoder(latent_dim=2, kl_regularizer=kl_regularizer)
#encoder.losses
encoder.summary()
```

```
Model: "encoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d (Conv2D)              (None, 18, 18, 32)        896       
_________________________________________________________________
batch_normalization (BatchNo (None, 18, 18, 32)        128       
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 9, 9, 64)          18496     
_________________________________________________________________
batch_normalization_1 (Batch (None, 9, 9, 64)          256       
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 3, 3, 128)         73856     
_________________________________________________________________
batch_normalization_2 (Batch (None, 3, 3, 128)         512       
_________________________________________________________________
flatten (Flatten)            (None, 1152)              0         
_________________________________________________________________
dense (Dense)                (None, 5)                 5765      
_________________________________________________________________
multivariate_normal_tri_l (M ((None, 2), (None, 2))    0         
=================================================================
Total params: 99,909
Trainable params: 99,461
Non-trainable params: 448
_________________________________________________________________
```

In [5]:

```
decoder = get_decoder(latent_dim=2)
decoder.summary(line_length=70)
```

```
Model: "decoder"
______________________________________________________________________
Layer (type)                Output Shape                Param #
======================================================================
dense_1 (Dense)             (None, 1152)                3456
______________________________________________________________________
reshape (Reshape)           (None, 3, 3, 128)           0
______________________________________________________________________
up_sampling2d (UpSampling2D)  (None, 9, 9, 128)         0
______________________________________________________________________
conv2d_3 (Conv2D)           (None, 9, 9, 64)            73792
______________________________________________________________________
up_sampling2d_1 (UpSampling2D) (None, 18, 18, 64)       0
______________________________________________________________________
conv2d_4 (Conv2D)           (None, 18, 18, 32)          8224
______________________________________________________________________
up_sampling2d_2 (UpSampling2D) (None, 36, 36, 32)       0
______________________________________________________________________
conv2d_5 (Conv2D)           (None, 36, 36, 128)         16512
______________________________________________________________________
conv2d_6 (Conv2D)           (None, 36, 36, 3)           1539
______________________________________________________________________
flatten_1 (Flatten)         (None, 3888)                0
______________________________________________________________________
independent_bernoulli (Indepen ((None, 36, 36, 3), (None,  0
======================================================================
Total params: 103,523
Trainable params: 103,523
Non-trainable params: 0
______________________________________________________________________
```

In [7]:

```
def reconstruction_loss(batch_of_images, decoding_dist):
    return -tf.reduce_mean(decoding_dist.log_prob(batch_of_images))
```

5. Train the variational autoencoder

- You should now train the variational autoencoder. Build the VAE using the `Model` class and the encoder and decoder models. Print the model summary.
- Compile the VAE with the negative log likelihood loss and train with the `fit` method, using the training and validation Datasets.
- Plot the learning curves for loss vs epoch for both training and validation sets.

In [8]:

```
vae = Model(inputs=encoder.inputs, outputs=decoder(encoder.outputs))
vae.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_3_input (InputLayer)  [(None, 36, 36, 3)]       0         
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 18, 18, 32)        896       
_________________________________________________________________
batch_normalization_3 (Batch (None, 18, 18, 32)        128       
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 9, 9, 64)          18496     
_________________________________________________________________
batch_normalization_4 (Batch (None, 9, 9, 64)          256       
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 3, 3, 128)         73856     
_________________________________________________________________
batch_normalization_5 (Batch (None, 3, 3, 128)         512       
_________________________________________________________________
flatten_1 (Flatten)          (None, 1152)              0         
_________________________________________________________________
dense_1 (Dense)              (None, 5)                 5765      
_________________________________________________________________
multivariate_normal_tri_l_1  ((None, 2), (None, 2))    0         
_________________________________________________________________
decoder (Sequential)         (None, 36, 36, 3)         103523    
=================================================================
Total params: 203,432
Trainable params: 202,984
Non-trainable params: 448
_________________________________________________________________
```

In [ ]:

```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.0005)
vae.compile(optimizer=optimizer, loss=reconstruction_loss)
```

In [130]:

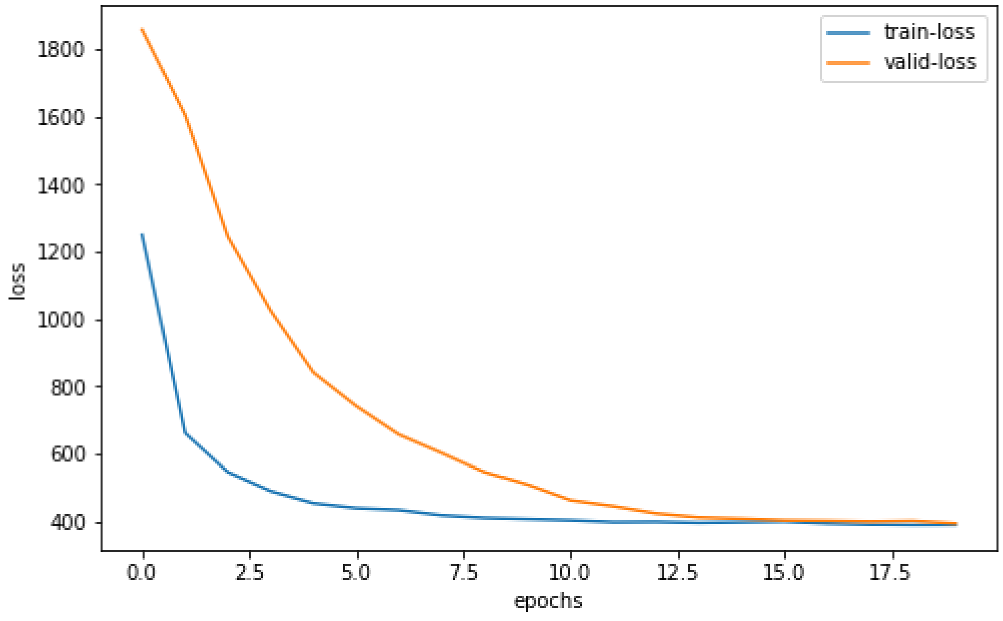
```
history = vae.fit(training, validation_data=validation, epochs=20)
```

```
Epoch 1/20
40/40 [==============================] - 34s 777ms/step - loss: 1250.2296 -
val_loss: 1858.7103
Epoch 2/20
40/40 [==============================] - 29s 731ms/step - loss: 661.8687 - v
al_loss: 1605.1261
Epoch 3/20
40/40 [==============================] - 29s 720ms/step - loss: 545.2802 - v
al_loss: 1245.0518
Epoch 4/20
40/40 [==============================] - 28s 713ms/step - loss: 489.1101 - v
al_loss: 1024.5863
Epoch 5/20
40/40 [==============================] - 29s 718ms/step - loss: 453.3464 - v
al_loss: 841.4725
Epoch 6/20
40/40 [==============================] - 29s 733ms/step - loss: 438.8413 - v
al_loss: 742.0212
Epoch 7/20
40/40 [==============================] - 30s 751ms/step - loss: 433.2563 - v
al_loss: 657.4024
Epoch 8/20
40/40 [==============================] - 30s 751ms/step - loss: 417.5353 - v
al_loss: 602.7039
Epoch 9/20
40/40 [==============================] - 29s 726ms/step - loss: 409.8351 - v
al_loss: 545.5004
Epoch 10/20
40/40 [==============================] - 30s 741ms/step - loss: 406.3284 - v
al_loss: 507.9868
Epoch 11/20
40/40 [==============================] - 30s 741ms/step - loss: 402.9056 - v
al_loss: 462.0777
Epoch 12/20
40/40 [==============================] - 29s 733ms/step - loss: 397.4801 - v
al_loss: 444.4444
Epoch 13/20
40/40 [==============================] - 30s 741ms/step - loss: 398.2078 - v
al_loss: 423.1287
Epoch 14/20
40/40 [==============================] - 29s 723ms/step - loss: 395.5187 - v
al_loss: 411.3030
Epoch 15/20
40/40 [==============================] - 30s 739ms/step - loss: 397.3987 - v
al_loss: 407.5134
Epoch 16/20
40/40 [==============================] - 29s 721ms/step - loss: 399.3271 - v
al_loss: 402.7288
Epoch 17/20
40/40 [==============================] - 29s 736ms/step - loss: 393.4259 - v
al_loss: 401.4711
Epoch 18/20
40/40 [==============================] - 29s 726ms/step - loss: 390.5508 - v
al_loss: 399.1924
Epoch 19/20
40/40 [==============================] - 29s 736ms/step - loss: 389.3187 - v
al_loss: 401.1656
```

```
Epoch 20/20
40/40 [==============================] - 29s 728ms/step - loss: 389.4718 - v
al_loss: 393.5178
```

In [134]:

```
nepochs = 20
plt.figure(figsize=(8,5))
plt.plot(range(nepochs), history.history['loss'], label='train-loss')
plt.plot(range(nepochs), history.history['val_loss'], label='valid-loss')
plt.legend()
plt.xlabel('epochs')
plt.ylabel('loss')
plt.show()
```



6. Use the encoder and decoder networks

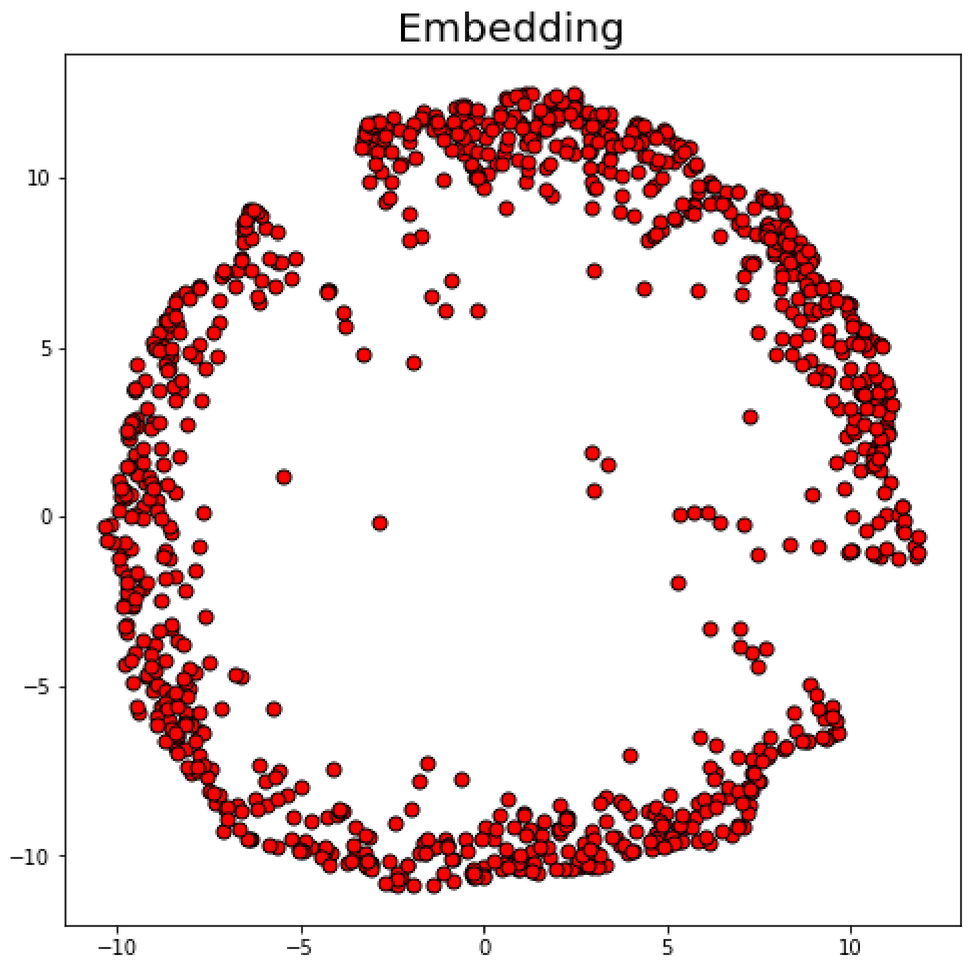
- You can now put your encoder and decoder networks into practice!
- Randomly sample 1000 images from the dataset, and pass them through the encoder. Display the embeddings in a scatter plot (project to 2 dimensions if the latent space has dimension higher than two).
- Randomly sample 4 images from the dataset and for each image, display the original and reconstructed image from the VAE in a figure.
  - Use the mean of the output distribution to display the images.
- Randomly sample 6 latent variable realisations from the prior distribution, and display the images in a figure.
  - Again use the mean of the output distribution to display the images.

In [135]:

```
def reconstruct(encoder, decoder, batch_of_images):
    approx_distribution = encoder(batch_of_images)
    decoding_dist = decoder(approx_distribution.mean())
    return decoding_dist.mean()
```

In [137]:

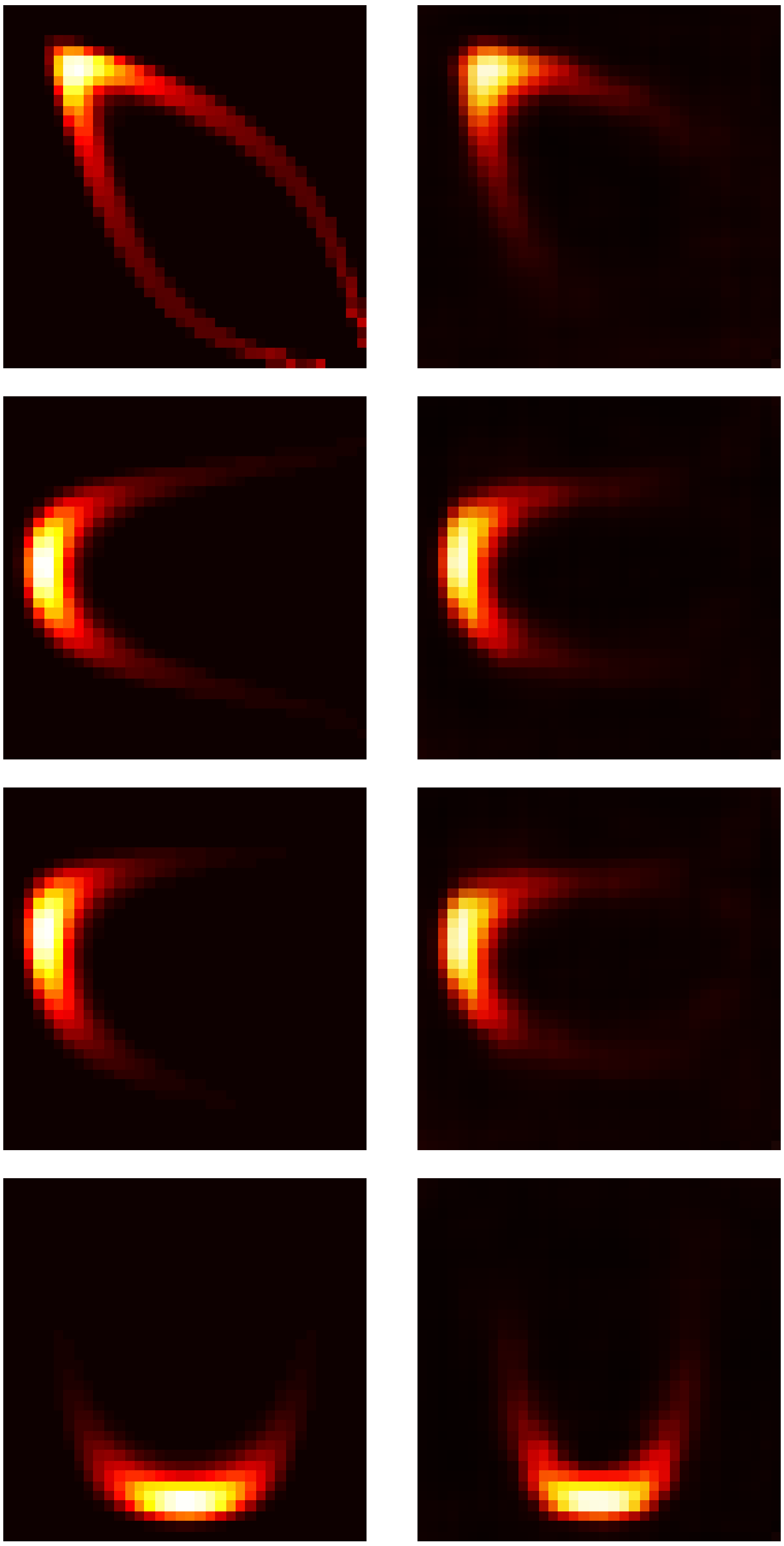
```
embedding = encoder(image_dataset / 255).mean()
fig, ax = plt.subplots(figsize=(8,8))
plt.scatter(embedding[:,0], embedding[:,1], c='red', s=50, edgecolor='k')
plt.title('Embedding', size=20)
plt.show()
```



In [138]:

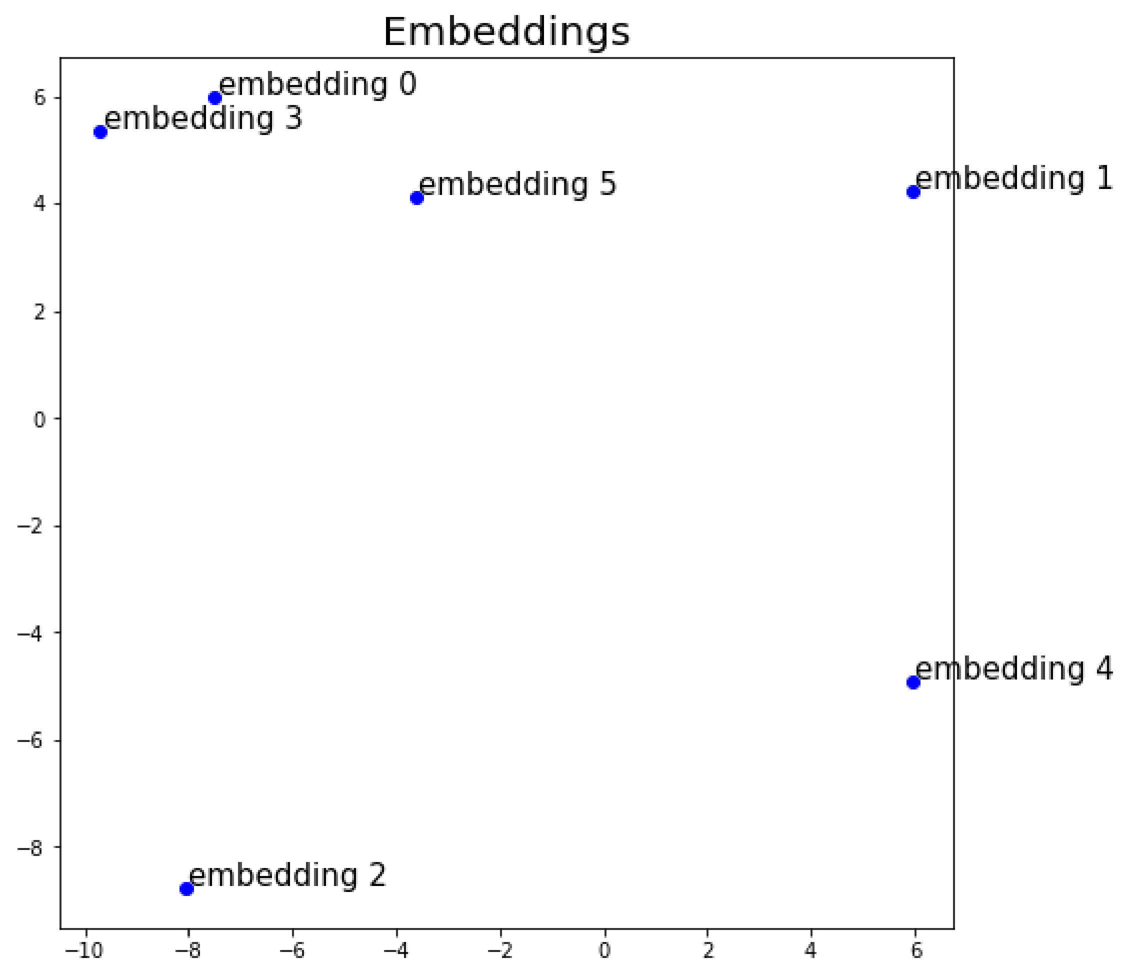
```
plt.figure(figsize=(6,12))
plt.subplots_adjust(0, 0, 1, 0.95, 0.05, 0.08)
indices = np.random.choice(len(image_dataset), 4)
for i in range(4):
    image = image_dataset[indices[i]].numpy()
    image = image / image.max()
    plt.subplot(4,2,2*i+1), plt.imshow(image), plt.axis('off')
    reconstructions = reconstruct(encoder, decoder, np.expand_dims(image, axis=0))
    plt.subplot(4,2,2*i+2), plt.imshow(reconstructions[0].numpy()), plt.axis('off')
plt.suptitle('original (left column) vs. VAE-reconstructed (right column)', size=15)
plt.show()
```

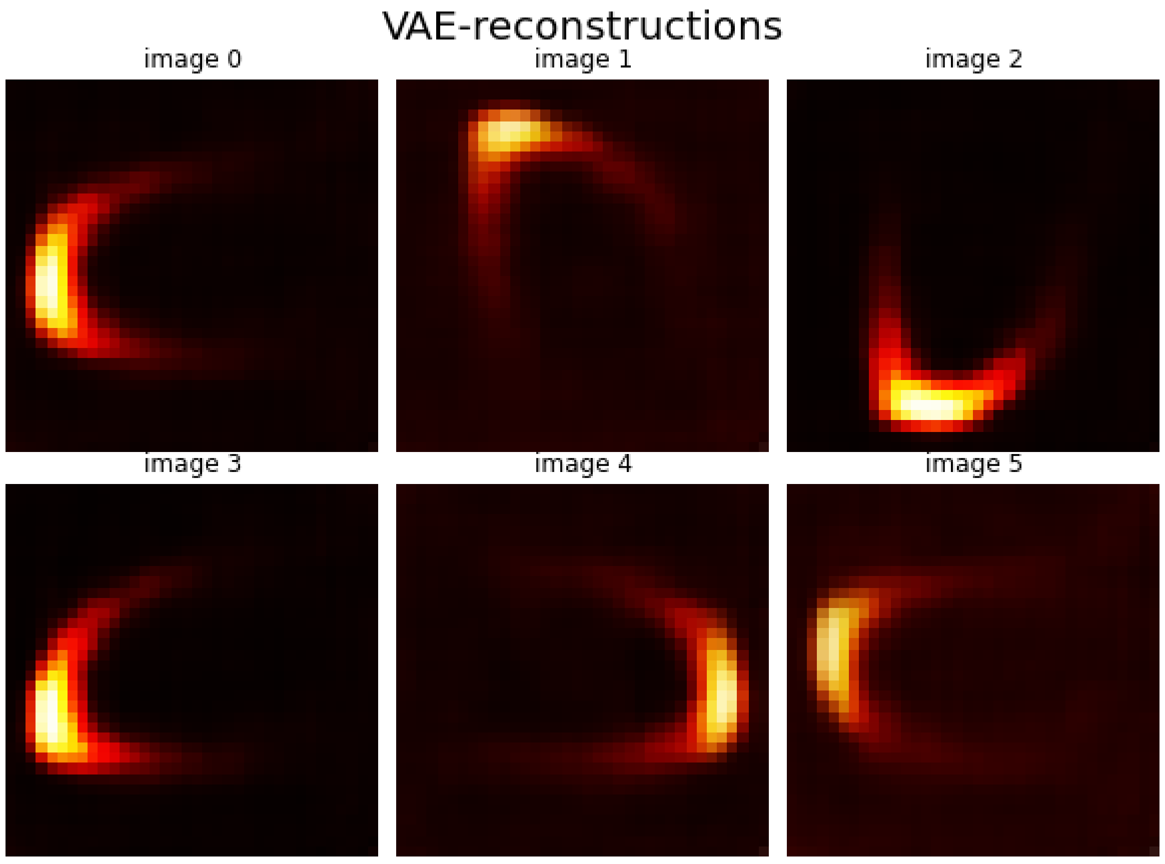
original (left column) vs. VAE-reconstructed (right column)



In [143]:

```
nsample = 6
samples = np.random.uniform(-10, 10, (nsample, latent_dim)) #prior.sample(6)
fig, ax = plt.subplots(figsize=(8,8))
plt.scatter(samples[:,0], samples[:,1], color='blue')
for i in range(nsample):
    plt.text(samples[i,0] + 0.05, samples[i,1] + 0.05, 'embedding {}'.format(i), fontsize=1
plt.title('Embeddings', size=20)
plt.show()
reconstructions = decoder(samples).mean()
#print(samples.shape, reconstructions.shape)
plt.figure(figsize=(8,6))
plt.subplots_adjust(0, 0, 1, 0.9, 0.05, 0.08)
indices = np.random.choice(len(image_dataset), 4)
for i in range(nsample):
    plt.subplot(2,3,i+1), plt.imshow(reconstructions[i]), plt.title('image {}'.format(i)),
plt.suptitle('VAE-reconstructions', size=20)
plt.show()
```





Make a video of latent space interpolation (not assessed)

- Just for fun, you can run the code below to create a video of your decoder's generations, depending on the latent space.

In [11]:

```
# Function to create animation

import matplotlib.animation as anim
from IPython.display import HTML


def get_animation(latent_size, decoder, interpolation_length=500):
    assert latent_size >= 2, "Latent space must be at least 2-dimensional for plotting"
    fig = plt.figure(figsize=(9, 4))
    ax1 = fig.add_subplot(1,2,1)
    ax1.set_xlim([-6, 6])
    ax1.set_ylim([-6, 6])
    ax1.set_title("Latent space")
    ax1.axes.get_xaxis().set_visible(False)
    ax1.axes.get_yaxis().set_visible(False)
    ax2 = fig.add_subplot(1,2,2)
    ax2.set_title("Data space")
    ax2.axes.get_xaxis().set_visible(False)
    ax2.axes.get_yaxis().set_visible(False)

    # initializing a line variable
    line, = ax1.plot([], [], marker='o')
    img2 = ax2.imshow(np.zeros((36, 36, 3)))

    freqs = np.random.uniform(low=0.1, high=0.2, size=(latent_size,))
    phases = np.random.randn(latent_size)
    input_points = np.arange(interpolation_length)
    latent_coords = []
    for i in range(latent_size):
        latent_coords.append(2 * np.sin((freqs[i]*input_points + phases[i])).astype(np.floa

    def animate(i):
        z = tf.constant([coord[i] for coord in latent_coords])
        img_out = np.squeeze(decoder(z[np.newaxis, ...]).mean().numpy())
        line.set_data(z.numpy()[0], z.numpy()[1])
        img2.set_data(np.clip(img_out, 0, 1))
        return (line, img2)

    return anim.FuncAnimation(fig, animate, frames=interpolation_length,
                              repeat=False, blit=True, interval=150)
```

In [ ]:

```
# Create the animation
latent_size = 2
a = get_animation(latent_size, decoder, interpolation_length=200)
HTML(a.to_html5_video())
```