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 [185]:

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

from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split

import numpy as np
import matplotlib.pyplot as plt
plt.style.use('seaborn-white')
%matplotlib inline

from tqdm import tqdm
```

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 0

and covariance matrix $\Sigma = \sigma^2 I_2$
, 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:

- $f_1(z) = (z_1, z_2 - 2)$
  ,
- $f_2(z) = (z_1, \frac{z_2}{2})$
  ,
- $f_3(z) = (z_1, z_2 + a z_1^2)$
  ,
- $f_4(z) = Rz$
  , 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]:

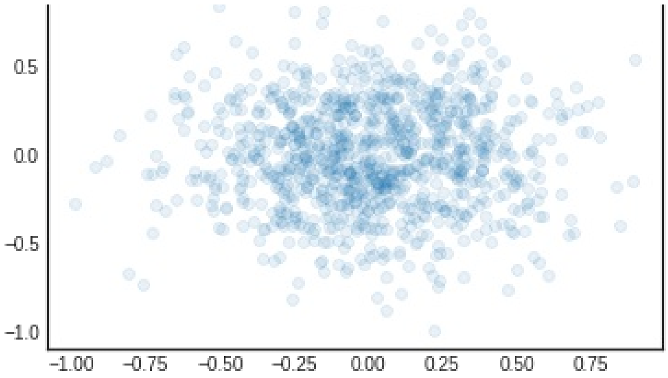
```
# Define distributions to parametrize normalizing flow
dist_theta = tfd.Uniform(0, 2*np.pi)
dist_a = tfd.Normal(loc=3, scale=1)
```

In [4]:

```
# Define the starting distribution
starting_dist = tfd.MultivariateNormalDiag(loc=0., scale_diag=[.3, .3])
```

In [5]:

```
# Plot the starting distribution
starting_samples = starting_dist.sample(1000).numpy()

plt.scatter(starting_samples[:, 0], starting_samples[:, 1], alpha=.1)
plt.title('Original (starting) distribution')
plt.show()
```



In [6]:

```
# Define bijector helpers

def get_rotation_matrix(theta):

    """Takes an angle theta and returns a rotation matrix wrapped in a tf linear opertato
r"""

    rotation_matrix = tf.convert_to_tensor([[tf.cos(theta), -tf.sin(theta)],
                                           [tf.sin(theta), tf.cos(theta)]])

    return tf.linalg.LinearOperatorFullMatrix(rotation_matrix)


class PolyBijector(tfb.Bijector):

    def __init__(self, a, name='poly_bijector', **kwargs):
        super(PolyBijector, self).__init__(forward_min_event_ndims=1,
                                           name=name,
                                           is_constant_jacobian=True,
                                           validate_args=False,
                                           **kwargs)

        self.a = tf.cast(a, dtype=tf.float32)

    def _forward(self, x):
        x = tf.cast(x, dtype=tf.float32)
        return tf.concat([x[..., 0:1],
                          x[..., 1:] + self.a * tf.square(x[..., 0:1])], axis=-1)

    def _inverse(self, y):
        y = tf.cast(y, dtype=tf.float32)
        return tf.concat([y[..., 0:1],
                          y[..., 1:] - self.a * tf.square(y[..., 0:1])], axis=-1)

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

In [7]:

```
# Define the chain
def get_bi_chain(a, theta):
    bi_chain = tfb.Chain([
        tfb.Tanh(),
        tfb.ScaleMatvecLinearOperator(get_rotation_matrix(theta)),
        PolyBijector(a),
        tfb.Scale([1., .5]),
        tfb.Shift([0., -2.])
    ])
    return bi_chain
```

In [8]:

```
# Initialize the transformed distribution
def get_transformed_distribution(starting_dist, a, theta):
    return tfd.TransformedDistribution(starting_dist, get_bi_chain(a, theta))
```

In [9]:

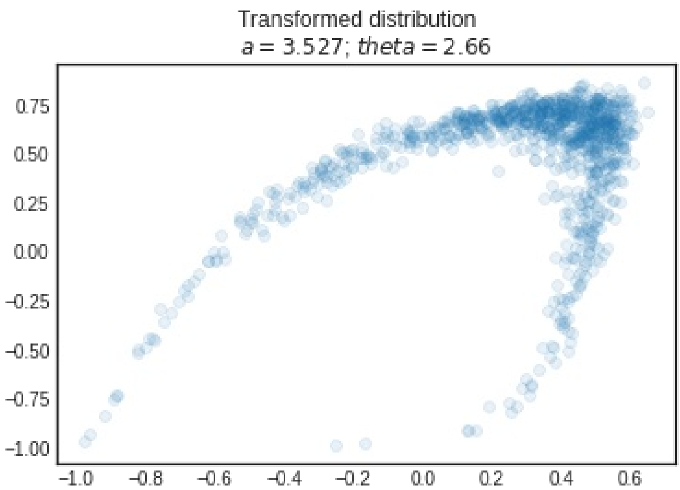
```
get_transformed_distribution(starting_dist, 2., .5)
```

Out[9]:

```
<tfp.distributions.TransformedDistribution 'chain_of_tanh_of_scale_matvec_linear_operator
_of_poly_bijector_of_scale_of_shiftMultivariateNormalDiag' batch_shape=[] event_shape=[2]
dtype=float32>
```

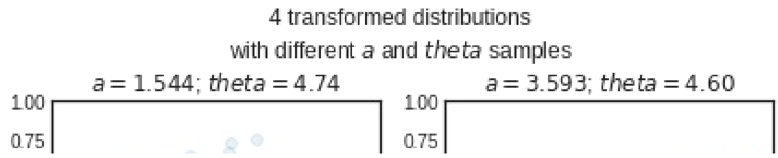
In [10]:

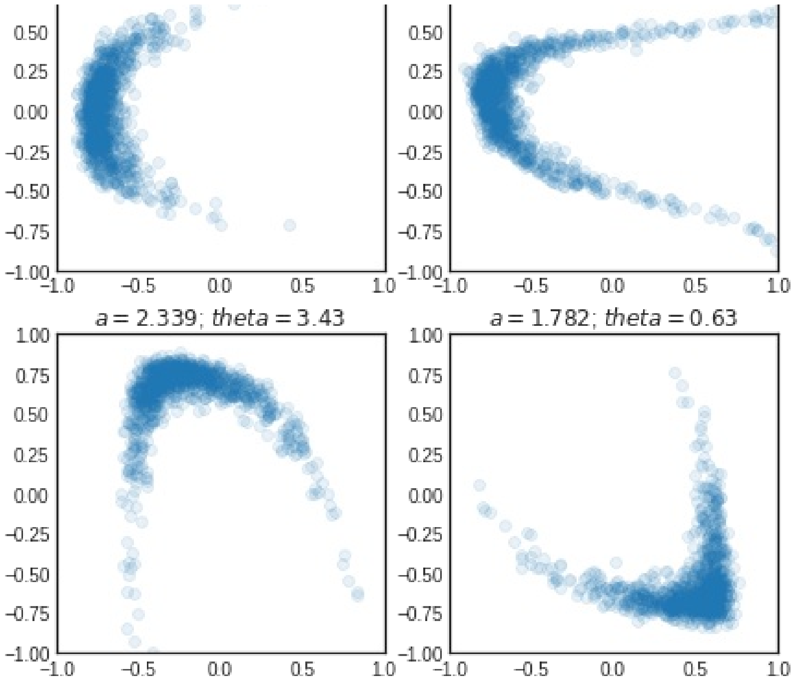
```
# Display a scatter plot of samples from the base distribution.
val_a = dist_a.sample()
val_theta = dist_theta.sample()

transformed_distribution = get_transformed_distribution(starting_dist, val_a, val_theta)

td_samples = transformed_distribution.sample(1000).numpy()

plt.scatter(td_samples[:, 0], td_samples[:, 1], alpha=.1)
plt.suptitle('Transformed distribution')
plt.title(f'$a = {val_a:.3f}$; $theta = {val_theta:.2f}$')
plt.show()
```



In [11]:

```
# Display 4 scatter plot images of the transformed distribution from your random normalis
ing flow,
# using samples of □ and a. Fix the axes of these 4 plots to the range [-1,1].

plt.figure(figsize=(7, 7))

for i in range(4):
    val_a = dist_a.sample()
    val_theta = dist_theta.sample()

    transformed_distribution = get_transformed_distribution(starting_dist, val_a, val_the
ta)
    td_samples = transformed_distribution.sample(1000).numpy()
    plt.subplot(int(f'22{i + 1}'))
    plt.xlim(-1, 1)
    plt.ylim(-1, 1)
    plt.scatter(td_samples[:, 0], td_samples[:, 1], alpha=.1)
    plt.title(f'$a = {val_a:.3f}$; $theta = {val_theta:.2f}$')


plt.suptitle('4 transformed distributions\nwith different $a$ and $theta$ samples')
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 [12]:

```
# 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 num
py
    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 [13]:

```
# 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
```

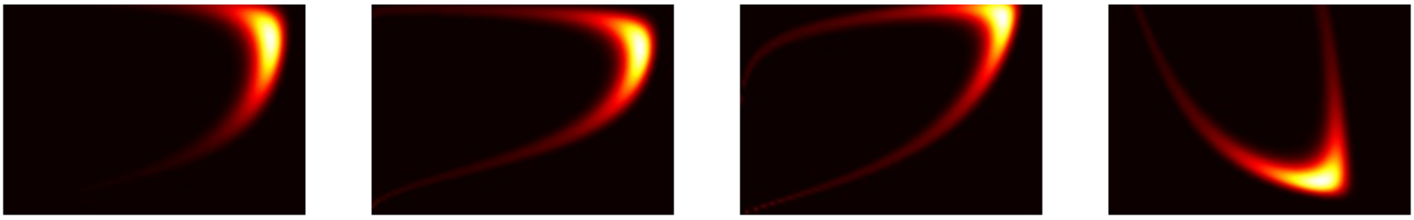
In [14]:

```
# Turn off solver efficiency warnings
tf.get_logger().setLevel('ERROR')
```

Display a sample of 4 contour plot images from your normalising flow network using 4 independently sampled sets of parameters.

In [15]:

```
# Plot 4 contours

plt.figure(figsize = (18, 8))

for i in range(4):

    # Get params
    val_a = dist_a.sample()
    val_theta = dist_theta.sample()

    # Initialize the dist
    transformed_distribution = get_transformed_distribution(starting_dist, val_a, val_the
ta)
    transformed_distribution = tfd.BatchReshape(transformed_distribution, [1])

    # Setup subplot
    plt.subplot(int(f'24{i + 1}'))
    plt.axis('off')

    # Plot contuors
    plt.contourf(X, Y, get_densities(transformed_distribution).squeeze(), cmap='hot', lev
els=100)

    # Add title
    plt.title(f'$a = {val_a:.3f}$; $theta = {val_theta:.2f}$')


plt.suptitle('4 transformed distribution contours')
plt.show()
```



Create a dataset consisting of at least 1000 images, stored in a numpy array of shape (N, 36, 36, 3).

In [17]:

```
# Generate the dataset
dataset = []
dataset_params = []
N = 1000

for i in tqdm(range(N)):

    # Get params
    val_a = dist_a.sample()
    val_theta = dist_theta.sample()

    # Store params for future ref
    dataset_params.append({'a': val_a, 'theta': val_theta})

    # Initialize the dist
    transformed_distribution = get_transformed_distribution(starting_dist, val_a, val_the
ta)
    transformed_distribution = tfd.BatchReshape(transformed_distribution, [1])

    # Get densities
    densities = get_densities(transformed_distribution).squeeze()

    # Store images
    dataset.append(get_image_array_from_density_values(densities))

dataset = np.array(dataset)
```

```
100%|██████████| 1000/1000 [03:27<00:00,  4.82it/s]
```

In [ ]:

```
# np.save('gen_dataset.npy', dataset)
# dataset = np.load('gen_dataset.npy')
```

In [18]:

```
# Sanity check
dataset.shape
```

Out[18]:

```
(1000, 36, 36, 3)
```

Display a sample of 20 images from your generated dataset in a figure

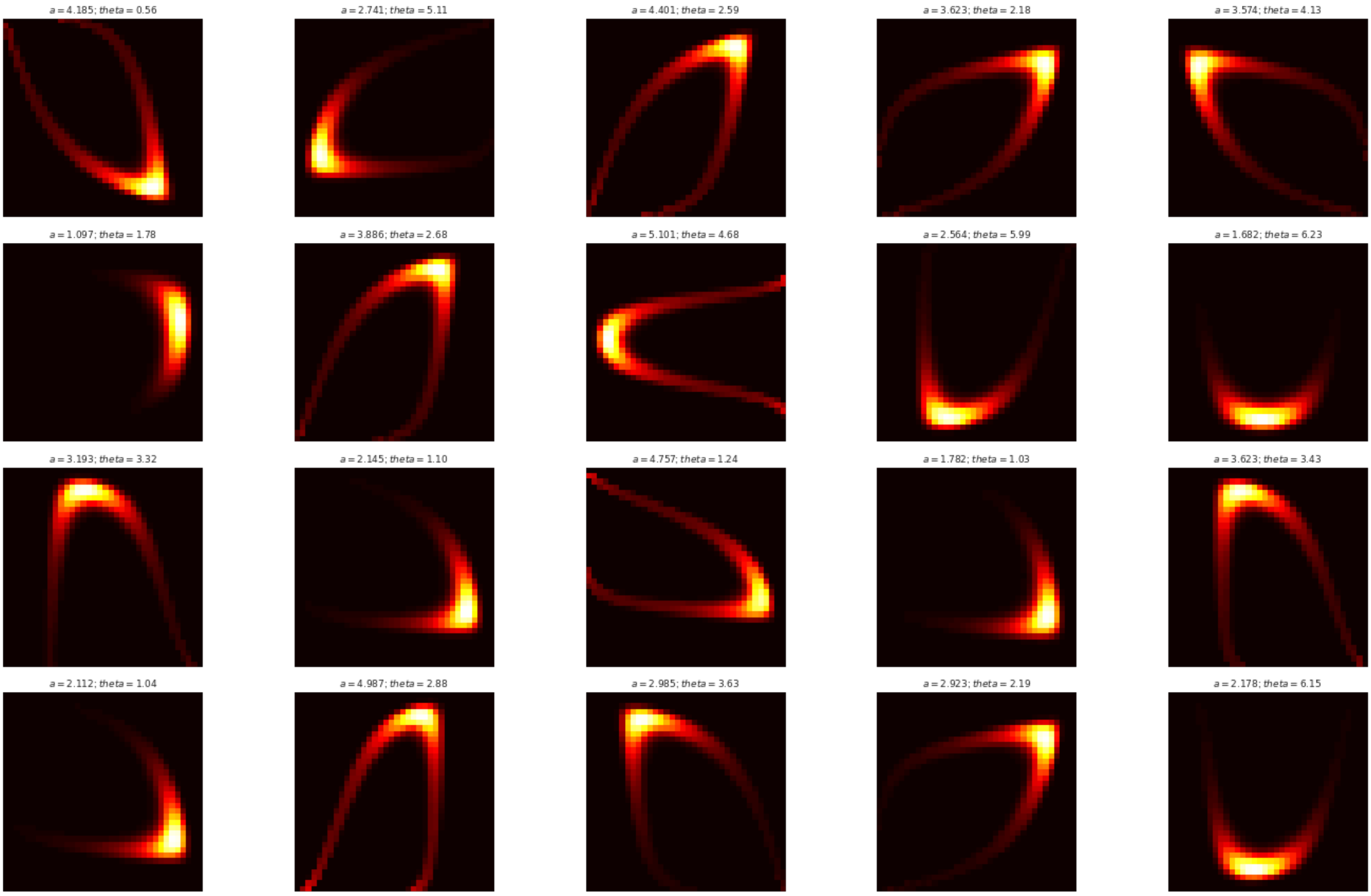
In [19]:

```
# Draw 20 images randomly
indices = np.random.choice(np.arange(dataset.shape[0]), 20)
```

In [21]:

```
# Plot

plt.figure(figsize=(20, 12))

for i, idx in enumerate(indices):
```

```
    # Get image and params
    img = dataset[idx]
    params = dataset_params[idx]

    # Setup a subplot
    plt.subplot(4, 5, int(f'{i + 1}'))
    plt.axis('off')

    # Plot contours
    plt.imshow(img)

    # Add title
    plt.title(f'$a = {params["a"]:.3f}$; $theta = {params["theta"]:.2f}$', fontsize=9)

plt.tight_layout()
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.**

You should now split your dataset

In [37]:

```
# Train-test split
X_train, X_test = train_test_split(dataset, test_size=.2)
```

In [38]:

```
# Sanity check
X_train.shape, X_test.shape
```

Out[38]:

```
((800, 36, 36, 3), (200, 36, 36, 3))
```

In [42]:

```
# Convert to float
X_train = X_train.astype(float)
X_test = X_test.astype(float)
```

Create `tf.data.Dataset` objects for training and validation data.

In [45]:

```
def get_datasets(X_train, X_test, batch_size=20):

  # Instantiate
  ds_train = tf.data.Dataset.from_tensor_slices(X_train)
  ds_test = tf.data.Dataset.from_tensor_slices(X_test)

  # Normalize
  ds_train = ds_train.map(lambda x: x / 255.)
  ds_test = ds_test.map(lambda x: x / 255.)

  # Duplicte images (x -> y)
  ds_train = ds_train.map(lambda x: (x, x))
  ds_test = ds_test.map(lambda x: (x, x))

  # Shuffle
  ds_train = ds_train.shuffle(100)
  ds_test = ds_test.shuffle(100)

  # Batch and drop remainder
  ds_train = ds_train.batch(batch_size)
  ds_test = ds_test.batch(batch_size)

  return ds_train, ds_test
```

In [46]:

```
# Get the datasets
ds_train, ds_test = get_datasets(X_train, X_test)
```

Print the element_spec property for one of the Dataset objects.

In [48]:

```
ds_train.element_spec
```

Out[48]:

```
(TensorSpec(shape=(None, 36, 36, 3), dtype=tf.float64, name=None),
 TensorSpec(shape=(None, 36, 36, 3), dtype=tf.float64, 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 [49]:

```
dataset.shape[1:]
```

Out[49]:

```
(36, 36, 3)
```

In [168]:

```
# Define the prior
latent_size = 4
prior = tfd.MultivariateNormalDiag(loc=tf.zeros(latent_size))
```

In [169]:

```
# Define the encoder
event_shape = dataset.shape[1:]

encoder = tf.keras.Sequential([
                      layers.Flatten(input_shape=event_shape),
                      layers.Dense(128, activation='selu'),
                      layers.Dense(64, activation='selu'),
                      layers.Dense(32, activation='selu'),
                      layers.Dense(tfpl.MultivariateNormalTriL.params_size(latent_size))
,
                      tfpl.MultivariateNormalTriL(latent_size),
                      tfpl.KLDivergenceAddLoss(prior,
                                               use_exact_kl=False,
                                               weight=1.5,
                                               test_points_fn=lambda q: q.sample(10),
                                               test_points_reduce_axis=0)
])
```

In [170]:

```
encoder.summary()
```

```
Model: "sequential_26"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_19 (Flatten)         (None, 3888)              0
_________________________________________________________________
dense_164 (Dense)            (None, 128)               497792
_________________________________________________________________
dense_165 (Dense)            (None, 64)                8256
_________________________________________________________________
dense_166 (Dense)            (None, 32)                2080
_________________________________________________________________
dense_167 (Dense)            (None, 14)                462
_________________________________________________________________
multivariate_normal_tri_l_7  multiple                  0
_________________________________________________________________
kl_divergence_add_loss_6 (KL multiple                  0
=================================================================
Total params: 508,590
Trainable params: 508,590
Non-trainable params: 0
_________________________________________________________________
```

In [171]:

```
# Define the decoder
decoder = tf.keras.Sequential([
    layers.Dense(16, activation='selu', input_shape=(latent_size,)),
    layers.Dense(32, activation='selu'),
    layers.Dense(64, activation='selu'),
    layers.Dense(128, activation='selu'),
    layers.Dropout(.2),
```

```
    layers.Flatten(),
    layers.Dense(tfpl.IndependentBernoulli.params_size(event_shape)),
    tfpl.IndependentBernoulli(event_shape=event_shape)
])
```

In [172]:

```
decoder.summary()
```

```
Model: "sequential_27"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_168 (Dense)            (None, 16)                80        
_________________________________________________________________
dense_169 (Dense)            (None, 32)                544       
_________________________________________________________________
dense_170 (Dense)            (None, 64)                2112      
_________________________________________________________________
dense_171 (Dense)            (None, 128)               8320      
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0         
_________________________________________________________________
flatten_20 (Flatten)         (None, 128)               0         
_________________________________________________________________
dense_172 (Dense)            (None, 3888)              501552    
_________________________________________________________________
independent_bernoulli_18 (In multiple                  0         
=================================================================
Total params: 512,608
Trainable params: 512,608
Non-trainable params: 0
_________________________________________________________________
```

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.**

You should now train the variational autoencoder. Build the VAE using the Model class and the encoder and decoder models.

In [173]:

```
# Instantiate the model
vae = tf.keras.Model(inputs=encoder.inputs, outputs=decoder(encoder.outputs))
```

Print the model summary.

In [174]:

```
vae.summary()
```

```
Model: "model_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
flatten_19_input (InputLayer [(None, 36, 36, 3)]       0         
_________________________________________________________________
flatten_19 (Flatten)         (None, 3888)              0         
_________________________________________________________________
dense_164 (Dense)            (None, 128)               497792    
_________________________________________________________________
dense_165 (Dense)            (None, 64)                8256      
```

```
_________________________________________________________________
dense_166 (Dense)            (None, 32)                2080      
_________________________________________________________________
dense_167 (Dense)            (None, 14)                462       
_________________________________________________________________
multivariate_normal_tri_l_7  multiple                  0         
_________________________________________________________________
kl_divergence_add_loss_6 (KL multiple                  0         
_________________________________________________________________
sequential_27 (Sequential)   multiple                  512608    
=================================================================
Total params: 1,021,198
Trainable params: 1,021,198
Non-trainable params: 0
_________________________________________________________________
```

Compile the VAE with the negative log likelihood loss and train with the fit method, using the training and validation Datasets.

In [175]:

```
# Compile
vae.compile(optimizer="adam", loss=lambda true, pred: -tf.reduce_mean(pred.log_prob(true
)))
```

In [176]:

```
# Define early stopping
early = tf.keras.callbacks.EarlyStopping(
    monitor="val_loss",
    min_delta=0.1,
    patience=10,
    restore_best_weights=True)

# Train
vae.fit(
    ds_train,
    validation_data=ds_test,
    callbacks=[early],
    epochs=500
    )
```

```
Epoch 1/500
40/40 [==============================] - 3s 32ms/step - loss: 1528.8812 - val_loss: 986.6
643
Epoch 2/500
40/40 [==============================] - 1s 17ms/step - loss: 859.8130 - val_loss: 745.20
34
Epoch 3/500
40/40 [==============================] - 1s 17ms/step - loss: 712.2329 - val_loss: 691.81
34
Epoch 4/500
40/40 [==============================] - 1s 16ms/step - loss: 750.8917 - val_loss: 705.20
37
Epoch 5/500
40/40 [==============================] - 1s 17ms/step - loss: 722.7554 - val_loss: 690.27
72
Epoch 6/500
40/40 [==============================] - 1s 17ms/step - loss: 688.4216 - val_loss: 683.90
80
Epoch 7/500
40/40 [==============================] - 1s 18ms/step - loss: 693.7886 - val_loss: 671.62
56
Epoch 8/500
40/40 [==============================] - 1s 18ms/step - loss: 682.1223 - val_loss: 674.86
41
Epoch 9/500
40/40 [==============================] - 1s 17ms/step - loss: 738.7687 - val_loss: 694.11
94
Epoch 10/500
40/40 [==============================] - 1s 17ms/step - loss: 665.3533 - val_loss: 653.29
```

```
41
Epoch 11/500
40/40 [==============================] - 1s 17ms/step - loss: 645.5778 - val_loss: 633.29
79
Epoch 12/500
40/40 [==============================] - 1s 18ms/step - loss: 633.4882 - val_loss: 626.89
18
Epoch 13/500
40/40 [==============================] - 1s 17ms/step - loss: 631.0038 - val_loss: 624.38
34
Epoch 14/500
40/40 [==============================] - 1s 17ms/step - loss: 629.4517 - val_loss: 621.38
84
Epoch 15/500
40/40 [==============================] - 1s 17ms/step - loss: 624.3557 - val_loss: 609.81
13
Epoch 16/500
40/40 [==============================] - 1s 18ms/step - loss: 616.5243 - val_loss: 611.55
78
Epoch 17/500
40/40 [==============================] - 1s 18ms/step - loss: 615.2584 - val_loss: 605.24
81
Epoch 18/500
40/40 [==============================] - 1s 17ms/step - loss: 613.1559 - val_loss: 607.90
26
Epoch 19/500
40/40 [==============================] - 1s 18ms/step - loss: 611.7126 - val_loss: 607.78
19
Epoch 20/500
40/40 [==============================] - 1s 17ms/step - loss: 608.8138 - val_loss: 607.73
40
Epoch 21/500
40/40 [==============================] - 1s 17ms/step - loss: 607.7188 - val_loss: 605.39
86
Epoch 22/500
40/40 [==============================] - 1s 17ms/step - loss: 606.1666 - val_loss: 604.01
17
Epoch 23/500
40/40 [==============================] - 1s 17ms/step - loss: 602.3547 - val_loss: 599.89
09
Epoch 24/500
40/40 [==============================] - 1s 18ms/step - loss: 603.8007 - val_loss: 597.47
97
Epoch 25/500
40/40 [==============================] - 1s 17ms/step - loss: 602.8115 - val_loss: 597.44
86
Epoch 26/500
40/40 [==============================] - 1s 17ms/step - loss: 601.0762 - val_loss: 594.01
51
Epoch 27/500
40/40 [==============================] - 1s 17ms/step - loss: 601.5247 - val_loss: 596.25
71
Epoch 28/500
40/40 [==============================] - 1s 18ms/step - loss: 601.1215 - val_loss: 594.50
20
Epoch 29/500
40/40 [==============================] - 1s 17ms/step - loss: 597.6091 - val_loss: 591.28
08
Epoch 30/500
40/40 [==============================] - 1s 18ms/step - loss: 599.4555 - val_loss: 592.34
06
Epoch 31/500
40/40 [==============================] - 1s 17ms/step - loss: 596.4072 - val_loss: 589.08
42
Epoch 32/500
40/40 [==============================] - 1s 17ms/step - loss: 596.3175 - val_loss: 586.40
00
Epoch 33/500
40/40 [==============================] - 1s 17ms/step - loss: 593.0443 - val_loss: 587.14
54
Epoch 34/500
40/40 [==============================] - 1s 17ms/step - loss: 589.4026 - val_loss: 594.17
```

```
61
Epoch 35/500
40/40 [==============================] - 1s 17ms/step - loss: 596.6394 - val_loss: 586.96
44
Epoch 36/500
40/40 [==============================] - 1s 17ms/step - loss: 593.9699 - val_loss: 585.02
48
Epoch 37/500
40/40 [==============================] - 1s 17ms/step - loss: 591.2084 - val_loss: 583.76
29
Epoch 38/500
40/40 [==============================] - 1s 17ms/step - loss: 581.8942 - val_loss: 572.33
22
Epoch 39/500
40/40 [==============================] - 1s 17ms/step - loss: 579.9634 - val_loss: 567.34
78
Epoch 40/500
40/40 [==============================] - 1s 18ms/step - loss: 577.5246 - val_loss: 570.63
55
Epoch 41/500
40/40 [==============================] - 1s 17ms/step - loss: 575.6866 - val_loss: 566.62
21
Epoch 42/500
40/40 [==============================] - 1s 17ms/step - loss: 570.9831 - val_loss: 561.76
21
Epoch 43/500
40/40 [==============================] - 1s 18ms/step - loss: 568.7010 - val_loss: 554.48
41
Epoch 44/500
40/40 [==============================] - 1s 17ms/step - loss: 566.1334 - val_loss: 553.21
24
Epoch 45/500
40/40 [==============================] - 1s 17ms/step - loss: 557.4902 - val_loss: 555.63
67
Epoch 46/500
40/40 [==============================] - 1s 18ms/step - loss: 562.6343 - val_loss: 546.53
40
Epoch 47/500
40/40 [==============================] - 1s 17ms/step - loss: 556.3130 - val_loss: 553.47
97
Epoch 48/500
40/40 [==============================] - 1s 17ms/step - loss: 558.8313 - val_loss: 546.82
82
Epoch 49/500
40/40 [==============================] - 1s 17ms/step - loss: 564.6194 - val_loss: 550.69
11
Epoch 50/500
40/40 [==============================] - 1s 16ms/step - loss: 556.6409 - val_loss: 563.39
04
Epoch 51/500
40/40 [==============================] - 1s 17ms/step - loss: 563.7345 - val_loss: 554.75
03
Epoch 52/500
40/40 [==============================] - 1s 17ms/step - loss: 552.9907 - val_loss: 542.85
40
Epoch 53/500
40/40 [==============================] - 1s 17ms/step - loss: 548.5219 - val_loss: 539.84
58
Epoch 54/500
40/40 [==============================] - 1s 17ms/step - loss: 547.3892 - val_loss: 542.06
12
Epoch 55/500
40/40 [==============================] - 1s 18ms/step - loss: 556.7477 - val_loss: 530.73
85
Epoch 56/500
40/40 [==============================] - 1s 18ms/step - loss: 548.0936 - val_loss: 536.69
95
Epoch 57/500
40/40 [==============================] - 1s 17ms/step - loss: 545.6442 - val_loss: 536.99
48
Epoch 58/500
40/40 [==============================] - 1s 18ms/step - loss: 546.4644 - val_loss: 530.50
```

```
02
Epoch 59/500
40/40 [==============================] - 1s 18ms/step - loss: 546.1219 - val_loss: 533.71
76
Epoch 60/500
40/40 [==============================] - 1s 18ms/step - loss: 546.7543 - val_loss: 534.23
80
Epoch 61/500
40/40 [==============================] - 1s 17ms/step - loss: 542.3483 - val_loss: 549.72
74
Epoch 62/500
40/40 [==============================] - 1s 18ms/step - loss: 549.6976 - val_loss: 525.31
04
Epoch 63/500
40/40 [==============================] - 1s 16ms/step - loss: 539.2811 - val_loss: 535.53
42
Epoch 64/500
40/40 [==============================] - 1s 16ms/step - loss: 542.2246 - val_loss: 537.00
42
Epoch 65/500
40/40 [==============================] - 1s 17ms/step - loss: 547.6266 - val_loss: 534.89
37
Epoch 66/500
40/40 [==============================] - 1s 17ms/step - loss: 544.0818 - val_loss: 530.44
12
Epoch 67/500
40/40 [==============================] - 1s 18ms/step - loss: 536.3462 - val_loss: 526.30
21
Epoch 68/500
40/40 [==============================] - 1s 17ms/step - loss: 537.4473 - val_loss: 527.28
08
Epoch 69/500
40/40 [==============================] - 1s 17ms/step - loss: 535.9975 - val_loss: 523.15
54
Epoch 70/500
40/40 [==============================] - 1s 17ms/step - loss: 540.0471 - val_loss: 532.13
53
Epoch 71/500
40/40 [==============================] - 1s 17ms/step - loss: 544.5071 - val_loss: 548.47
58
Epoch 72/500
40/40 [==============================] - 1s 18ms/step - loss: 536.0612 - val_loss: 527.94
17
Epoch 73/500
40/40 [==============================] - 1s 18ms/step - loss: 533.4623 - val_loss: 531.89
09
Epoch 74/500
40/40 [==============================] - 1s 17ms/step - loss: 538.6234 - val_loss: 539.83
40
Epoch 75/500
40/40 [==============================] - 1s 17ms/step - loss: 533.9480 - val_loss: 521.58
89
Epoch 76/500
40/40 [==============================] - 1s 17ms/step - loss: 532.0060 - val_loss: 520.63
51
Epoch 77/500
40/40 [==============================] - 1s 18ms/step - loss: 536.0676 - val_loss: 519.93
28
Epoch 78/500
40/40 [==============================] - 1s 17ms/step - loss: 536.0093 - val_loss: 523.39
34
Epoch 79/500
40/40 [==============================] - 1s 17ms/step - loss: 532.2665 - val_loss: 524.41
33
Epoch 80/500
40/40 [==============================] - 1s 17ms/step - loss: 535.0480 - val_loss: 518.58
62
Epoch 81/500
40/40 [==============================] - 1s 17ms/step - loss: 538.4529 - val_loss: 531.90
48
Epoch 82/500
40/40 [==============================] - 1s 17ms/step - loss: 536.7351 - val_loss: 531.55
```

```
98
Epoch 83/500
40/40 [==============================] - 1s 18ms/step - loss: 536.4313 - val_loss: 517.23
19
Epoch 84/500
40/40 [==============================] - 1s 19ms/step - loss: 532.3122 - val_loss: 524.95
81
Epoch 85/500
40/40 [==============================] - 1s 18ms/step - loss: 534.9319 - val_loss: 529.20
64
Epoch 86/500
40/40 [==============================] - 1s 17ms/step - loss: 526.1156 - val_loss: 518.82
33
Epoch 87/500
40/40 [==============================] - 1s 18ms/step - loss: 531.9188 - val_loss: 520.08
58
Epoch 88/500
40/40 [==============================] - 1s 17ms/step - loss: 528.1031 - val_loss: 545.02
44
Epoch 89/500
40/40 [==============================] - 1s 18ms/step - loss: 535.9907 - val_loss: 517.82
40
Epoch 90/500
40/40 [==============================] - 1s 17ms/step - loss: 531.4434 - val_loss: 521.28
04
Epoch 91/500
40/40 [==============================] - 1s 18ms/step - loss: 528.4277 - val_loss: 530.93
73
Epoch 92/500
40/40 [==============================] - 1s 18ms/step - loss: 529.2487 - val_loss: 518.33
12
Epoch 93/500
40/40 [==============================] - 1s 18ms/step - loss: 528.7942 - val_loss: 515.46
90
Epoch 94/500
40/40 [==============================] - 1s 18ms/step - loss: 527.6983 - val_loss: 517.78
38
Epoch 95/500
40/40 [==============================] - 1s 19ms/step - loss: 529.1889 - val_loss: 543.35
08
Epoch 96/500
40/40 [==============================] - 1s 17ms/step - loss: 532.2197 - val_loss: 522.92
86
Epoch 97/500
40/40 [==============================] - 1s 17ms/step - loss: 527.0184 - val_loss: 512.52
07
Epoch 98/500
40/40 [==============================] - 1s 17ms/step - loss: 534.2608 - val_loss: 527.02
61
Epoch 99/500
40/40 [==============================] - 1s 18ms/step - loss: 530.5549 - val_loss: 509.19
50
Epoch 100/500
40/40 [==============================] - 1s 18ms/step - loss: 532.2857 - val_loss: 519.82
78
Epoch 101/500
40/40 [==============================] - 1s 17ms/step - loss: 526.7936 - val_loss: 509.58
63
Epoch 102/500
40/40 [==============================] - 1s 18ms/step - loss: 526.2722 - val_loss: 517.86
74
Epoch 103/500
40/40 [==============================] - 1s 17ms/step - loss: 529.6471 - val_loss: 517.25
43
Epoch 104/500
40/40 [==============================] - 1s 18ms/step - loss: 525.2417 - val_loss: 517.70
78
Epoch 105/500
40/40 [==============================] - 1s 18ms/step - loss: 521.9393 - val_loss: 516.23
22
Epoch 106/500
40/40 [==============================] - 1s 17ms/step - loss: 530.2859 - val_loss: 512.48
```

```
61
Epoch 107/500
40/40 [==============================] - 1s 18ms/step - loss: 523.5263 - val_loss: 516.06
86
Epoch 108/500
40/40 [==============================] - 1s 17ms/step - loss: 525.7141 - val_loss: 510.90
76
Epoch 109/500
40/40 [==============================] - 1s 16ms/step - loss: 523.4239 - val_loss: 518.41
66
```

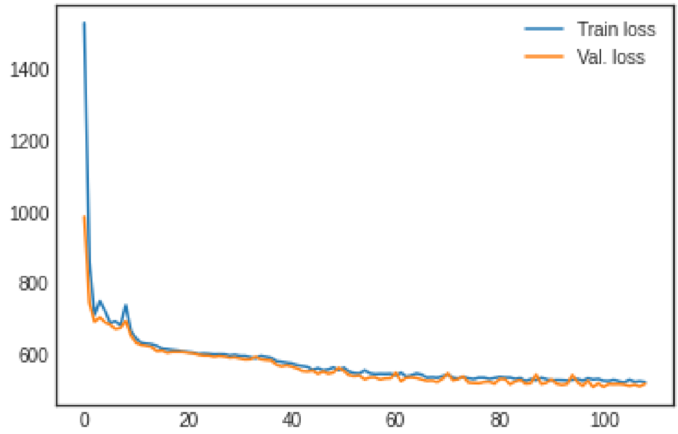
Out[176]:

```
<keras.callbacks.History at 0x7f4a95e4ec10>
```

Plot the learning curves for loss vs epoch for both training and validation sets.

In [177]:

```
history = vae.history.history
plt.plot(history['loss'], label='Train loss')
plt.plot(history['val_loss'], label='Val. loss')
plt.legend()
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.**

Randomly sample 1000 images from the dataset, and pass them through the encoder.

In [201]:

```
sample = dataset[np.random.choice(dataset.shape[0], 1000)] / 255.
```

In [202]:

```
# Sanity check
sample.shape
```

Out[202]:

```
(1000, 36, 36, 3)
```

Display the embeddings in a scatter plot (project to 2 dimensions if the latent space has dimension higher than

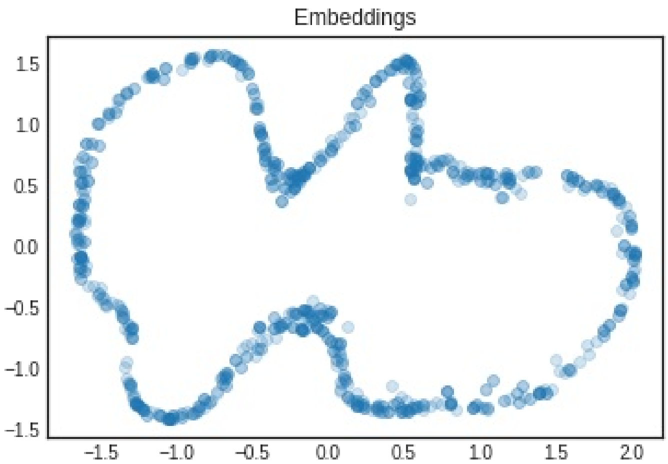
Display the embeddings in a scatter plot (project to 2 dimensions if the latent space has dimension higher than two).

In [205]:

```
# Get the embeddings
embeddings = encoder(sample)

# Instantiate PCA
pca = PCA(n_components=2)

# Reduce the dimension
emb_2d = pca.fit_transform(embeddings.mean().numpy())
```

In [206]:

```
# Plot
plt.scatter(emb_2d[:, 0], emb_2d[:, 1], alpha=.2)
plt.title('Embeddings')
plt.show()
```



Randomly sample 4 images from the dataset and for each image, display the original and reconstructed image from the VAE in a figure.

In [211]:

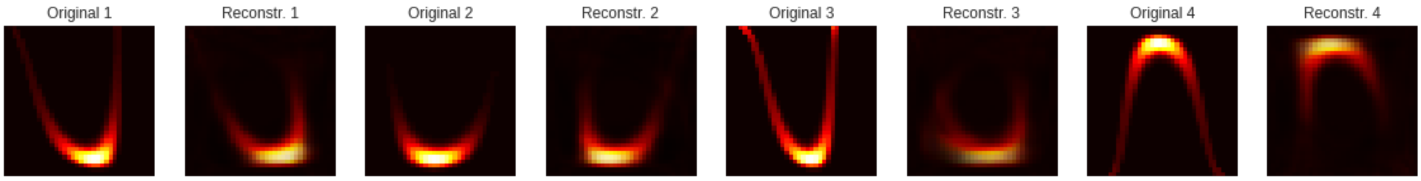
```
# Sample
sample_2 = dataset[np.random.choice(dataset.shape[0], 4)] / 255.
```

In [214]:

```
# Reconstruction
reconstruction = vae(sample_2).mean()
```

In [227]:

```
# Plot
plt.figure(figsize=(20, 7))

for i, (s, r) in enumerate(zip(sample_2, reconstruction)):

    # Plot original
    plt.subplot(1, 8, 2*i + 1)
    plt.imshow(s)
    plt.title(f'Original {i + 1}')
    plt.axis('off')

    # Plot reconstruction
    plt.subplot(1, 8, 2*i + 2)
    plt.imshow(r)
    plt.title(f'Reconstr. {i + 1}')
    plt.axis('off')

plt.show()
```

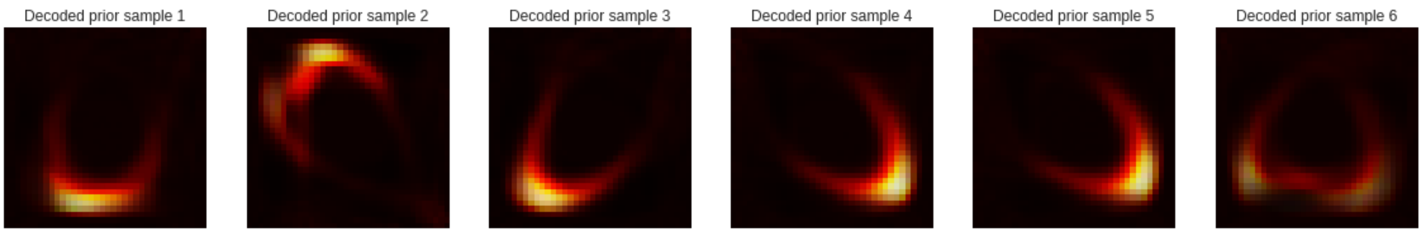


Randomly sample 6 latent variable realisations from the prior distribution, and display the images in a figure.

- **Again use the mean of the output distribution to display the images.**

In [244]:

```
# Get samples
prior_samples = prior.sample(6)

prior_images = decoder(prior_samples).mean()

# Plot
plt.figure(figsize=(20, 7))

for i, img in enumerate(prior_images):

  # Plot
  plt.subplot(1, 6, i + 1)
  plt.imshow(img)
  plt.title(f'Decoded prior sample {i + 1}')
  plt.axis('off')

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 [228]:

```
# 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([-3, 3])
    ax1.set_ylim([-3, 3])
    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
```

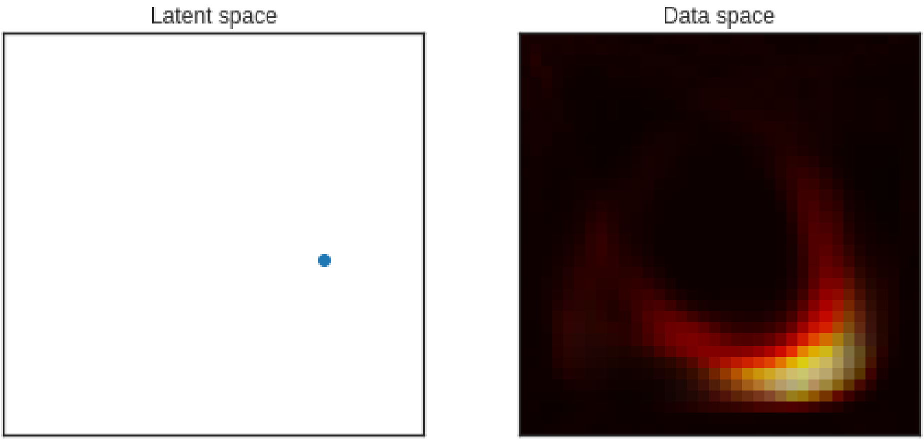
```
    line, = ax1.plot([], [], marker='o')
    img2 = ax2.imshow(np.zeros((36, 36, 3)))

    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.f
loat32))

    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 [229]:

```
# Create the animation

a = get_animation(latent_size, decoder, interpolation_length=200)
HTML(a.to_html5_video())
```

Out[229]:

Your browser does not support the video tag.



In [ ]: