Capstone Project

February 13, 2022

1 Capstone Project

1.1 Probabilistic generative models

1.1.1 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.

1.1.2 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.

1.1.3 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.

[ ]:
```
from google.colab import drive
drive.mount('/content/drive',force_remount=True)
```

```
Mounted at /content/drive
```

[ ]:
```
import os
os.chdir('/content/drive/MyDrive/Colab_Notebooks/coursera/
 ↪coursera_Probabilistic_Deep_Learning_TF2/Week5')
```

[ ]:
```
!pip install tensorflow=='2.1.0'
```

[ ]:
```
!pip install tensorflow_probability=='0.9.0'
```

[ ]:
```
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:
- $\theta \sim U[0, 2\pi)$
- $a \sim N(3, 1)$

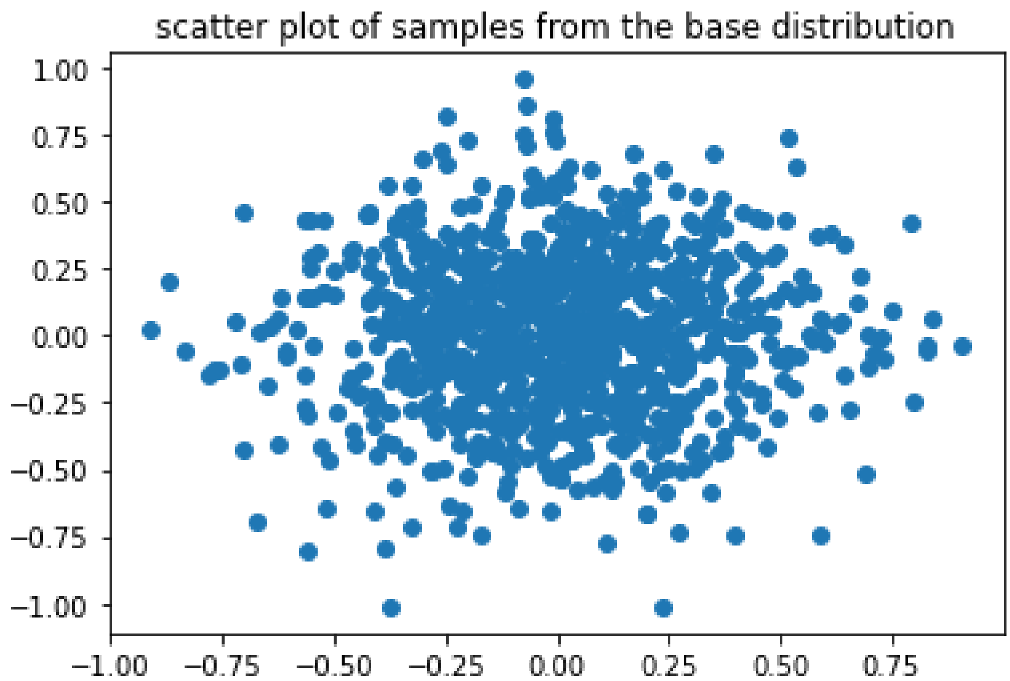
The complete normalising flow is given by the following chain of transformations: * $f_1(z) = (z_1, z_2 - 2)$, * $f_2(z) = (z_1, \frac{z_2}{2})$, * $f_3(z) = (z_1, z_2 + az_1^2)$, * $f_4(z) = Rz$, where R is a rotation matrix with angle θ, * $f_5(z) = \tanh(z)$, where the tanh function is applied elementwise.

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 $[-1, 1]$.

[ ]:
```
# Define base distribution
base_distribution = tfd.MultivariateNormalDiag(loc=[0,0],scale_diag=[0.3,0.3])
theta_distribution=tfd.Uniform(low=0,high=2*np.pi)
alpha_distribution=tfd.Normal(loc=3,scale=1)
```

[ ]:
```
# scatter of base distribution
n_samples=1000
samples=base_distribution.sample(n_samples)
#plt.scatter(np.arange(n_samples),samples.numpy().squeeze()[:,1])
plt.scatter(samples.numpy().squeeze()[:,0],samples.numpy().squeeze()[:,1])
plt.title('scatter plot of samples from the base distribution')
```

```
plt.show()
```



[ ]:
```
class ScaleSquare(tfb.Bijector):
  def __init__(self,a,name='ScaleSquare',**kwargs):
    super(ScaleSquare,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,z):
    z=tf.cast(z,dtype=tf.float32)
    return tf.concat([z[...,0:1],
                      z[...,1:]+self.a*tf.square(z[...,0:1])],axis=-1)
  def _inverse(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 _forward_log_det_jacobian(self, z):
      return tf.constant(0.,dtype=z.dtype)
```

[ ]:
```
class Rotate(tfb.Bijector):
  def __init__(self,theta,name='Rotate',**kwargs):
    super(Rotate,self).__init__(forward_min_event_ndims=1,
```

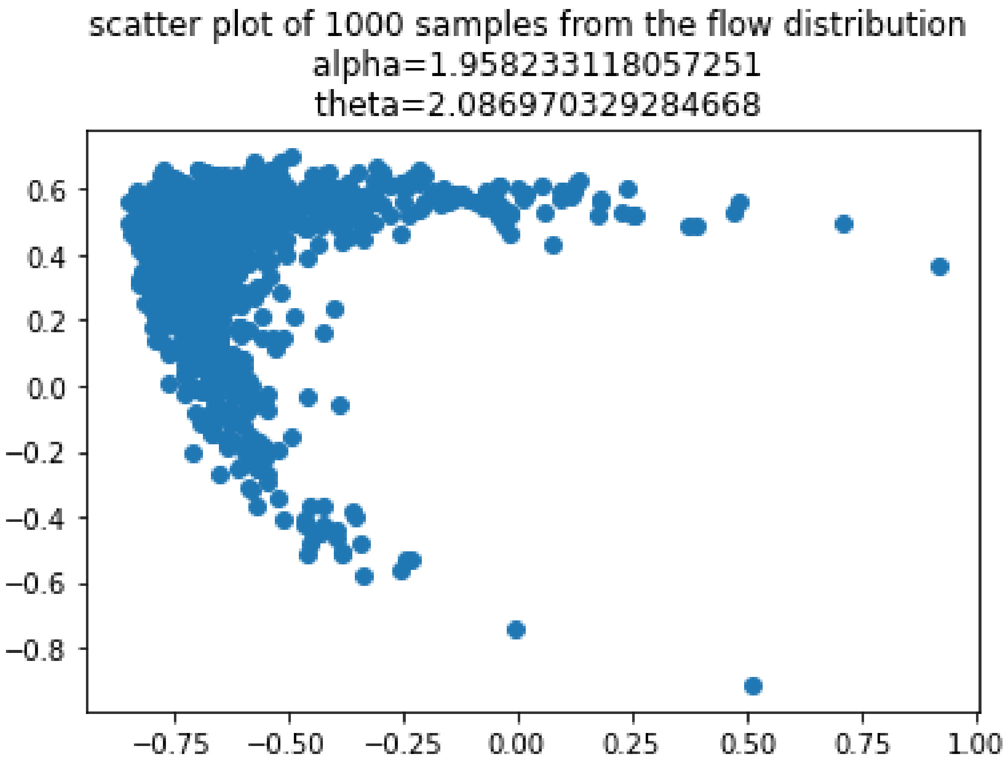
```
                                    name=name,
                                    is_constant_jacobian=True,
                                    validate_args=False,
                                    **kwargs)
    self.R=np.array([[np.cos(theta),-np.sin(theta)],
                    [np.sin(theta),np.cos(theta)]])
    self.R=tf.convert_to_tensor(self.R)
    self.R=tf.cast(self.R,tf.float32)

  def _forward(self,z):
    z=tf.cast(z,dtype=tf.float32)
    return tf.cast(np.dot(z.numpy(),self.R.numpy()),tf.float32)
  def _inverse(self,x):
    x=tf.cast(x,dtype=tf.float32)
    return tf.cast(np.dot(x.numpy(),self.R.numpy().T),tf.float32)
  def _forward_log_det_jacobian(self, z):
      return tf.constant(0.,dtype=z.dtype)
```

[ ]:

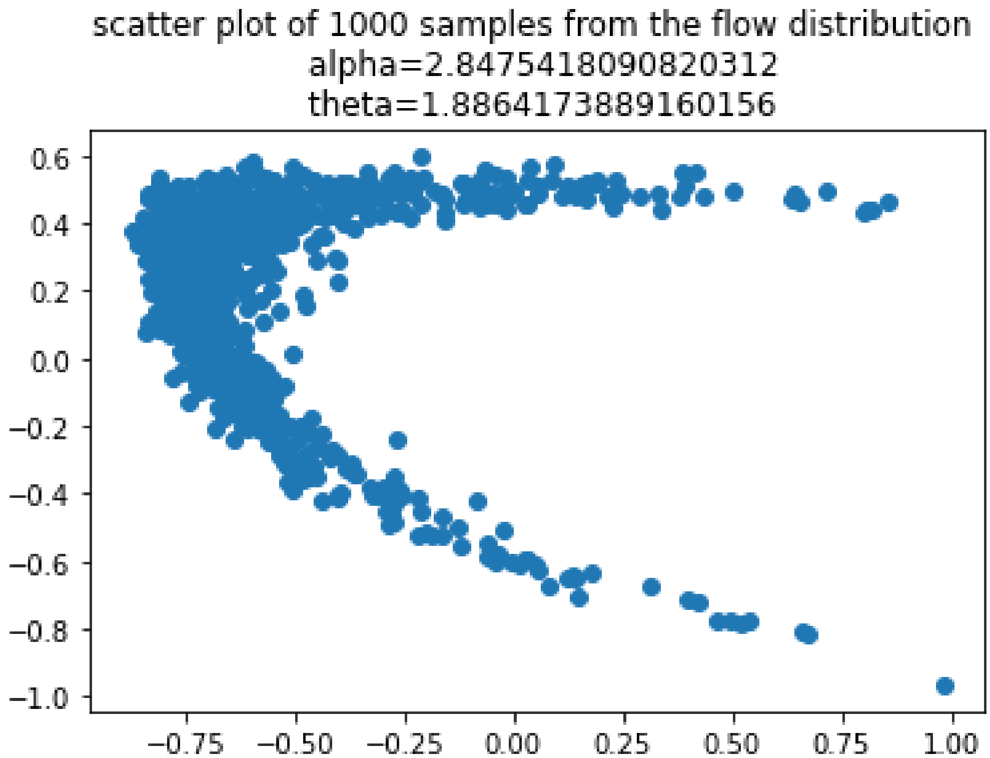
[ ]:
```
def chain_flow(a,theta):
  f1=tfb.Shift([0,-2])
  f2=tfb.Scale([1,0.5])
  f3=ScaleSquare(a)
  f4=Rotate(theta)
  f5=tfb.Tanh()
  chained=tfb.Chain([f5,f4,f3,f2,f1])
  return chained
```

[ ]:
```
def get_flow_dist(a,theta,base_distribution):
  dist=tfd.TransformedDistribution(distribution=base_distribution,
                                    bijector=chain_flow(a,theta))
  return dist
```

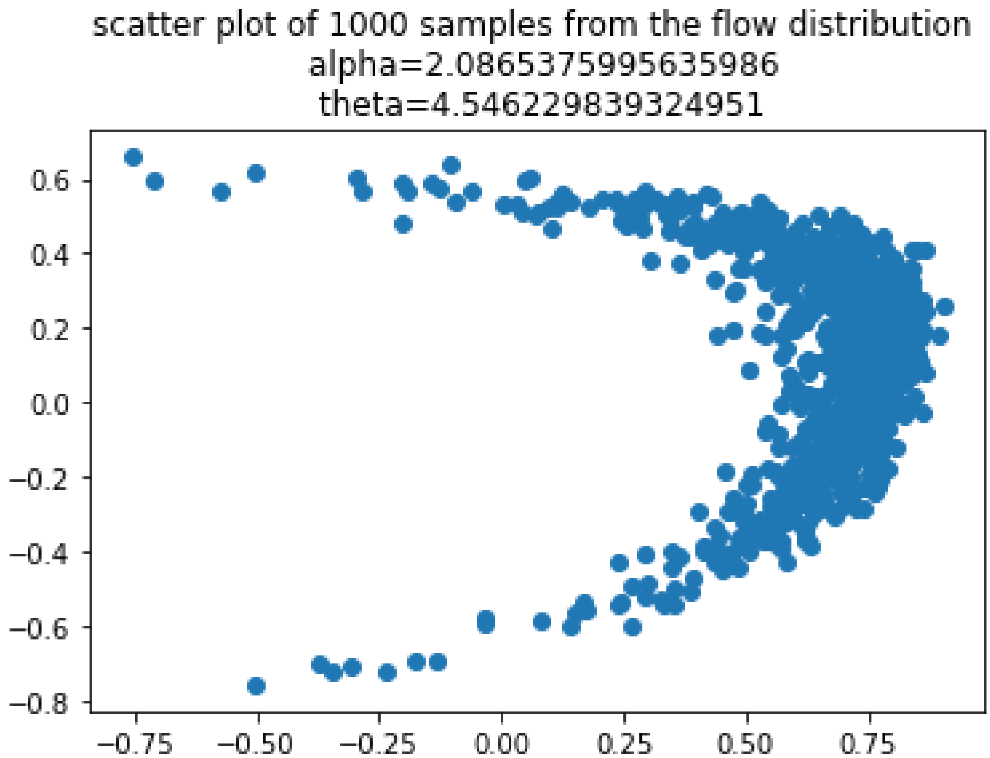
[ ]:
```
#plot 1
n_samples=1000
theta=theta_distribution.sample(1).numpy()[0]
a=alpha_distribution.sample(1).numpy()[0]
samples=base_distribution.sample(n_samples)
flow_distribution=get_flow_dist(a,theta,base_distribution)
samples=flow_distribution.sample(n_samples)

#plt.scatter(np.arange(n_samples),samples.numpy().squeeze()[:,1])
plt.scatter(samples.numpy().squeeze()[:,0],samples.numpy().squeeze()[:,1])
plt.title('scatter plot of {} samples from the flow distribution \n alpha={}\n␣
 ↪theta={}'.format(n_samples,a,theta))
plt.show()
```



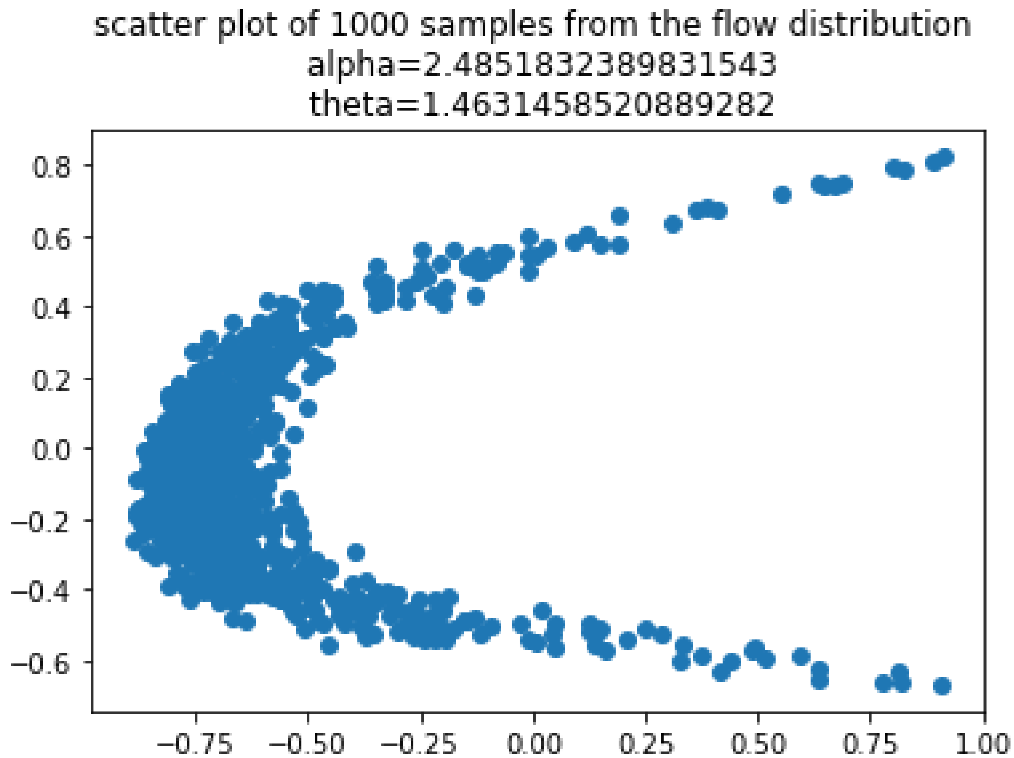
[ ]:
```
#plot 2
n_samples=1000
theta=theta_distribution.sample(1).numpy()[0]
a=alpha_distribution.sample(1).numpy()[0]
samples=base_distribution.sample(n_samples)
flow_distribution=get_flow_dist(a,theta,base_distribution)
samples=flow_distribution.sample(n_samples)

#plt.scatter(np.arange(n_samples),samples.numpy().squeeze()[:,1])
plt.scatter(samples.numpy().squeeze()[:,0],samples.numpy().squeeze()[:,1])
plt.title('scatter plot of {} samples from the flow distribution \n alpha={}\n␣
 ↪theta={}'.format(n_samples,a,theta))
plt.show()
```



[ ]:
```
#plot 3
n_samples=1000
theta=theta_distribution.sample(1).numpy()[0]
a=alpha_distribution.sample(1).numpy()[0]
samples=base_distribution.sample(n_samples)
flow_distribution=get_flow_dist(a,theta,base_distribution)
samples=flow_distribution.sample(n_samples)

#plt.scatter(np.arange(n_samples),samples.numpy().squeeze()[:,1])
plt.scatter(samples.numpy().squeeze()[:,0],samples.numpy().squeeze()[:,1])
plt.title('scatter plot of {} samples from the flow distribution \n alpha={}\n␣
 ↪theta={}'.format(n_samples,a,theta))
plt.show()
```



[ ]:
```
#plot 4
n_samples=1000
theta=theta_distribution.sample(1).numpy()[0]
a=alpha_distribution.sample(1).numpy()[0]
samples=base_distribution.sample(n_samples)
flow_distribution=get_flow_dist(a,theta,base_distribution)
samples=flow_distribution.sample(n_samples)

#plt.scatter(np.arange(n_samples),samples.numpy().squeeze()[:,1])
plt.scatter(samples.numpy().squeeze()[:,0],samples.numpy().squeeze()[:,1])
plt.title('scatter plot of {} samples from the flow distribution \n alpha={}\n␣
↪theta={}'.format(n_samples,a,theta))
plt.show()
```



1.2 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.

[ ]:
```
# Helper function to compute transformed distribution densities
```

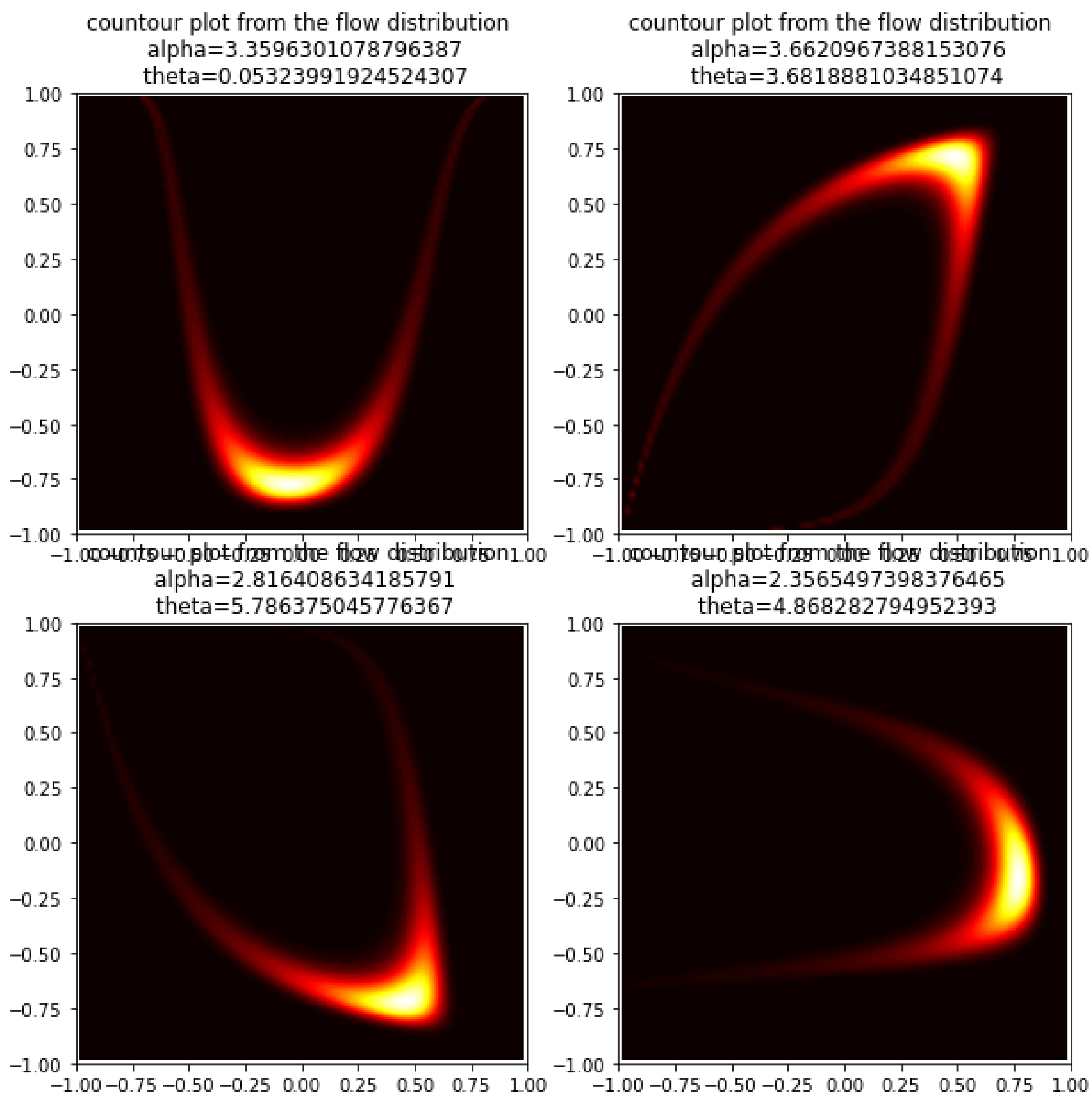
```python
X, Y = np.meshgrid(np.linspace(-1, 1, 100), np.linspace(-1, 1, 100))
inputs = np.transpose(np.stack((X, Y)), [1, 2, 0])

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
```

[ ]:
```python
# 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
```

[ ]:
```python
# plot 4 contours
plt.figure(figsize=(10,10))
for i in range(4):
```

```
    theta=theta_distribution.sample(1).numpy()[0]
    a=alpha_distribution.sample(1).numpy()[0]
    flow_distribution=get_flow_dist(a,theta,base_distribution)
    print(flow_distribution)
    flow_distribution=tfd.BatchReshape(flow_distribution,[1])
    print(flow_distribution)
    plt.subplot(2, 2, i+1)
    plt.contourf(X, Y, get_densities(flow_distribution).squeeze(), cmap='hot',␣
 ↪levels=100)
    plt.title('countour plot from the flow distribution \n alpha={}\n theta={}'.
 ↪format(a,theta))
plt.show()
```

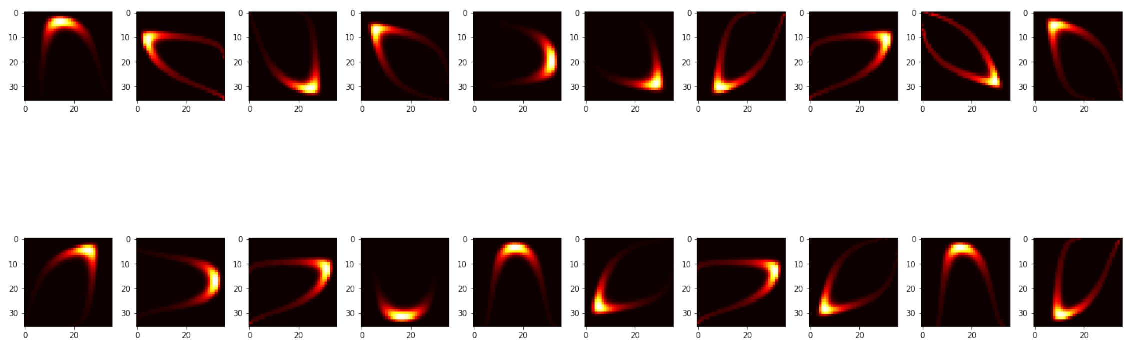
```
tfp.distributions.TransformedDistribution("chain_of_tanh_of_Rotate_of_ScaleSquar
e_of_scale_of_shiftMultivariateNormalDiag", batch_shape=[], event_shape=[2],
dtype=float32)
tfp.distributions.BatchReshape("BatchReshapechain_of_tanh_of_Rotate_of_ScaleSqua
re_of_scale_of_shiftMultivariateNormalDiag", batch_shape=[1], event_shape=[2],
dtype=float32)
tfp.distributions.TransformedDistribution("chain_of_tanh_of_Rotate_of_ScaleSquar
e_of_scale_of_shiftMultivariateNormalDiag", batch_shape=[], event_shape=[2],
dtype=float32)
tfp.distributions.BatchReshape("BatchReshapechain_of_tanh_of_Rotate_of_ScaleSqua
re_of_scale_of_shiftMultivariateNormalDiag", batch_shape=[1], event_shape=[2],
dtype=float32)
tfp.distributions.TransformedDistribution("chain_of_tanh_of_Rotate_of_ScaleSquar
e_of_scale_of_shiftMultivariateNormalDiag", batch_shape=[], event_shape=[2],
dtype=float32)
tfp.distributions.BatchReshape("BatchReshapechain_of_tanh_of_Rotate_of_ScaleSqua
re_of_scale_of_shiftMultivariateNormalDiag", batch_shape=[1], event_shape=[2],
dtype=float32)
tfp.distributions.TransformedDistribution("chain_of_tanh_of_Rotate_of_ScaleSquar
e_of_scale_of_shiftMultivariateNormalDiag", batch_shape=[], event_shape=[2],
dtype=float32)
tfp.distributions.BatchReshape("BatchReshapechain_of_tanh_of_Rotate_of_ScaleSqua
re_of_scale_of_shiftMultivariateNormalDiag", batch_shape=[1], event_shape=[2],
dtype=float32)
```



[ ]:
```
imgs = []
n_images = 1000
for i in range(n_images):
  theta=theta_distribution.sample(1).numpy()[0]
  a=alpha_distribution.sample(1).numpy()[0]
  flow_distribution=get_flow_dist(a,theta,base_distribution)
  #print(flow_distribution)
  flow_distribution=tfd.BatchReshape(flow_distribution,[1])
  #print(flow_distribution)
  #plt.subplot(2, 2, i+1)
  #plt.contourf(X, Y, get_densities(flow_distribution).squeeze(), cmap='hot',␣
 ↪levels=100)
```

```
  #plt.title('countour plot from the flow distribution \n alpha={}\n theta={}'.
↪format(a,theta))
  density_i=get_densities(flow_distribution).squeeze()
  imgs.append(get_image_array_from_density_values(density_i))

imgs = np.array(imgs)
```

[ ]:
```
#show 20 images
plt.figure(figsize=(20,10))
for i in range(20):
  plt.subplot(2,10,i+1)
  plt.imshow(imgs[i])
plt.tight_layout()
plt.show()
```



1.3 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.

[ ]:
```
from sklearn.model_selection import train_test_split
```

[ ]:
```
def create_tf_dataset(data,frac,batch_size=20):
  def split_train_test_idx(data,frac):
    train_idx,test_idx=train_test_split(np.arange(len(data)),test_size=frac)
    return train_idx,test_idx
  train_idx,test_idx=split_train_test_idx(data,frac)
  train=data[train_idx]
```

```
    test=data[test_idx]

    train=tf.data.Dataset.from_tensor_slices(tf.cast(train,tf.float32))
    train=train.map(lambda x: x/255.0)
    train=train.map(lambda x: (x,x))
    train=train.batch(batch_size,drop_remainder=True)

    test=tf.data.Dataset.from_tensor_slices(tf.cast(test,tf.float32))
    test=test.map(lambda x: x/255.0)
    test=test.map(lambda x: (x,x))
    test=test.batch(batch_size,drop_remainder=True)
    return train,test
```

[ ]:
```
train,test=create_tf_dataset(imgs,0.2)
print(train.element_spec)
print(test.element_spec)
```

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

[ ]:

[ ]:

[ ]:

[ ]:

1.4 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.

[ ]:
```
def get_prior(latent_dim=2):
    # Define the prior, p(z) - a standard bivariate Gaussian
    prior=tfd.MultivariateNormalDiag(loc=tf.zeros(latent_dim),scale_diag=tfp.
 ↪util.TransformedVariable(tf.random.uniform([latent_dim]),
                                                                              ␣
 ↪    bijector=tfb.Exp()))
    return prior
```

```
[ ]: from tensorflow.keras import Sequential, Model
     from tensorflow.keras.layers import (Dense, Flatten, Reshape, Concatenate,␣
      ↪Conv2D,
                                          UpSampling2D, BatchNormalization)
```

```
[ ]: def get_encoder(latent_dim,img_dim=imgs.shape[1:]):
         prior = get_prior(latent_dim)
         encoder_model = Sequential([

                         tf.keras.layers.InputLayer(input_shape=img_dim),
                        ␣
      ↪Conv2D(32,(4,4),strides=(2,2),activation='relu',padding='SAME'),
                         BatchNormalization(),
                        ␣
      ↪Conv2D(64,(4,4),strides=(2,2),activation='relu',padding='SAME'),
                         BatchNormalization(),
                        ␣
      ↪Conv2D(128,(4,4),strides=(2,2),activation='relu',padding='SAME'),
                         BatchNormalization(),
                        ␣
      ↪Conv2D(256,(4,4),strides=(2,2),activation='relu',padding='SAME'),
                         BatchNormalization(),
                         Flatten(),
                         Dense(tfpl.MultivariateNormalTriL.params_size(latent_dim)),
                         tfpl.MultivariateNormalTriL(latent_dim),
                         tfpl.KLDivergenceAddLoss(prior)])

         return encoder_model
```

```
[ ]: encoder=get_encoder(2)
```

```
WARNING:tensorflow:From /usr/local/lib/python3.7/dist-packages/tensorflow_core/p
ython/ops/linalg/linear_operator_lower_triangular.py:158: calling
LinearOperator.__init__ (from tensorflow.python.ops.linalg.linear_operator) with
graph_parents is deprecated and will be removed in a future version.
Instructions for updating:
Do not pass `graph_parents`.  They will  no longer be used.
```

```
[ ]: encoder.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 18, 18, 32)        1568
_________________________________________________________________
batch_normalization (BatchNo (None, 18, 18, 32)        128
```

```
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 9, 9, 64)          32832
_________________________________________________________________
batch_normalization_1 (Batch (None, 9, 9, 64)          256
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 5, 5, 128)         131200
_________________________________________________________________
batch_normalization_2 (Batch (None, 5, 5, 128)         512
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 3, 3, 256)         524544
_________________________________________________________________
batch_normalization_3 (Batch (None, 3, 3, 256)         1024
_________________________________________________________________
flatten (Flatten)            (None, 2304)              0
_________________________________________________________________
dense (Dense)                (None, 5)                 11525
_________________________________________________________________
multivariate_normal_tri_l (M ((None, 2), (None, 2))    0
_________________________________________________________________
kl_divergence_add_loss (KLDi (None, 2)                 2
=================================================================
Total params: 703,591
Trainable params: 702,631
Non-trainable params: 960
_________________________________________________________________
```

[ ]:
```
imgs.shape[1:]
```

[ ]:
```
(36, 36, 3)
```

[ ]:
```
from tensorflow_probability.python.layers.distribution_layer import␣
↪IndependentBernoulli
def get_decoder(latent_dim,img_dim=imgs.shape[1:]):
    decoder_model=Sequential([
                        ␣
↪#Dense(img_dim[0]*img_dim[1],activation='relu',input_shape=(latent_dim,)),
                        tf.keras.layers.InputLayer(input_shape=(latent_dim,)),
                        Dense(64,activation='relu'),
                        Dense(128,activation='relu'),
                        Dense(256,activation='relu'),
                        Reshape(target_shape=(8,8,4)),
                        UpSampling2D(size=(3,3)),
                        Conv2D(128,(3,3),activation='relu'),
                        UpSampling2D(size=(2,2)),
                        Conv2D(64,(3,3),activation='relu'),
                        UpSampling2D(size=(2,2)),
                        Conv2D(32,(3,3),activation='relu'),
                        UpSampling2D(size=(2,2)),
```

```
                    Conv2D(16,(3,3),activation='relu'),
                    Conv2D(1,(3,3),strides=(2,2)),
                    Flatten(),
                    Dense(tfpl.IndependentBernoulli.params_size(img_dim)),
                    tfpl.IndependentBernoulli(event_shape=img_dim)])

    return decoder_model
```

[ ]:
```
decoder=get_decoder(2)
decoder.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 64)                192
_________________________________________________________________
dense_2 (Dense)              (None, 128)               8320
_________________________________________________________________
dense_3 (Dense)              (None, 256)               33024
_________________________________________________________________
reshape (Reshape)            (None, 8, 8, 4)           0
_________________________________________________________________
up_sampling2d (UpSampling2D) (None, 24, 24, 4)         0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 22, 22, 128)       4736
_________________________________________________________________
up_sampling2d_1 (UpSampling2 (None, 44, 44, 128)       0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 42, 42, 64)        73792
_________________________________________________________________
up_sampling2d_2 (UpSampling2 (None, 84, 84, 64)        0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 82, 82, 32)        18464
_________________________________________________________________
up_sampling2d_3 (UpSampling2 (None, 164, 164, 32)      0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 162, 162, 16)      4624
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 80, 80, 1)         145
_________________________________________________________________
flatten_1 (Flatten)          (None, 6400)              0
_________________________________________________________________
dense_4 (Dense)              (None, 3888)              24887088
_________________________________________________________________
independent_bernoulli (Indep ((None, 36, 36, 3), (None 0
=================================================================
Total params: 25,030,385
```

```
Trainable params: 25,030,385
Non-trainable params: 0
_________________________________________________________________
```

1.5 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.

[ ]:
```
def log_loss(x_true,p_x_given_z):
  #print(p_x_given_z.log_prob(x_true))
  return (-tf.reduce_sum(p_x_given_z.log_prob(x_true)))
```

[ ]:
```
# Connect the encoder and decoder to form the VAE
vae=Model(inputs=encoder.inputs, outputs=decoder(encoder.outputs))
vae.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 36, 36, 3)]       0
_________________________________________________________________
conv2d (Conv2D)              (None, 18, 18, 32)        1568
_________________________________________________________________
batch_normalization (BatchNo (None, 18, 18, 32)        128
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 9, 9, 64)          32832
_________________________________________________________________
batch_normalization_1 (Batch (None, 9, 9, 64)          256
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 5, 5, 128)         131200
_________________________________________________________________
batch_normalization_2 (Batch (None, 5, 5, 128)         512
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 3, 3, 256)         524544
_________________________________________________________________
batch_normalization_3 (Batch (None, 3, 3, 256)         1024
_________________________________________________________________
flatten (Flatten)            (None, 2304)              0
_________________________________________________________________
dense (Dense)                (None, 5)                 11525
_________________________________________________________________
multivariate_normal_tri_l (M ((None, 2), (None, 2))    0
_________________________________________________________________
```

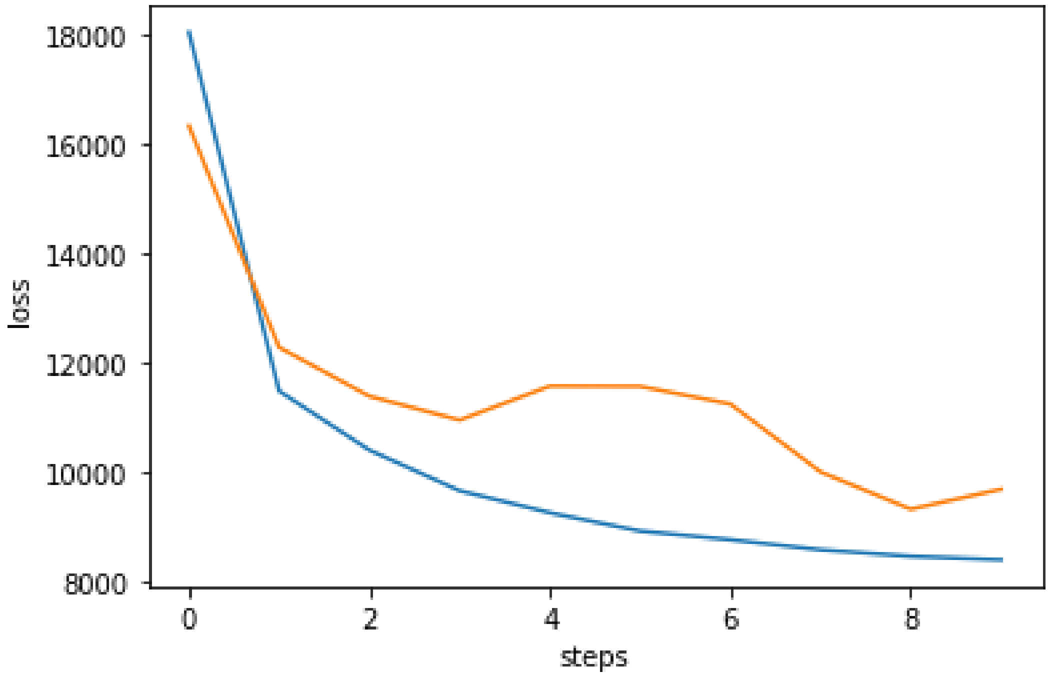
```
kl_divergence_add_loss (KLDi (None, 2)               2
_________________________________________________________________
sequential_1 (Sequential)    (None, 36, 36, 3)        25030385
=================================================================
Total params: 25,733,976
Trainable params: 25,733,016
Non-trainable params: 960
_________________________________________________________________
```

[ ]:
```
# Compile and fit the model
vae.compile(loss=log_loss)
history=vae.fit(train,validation_data=test,epochs=10)
```

```
Train for 40 steps, validate for 10 steps
Epoch 1/10
40/40 [==============================] - 141s 4s/step - loss: 14940.3044 -
val_loss: 12453.6943
Epoch 2/10
40/40 [==============================] - 102s 3s/step - loss: 10553.7632 -
val_loss: 11758.2558
Epoch 3/10
40/40 [==============================] - 105s 3s/step - loss: 9738.9897 -
val_loss: 10595.1911
Epoch 4/10
40/40 [==============================] - 104s 3s/step - loss: 9258.3634 -
val_loss: 11075.0947
Epoch 5/10
40/40 [==============================] - 105s 3s/step - loss: 8929.3205 -
val_loss: 10390.9727
Epoch 6/10
40/40 [==============================] - 103s 3s/step - loss: 8698.4174 -
val_loss: 9230.6545
Epoch 7/10
40/40 [==============================] - 108s 3s/step - loss: 8491.6470 -
val_loss: 8881.2083
Epoch 8/10
40/40 [==============================] - 105s 3s/step - loss: 8492.2896 -
val_loss: 8884.9748
Epoch 9/10
40/40 [==============================] - 109s 3s/step - loss: 8355.3349 -
val_loss: 9213.8297
Epoch 10/10
40/40 [==============================] - 110s 3s/step - loss: 8265.8589 -
val_loss: 8092.5147
```

[ ]:
```
plt.plot(history.history["loss"])
plt.plot(history.history["val_loss"])
```

```
plt.ylabel('loss')
plt.xlabel('steps')
plt.show()
```



[ ]:
```

```

[ ]:
```

```

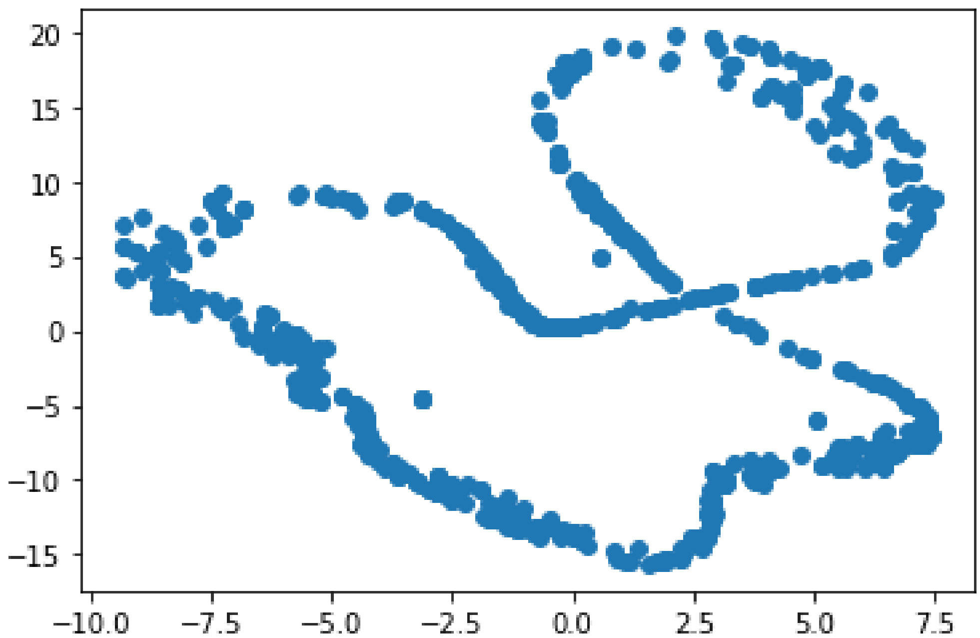
1.6 6. Use the encoder and decoder networks

- 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.

[ ]:
```
idx = np.random.choice(np.arange(imgs.shape[0]), 1000)
embeddings = encoder(imgs[idx]/255.0).mean()
```

```
WARNING:tensorflow:Layer conv2d is casting an input tensor from dtype float64 to
the layer's dtype of float32, which is new behavior in TensorFlow 2.  The layer
has dtype float32 because it's dtype defaults to floatx.

If you intended to run this layer in float32, you can safely ignore this
warning. If in doubt, this warning is likely only an issue if you are porting a
TensorFlow 1.X model to TensorFlow 2.

To change all layers to have dtype float64 by default, call
`tf.keras.backend.set_floatx('float64')`. To change just this layer, pass
dtype='float64' to the layer constructor. If you are the author of this layer,
you can disable autocasting by passing autocast=False to the base Layer
constructor.
```

[ ]:
```
plt.scatter(embeddings[:,0],embeddings[:,1])
plt.show()
```



[ ]:
```
idx = np.random.choice(np.arange(imgs.shape[0]), 4)
reconstructed_imgs= vae(tf.cast(imgs[idx],tf.float32)).mean().numpy() #mean of␣
 ↪outputs

plt.figure(figsize=(15, 20))
for i in range(4):
    plt.subplot(4, 2, 2*i+1)
    plt.imshow(imgs[idx[i]])
```

```
    plt.title("Image: {}".format(i+1))
    plt.axis("off")

    plt.subplot(4, 2, 2*i+2)
    plt.imshow(reconstructed_imgs[i])
    plt.title("Reconstructed: {}".format(i+1))
    plt.axis("off")
plt.show()
```

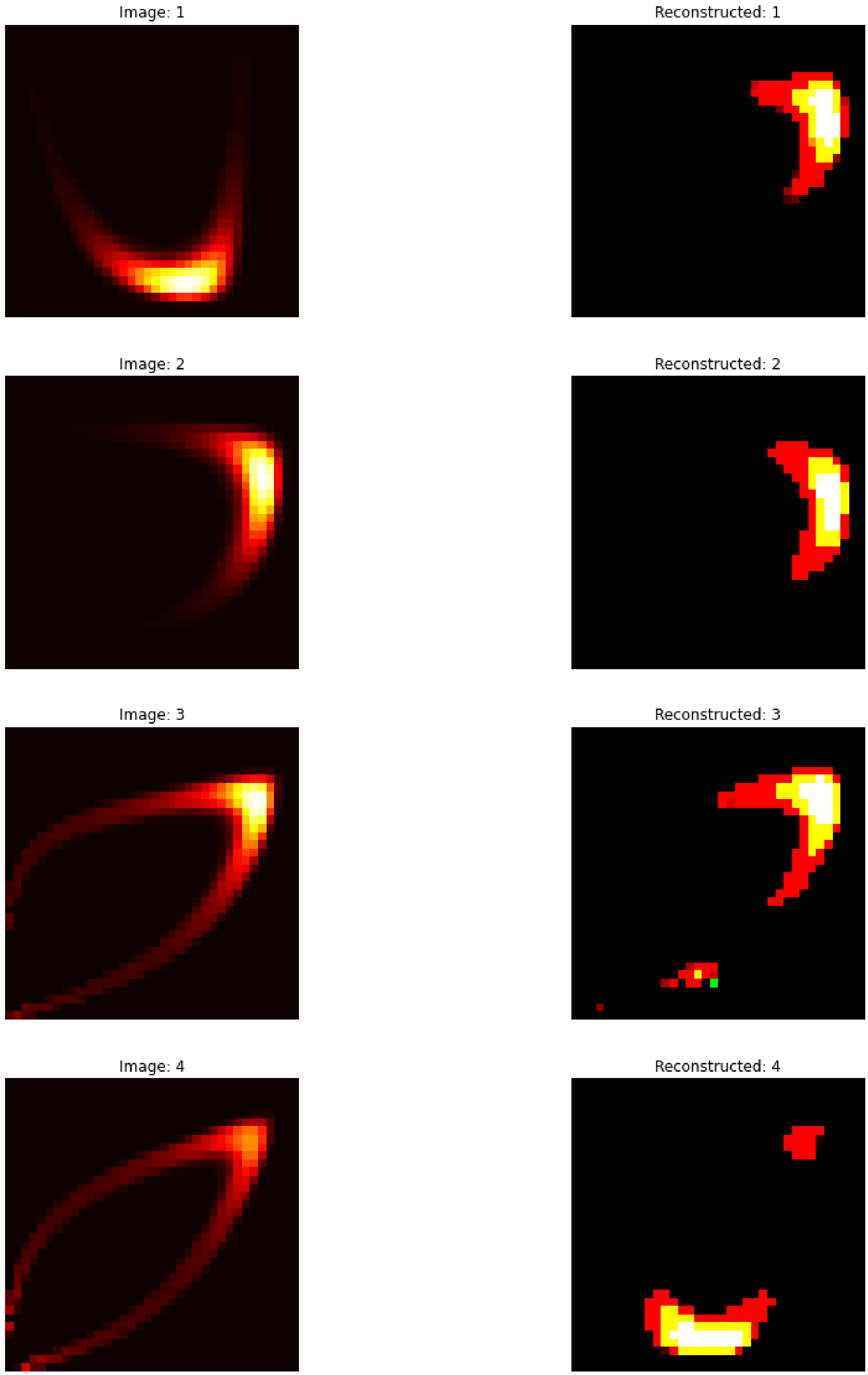


Image: 1
Reconstructed: 1
Image: 2
Reconstructed: 2
Image: 3
Reconstructed: 3
Image: 4
Reconstructed: 4

[57]:

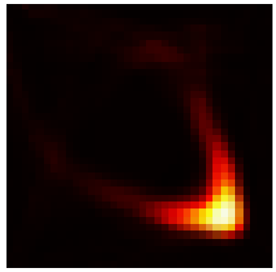
```

```

[57]: <tf.Tensor: shape=(2,), dtype=bool, numpy=array([False, False])>

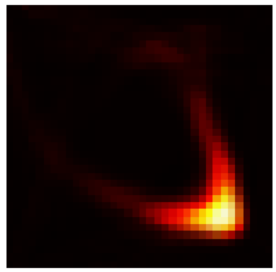
[61]:

```
prior_dist=get_prior(latent_dim=2)
embeddings=prior_dist.sample(6)
reconstructed_imgs= decoder(embeddings).mean()

plt.figure(figsize=(15, 20))
for i in range(6):
    plt.subplot(6, 2, 2*i+2)
    plt.imshow(reconstructed_imgs[i])
    plt.title("Reconstructed: {}".format(i+1))
    plt.axis("off")
plt.show()
```

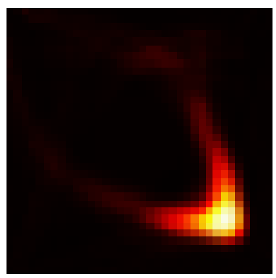
Reconstructed: 1



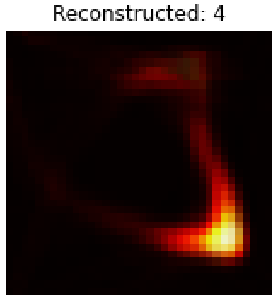
Reconstructed: 2



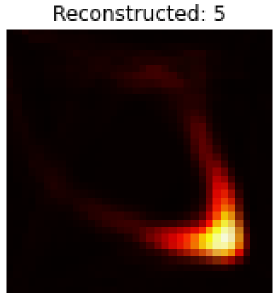
Reconstructed: 3



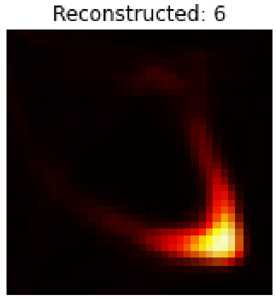
Reconstructed: 4



Reconstructed: 5



Reconstructed: 6



[ ]:

[ ]:

1.7 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.

[62]:
```
# 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.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)
```

[63]:
```
# Create the animation

a = get_animation(2, decoder, interpolation_length=200)
HTML(a.to_html5_video())
```

[63]: <IPython.core.display.HTML object>