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 [165]:
```
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

import math as m
from sklearn.model_selection import train_test_split

import pickle
```

In [166]:
```
from tensorflow.keras.layers import Conv2D, BatchNormalization, Dense, Flatten, Reshape, UpSampling2D
from tensorflow.keras.models import Model
```

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:

- $f_1(z) = (z_1, z_2 - 2)$,
- $f_2(z) = (z_1, \frac{z_2}{2})$,
- $f_3(z) = (z_1, z_2 + a z_1^2)$,
- $f_4(z) = Rz$, 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 $[-1, 1]$.

ADDITIONAL REFERENCE: https://tiao.io/post/building-probability-distributions-with-tensorflow-probability-bijector-api/

In [167]:
```
```

In [168]:
```
1
```

In [169]:
```
1
```

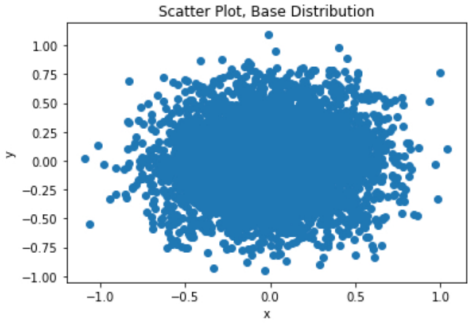
In [170]:
```
1
```

Out[170]:
```
<tfp.distributions.TransformedDistribution 'chain_of_shift_of_scaleMultivariateNormalDiag' batch_shape=[] event_shape=[2] dtype
=float32>
```

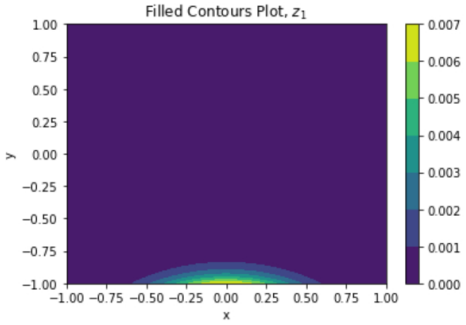
In [171]:
```
1
```

Scatter plot of the base distribution

In [172]:
```
1
```



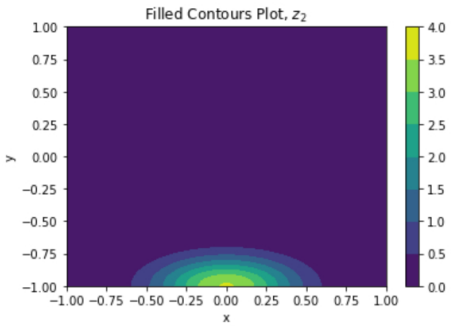
In [173]:
```
1
```



In [174]:
```
1
```

Out[174]:
```
<tfp.distributions.TransformedDistribution 'chain_of_shift_of_scaleMultivariateNormalDiag' batch_shape=[] event_shape=[2] dtype
=float32>
```

In [175]:
```
1
```



$$f_3(z) = (z_1, z_2 + az_1^2),$$

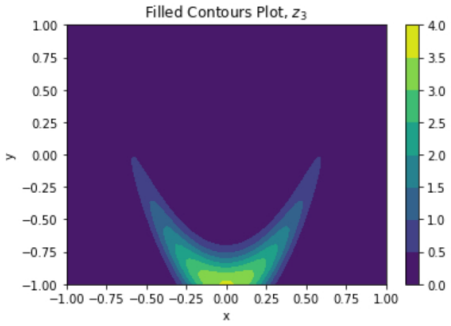
In [176]:
```
1
```

In [177]:
```
1
```

In [178]:
```
1
```

Out[178]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=2.615054>
```

In [179]:
```
1
```

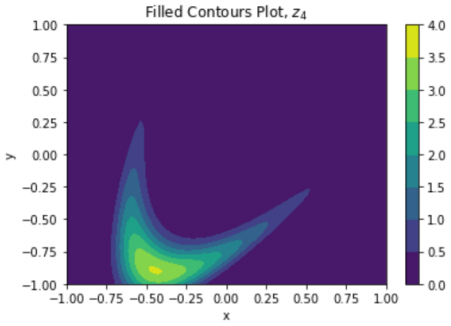


In [180]:
```
1
```

In [181]:
```
1
```

In [182]:
```
1
```

In [183]:
```
1
```

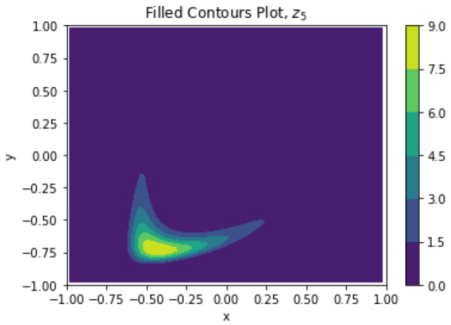


In [184]:
```
1
```

In [185]:
```
1
```

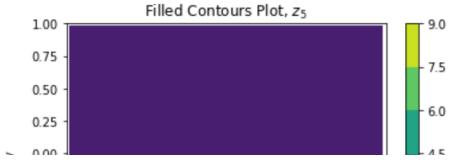
Out[185]:
```
<tfp.distributions.TransformedDistribution 'chain_of_shift_of_scaleMultivariateNormalDiag' batch_shape=[] event_shape=[2] dtype
=float32>
```

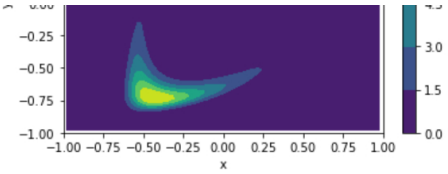
In [186]:
```
1
```



In [187]:
```
1
```

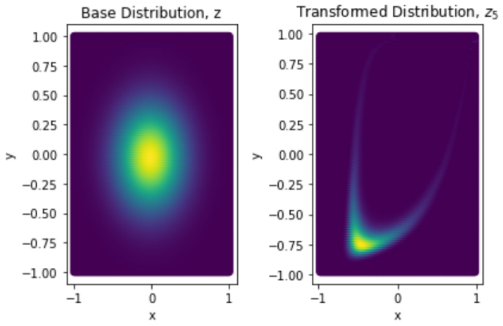
In [188]:
```
1
```





In [189]:
```
1
```

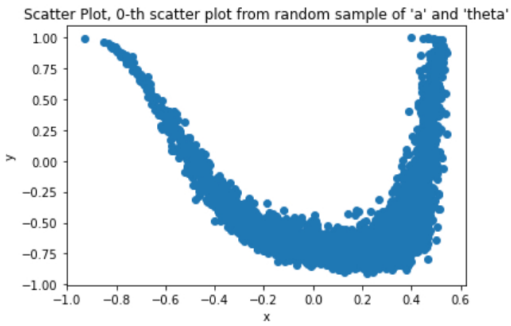
```
<Figure size 864x648 with 0 Axes>
```

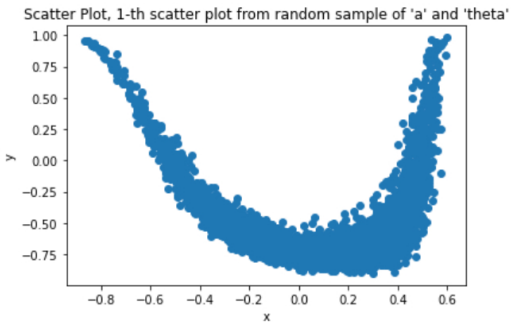


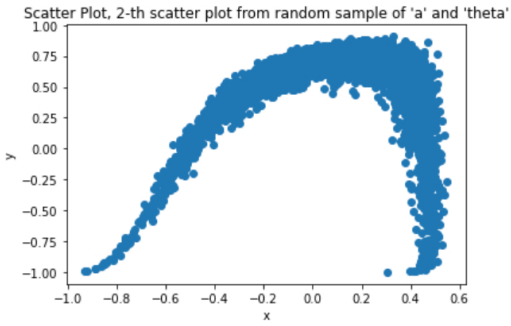
In [190]:
```
1
```

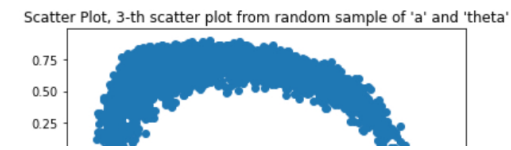
Plot the four scatter plots with random 'a' and 'theta' samples.

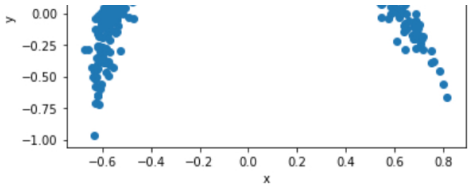
In [191]:
```
1
```











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.

TUTORIAL on VAE: https://tiao.io/post/tutorial-on-variational-autoencoders-with-a-concise-keras-implementation/

In [192]:
```
1
```

In [193]:
```
1
```

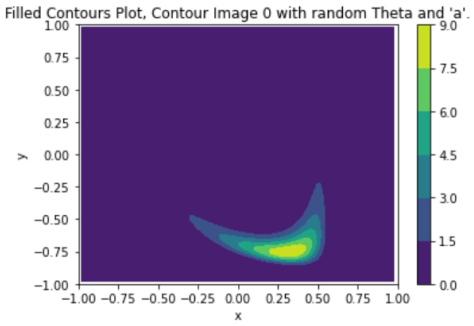
In [194]:
```
1
```

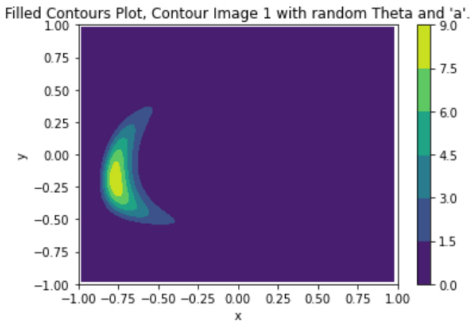
RUN THIS CELL TO LOAD THE `DATASET_IMAGES` IF the dataset has already been created a priori, this is to save time.

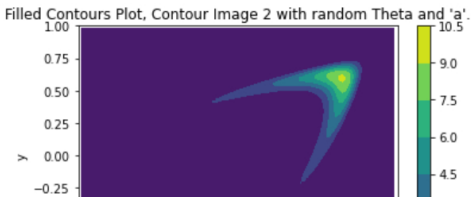
In [195]:
```
1
```

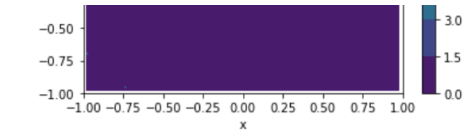
Codes to make the 4 scatter/contour plots

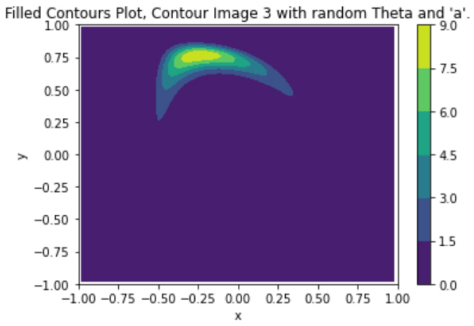
In [196]:
```
1
```











Codes to make the dataset

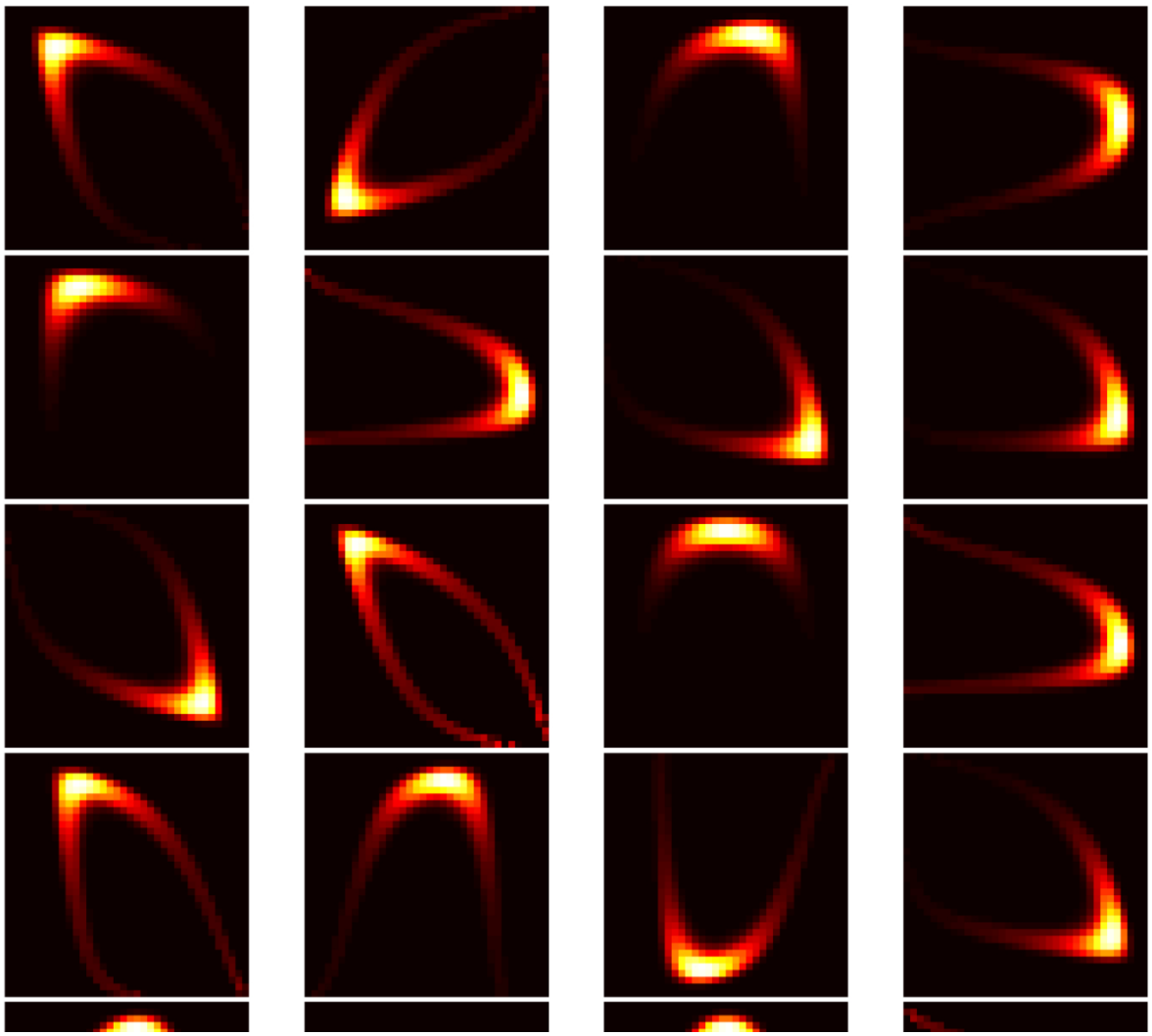
In [197]:
```
1
```

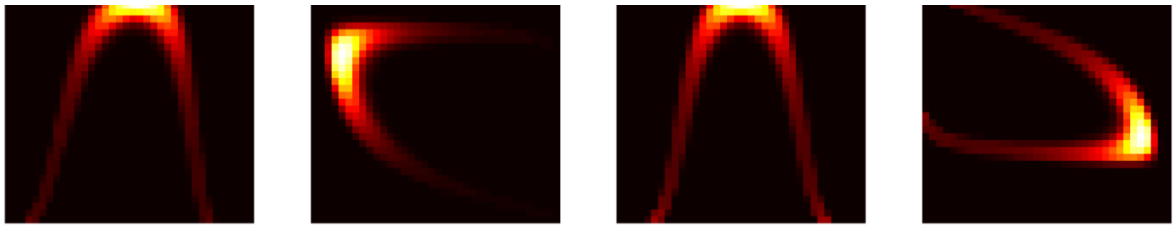
In [198]:
```
1
```

In [199]:
```
1
```

Plot the 20 images

In [200]:
```
1
```





---

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 [201]:
```
1
```

In [202]:
```
1
```

Out[202]:
```
(800, 36, 36, 3)
```

In [203]:
```
1
```

In [204]:
```
1
```

```
tf.Tensor(
[[[13.  0.  0.]
  [13.  0.  0.]
  [13.  0.  0.]
  ...
  [13.  0.  0.]
  [13.  0.  0.]
  [13.  0.  0.]]

 [[13.  0.  0.]
  [13.  0.  0.]
  [13.  0.  0.]
  ...
  [13.  0.  0.]
  [13.  0.  0.]
  [13.  0.  0.]]

 [[13.  0.  0.]
  [13.  0.  0.]
  [13.  0.  0.]
  ...
  [13.  0.  0.]
  [13.  0.  0.]
  [13.  0.  0.]]

 ...

 [[13.  0.  0.]
  [13.  0.  0.]
  [13.  0.  0.]
  ...
  [29.  0.  0.]
  [13.  0.  0.]
  [13.  0.  0.]]

 [[13.  0.  0.]
  [13.  0.  0.]
  [13.  0.  0.]
  ...
  [37.  0.  0.]
  [21.  0.  0.]
  [13.  0.  0.]]

 [[13.  0.  0.]
  [13.  0.  0.]
  [13.  0.  0.]
  ...
  [29.  0.  0.]
  [45.  0.  0.]
  [13.  0.  0.]]], shape=(36, 36, 3), dtype=float64)
```

In [205]:
```
1
```

In [206]:
```
1
```

In [207]:
```
1
```

In [208]:
```
1
```

Out[208]:
```
(TensorSpec(shape=(20, 36, 36, 3), dtype=tf.float64, name=None),
 TensorSpec(shape=(20, 36, 36, 3), dtype=tf.float64, name=None))
```

In [209]:
```
1
```

Out[209]:
```
(TensorSpec(shape=(20, 36, 36, 3), dtype=tf.float64, name=None),
 TensorSpec(shape=(20, 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 [210]:
```
1
```

In [211]:
```
1
```

In [212]:
```
1
```

In [213]:
```
1
```

In [214]:
```
1
```

In [215]:
```
1
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_16 (Conv2D)           (None, 18, 18, 256)       12544
_________________________________________________________________
batch_normalization_8 (Batch (None, 18, 18, 256)       1024
_________________________________________________________________
conv2d_17 (Conv2D)           (None, 9, 9, 128)         524416
_________________________________________________________________
batch_normalization_9 (Batch (None, 9, 9, 128)         512
_________________________________________________________________
conv2d_18 (Conv2D)           (None, 5, 5, 64)          131136
_________________________________________________________________
batch_normalization_10 (Batc (None, 5, 5, 64)          256
_________________________________________________________________
conv2d_19 (Conv2D)           (None, 3, 3, 32)          32800
_________________________________________________________________
batch_normalization_11 (Batc (None, 3, 3, 32)          128
_________________________________________________________________
flatten_4 (Flatten)          (None, 288)               0
_________________________________________________________________
dense_4 (Dense)              (None, 350)               101150
_________________________________________________________________
multivariate_normal_tri_l_2  multiple                  0
=================================================================
Total params: 803,966
Trainable params: 803,006
Non-trainable params: 960
_________________________________________________________________
```

In [216]:
```
1
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_5 (Dense)              (None, 288)               7488
_________________________________________________________________
reshape_2 (Reshape)          (None, 3, 3, 32)          0
_________________________________________________________________
up_sampling2d_6 (UpSampling2 (None, 6, 6, 32)          0
_________________________________________________________________
conv2d_20 (Conv2D)           (None, 6, 6, 64)          18496
_________________________________________________________________
up_sampling2d_7 (UpSampling2 (None, 12, 12, 64)        0
_________________________________________________________________
conv2d_21 (Conv2D)           (None, 12, 12, 128)       73856
_________________________________________________________________
up_sampling2d_8 (UpSampling2 (None, 36, 36, 128)       0
_________________________________________________________________
conv2d_22 (Conv2D)           (None, 36, 36, 256)       295168
_________________________________________________________________
conv2d_23 (Conv2D)           (None, 36, 36, 3)         6915
_________________________________________________________________
flatten_5 (Flatten)          (None, 3888)              0
_________________________________________________________________
independent_bernoulli_2 (Ind multiple                  0
=================================================================
Total params: 401,923
Trainable params: 401,923
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.

In [217]:
```
1
```

```
Model: "model_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_16_input (InputLayer) [(None, 36, 36, 3)]       0
_________________________________________________________________
conv2d_16 (Conv2D)           (None, 18, 18, 256)       12544
_________________________________________________________________
batch_normalization_8 (Batch (None, 18, 18, 256)       1024
_________________________________________________________________
conv2d_17 (Conv2D)           (None, 9, 9, 128)         524416
_________________________________________________________________
batch_normalization_9 (Batch (None, 9, 9, 128)         512
_________________________________________________________________
conv2d_18 (Conv2D)           (None, 5, 5, 64)          131136
_________________________________________________________________
batch_normalization_10 (Batc (None, 5, 5, 64)          256
_________________________________________________________________
conv2d_19 (Conv2D)           (None, 3, 3, 32)          32800
_________________________________________________________________
batch_normalization_11 (Batc (None, 3, 3, 32)          128
_________________________________________________________________
flatten_4 (Flatten)          (None, 288)               0
_________________________________________________________________
dense_4 (Dense)              (None, 350)               101150
_________________________________________________________________
multivariate_normal_tri_l_2  multiple                  0
_________________________________________________________________
sequential_5 (Sequential)    multiple                  401923
=================================================================
Total params: 1,205,889
Trainable params: 1,204,929
Non-trainable params: 960
_________________________________________________________________
```

In [218]:
```
1
```

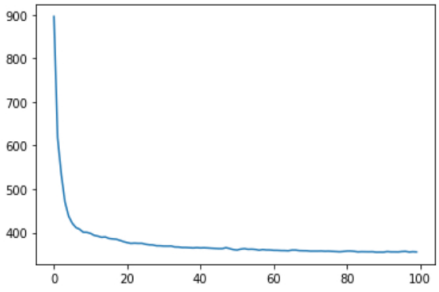
In [219]:
```
1
```

In [220]:
```
1
```

```
Epoch 1/100
40/40 [==============================] - 3s 33ms/step - loss: 1213.3822 - val_loss: 1323.9200
Epoch 2/100
40/40 [==============================] - 1s 14ms/step - loss: 644.4906 - val_loss: 946.5737
Epoch 3/100
40/40 [==============================] - 1s 14ms/step - loss: 552.1047 - val_loss: 776.0366
Epoch 4/100
40/40 [==============================] - 1s 14ms/step - loss: 483.6037 - val_loss: 764.7193
Epoch 5/100
40/40 [==============================] - 1s 14ms/step - loss: 440.5036 - val_loss: 707.3477
Epoch 6/100
40/40 [==============================] - 1s 13ms/step - loss: 426.4992 - val_loss: 691.2971
Epoch 7/100
40/40 [==============================] - 1s 14ms/step - loss: 412.9086 - val_loss: 656.0770
Epoch 8/100
40/40 [==============================] - 1s 14ms/step - loss: 410.7319 - val_loss: 624.9415
Epoch 9/100
40/40 [==============================] - 1s 14ms/step - loss: 402.8661 - val_loss: 597.3528
Epoch 10/100
40/40 [==============================] - 1s 13ms/step - loss: 401.0709 - val_loss: 545.9858
```

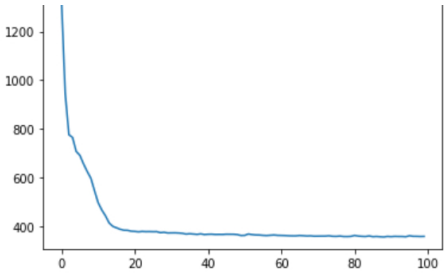
In [221]:
```
1
```

Out[221]:
```
[<matplotlib.lines.Line2D at 0x7fec7b5dbad0>]
```



In [222]:
```
1
```

Out[222]:
```
[<matplotlib.lines.Line2D at 0x7fec7c641410>]
```



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 [223]:
```
1
```

In [224]:
```
1
```

Out[224]:
```
(1000, 36, 36, 3)
```

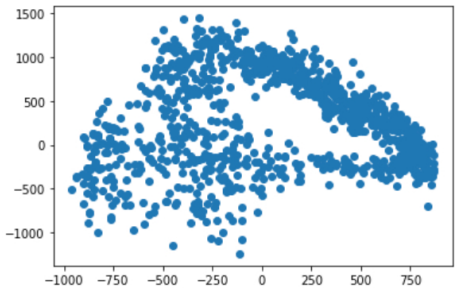
In [225]:
```
1
```

In [226]:
```
1
```

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

In [239]:
```
1
```

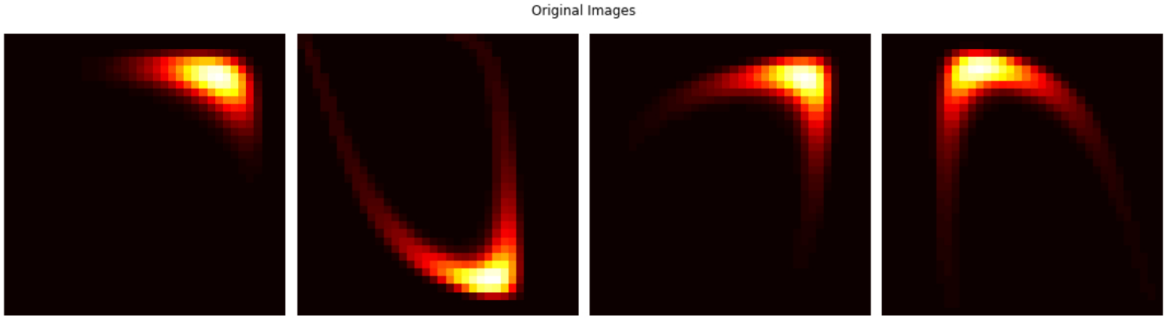
Out[239]:
```
<matplotlib.collections.PathCollection at 0x7fec7aec4b10>
```



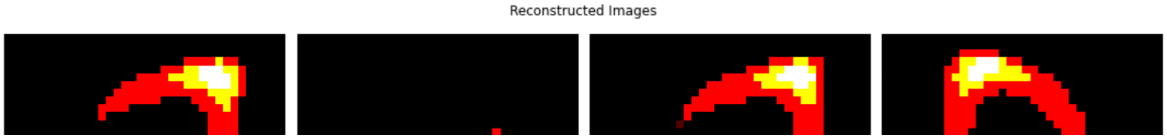
Four Original and its Reconstructed Images

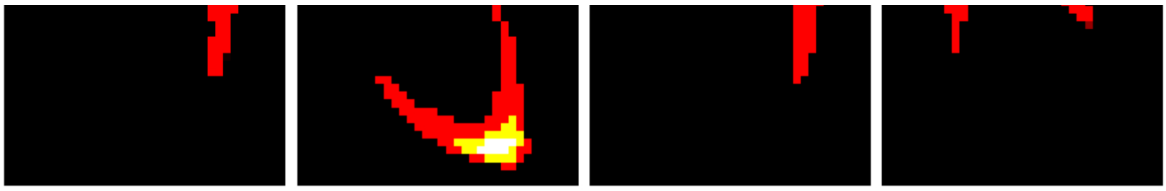
In [227]:
```
1
```

In [243]:
```
1
```



In [247]:
```
1
```

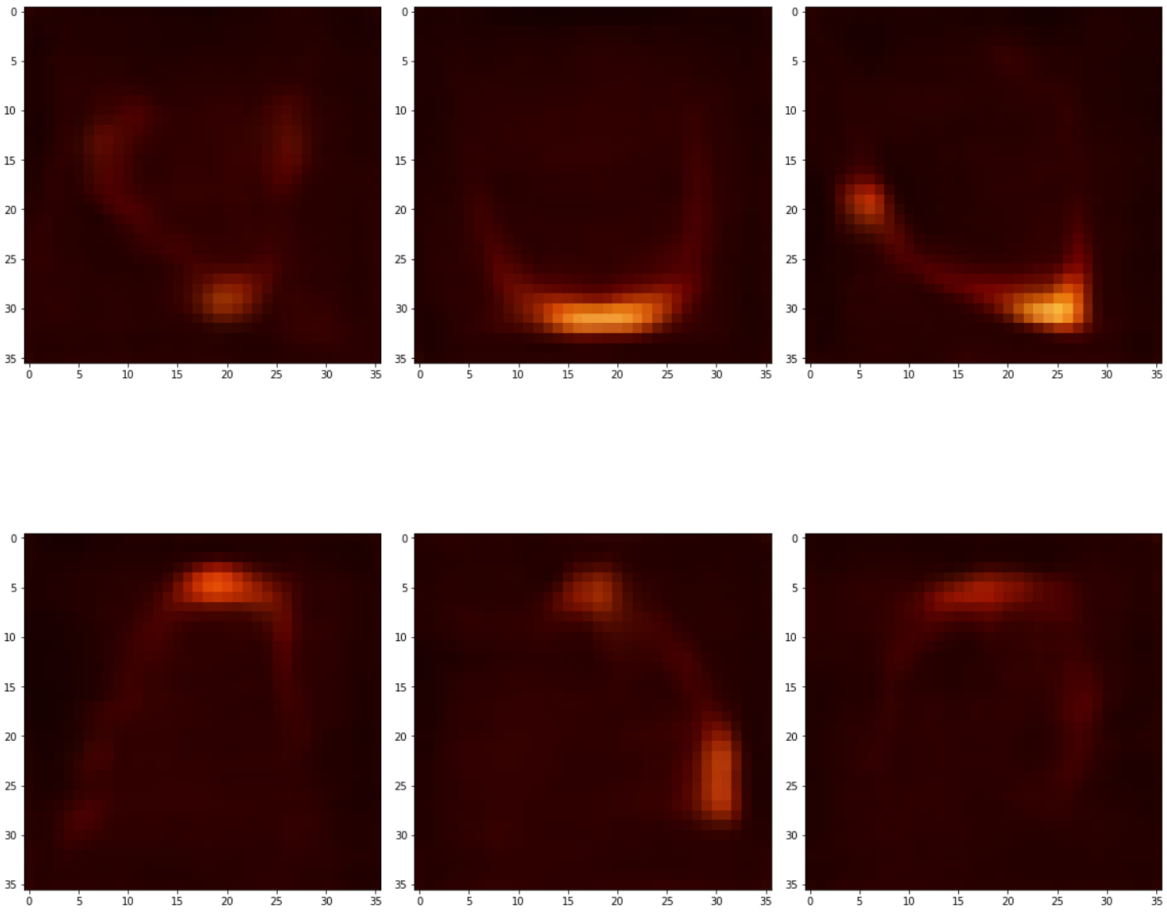




Randomly sample 6 latent variable realisations from the prior distribution, and display of the images.

In [231]:
```
1
```

In [232]:
```
1
```



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.

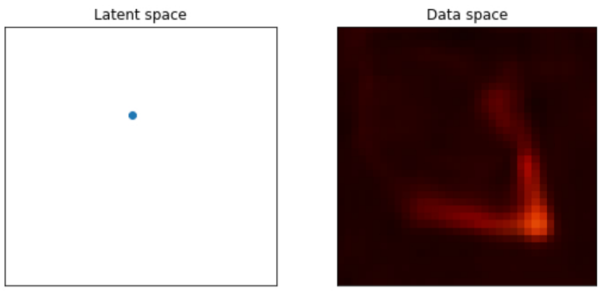
In [248]:
```
# Function to create animation

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)
```

```
    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.float32))

    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 [249]:

```
# Create the animation
latent_size = 25
a = get_animation(latent_size, decoder, interpolation_length=200)
HTML(a.to_html5_video())
```

Out[249]:



In [ ]:

```
```