C3W5_Capstone_Project

August 31, 2021

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.

```
In [1]: import tensorflow as tf
        import tensorflow_probability as tfp
        tfd = tfp.distributions
        tfb = tfp.bijectors
        tfpl = tfp.layers

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

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

The normalising flow

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

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

The transformed random variable x is given by $x = f_5(f_4(f_3(f_2(f_1(z)))))$. * You should use or construct bijectors for each of the transformations f_i, $i = 1, \ldots, 5$, and use `tfb.Chain` and `tfb.TransformedDistribution` to construct the final transformed distribution. * Ensure to implement the `log_det_jacobian` methods for any subclassed bijectors that you write. * Display a scatter plot of samples from the base distribution. * Display 4 scatter plot images of the transformed distribution from your random normalising flow, using samples of θ and a. Fix the axes of these 4 plots to the range $[-1, 1]$.

In [2]:
```
mu = 0
sigma = 0.3
base_distribution = tfd.MultivariateNormalDiag(loc=[mu, mu], scale_diag=[sigma, sigma]
```

In [3]:
```
import math
theta_dist = tfd.Uniform(low=0, high=2*math.pi)
a_dist = tfd.Normal(loc=3, scale=1.)
```

In [4]:
```
class F3Bijector(tfb.Bijector):

    def __init__(self, a, name='F3', **kwargs):
        self.a = a
        super(F3Bijector, self).__init__(
            is_constant_jacobian=True,
            forward_min_event_ndims=0,
            name=name,
             **kwargs)

    def _forward(self, x):
        x = tf.cast(x, tf.float32)
        return tf.concat([x[..., :1], x[..., 1:] + self.a * tf.pow(x[..., :1], 2)], ax
```

```
        def _inverse(self, y):
            y = tf.cast(y, tf.float32)
            return tf.concat([y[..., :1], y[..., 1:] - self.a * tf.pow(y[..., :1], 2)], ax

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

```
In [5]: class RotationBijector(tfb.Bijector):

        def __init__(self, theta, name='Rotation', **kwargs):
            super(RotationBijector, self).__init__(
                is_constant_jacobian=True,
                forward_min_event_ndims=1,
                validate_args=False,
                name=name,
                 **kwargs)
            self.rotation_matrix = tf.convert_to_tensor([[tf.cos(theta), -tf.sin(theta)],
                                                         [tf.sin(theta), tf.cos(theta)]], dtype

        def _forward(self, x):
            x = tf.cast(x, tf.float32)
            return tf.linalg.matvec(self.rotation_matrix, x)

        def _inverse(self, y):
            y = tf.cast(y, tf.float32)
            return tf.linalg.matvec(tf.transpose(self.rotation_matrix), y)

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

```
In [6]: def get_bjiectors_chain(a, theta):
        bjiectors = []
        bjiectors.append(tfb.Shift([0., -2.])) # f1
        bjiectors.append(tfb.Scale([1, 0.5])) # f2
        bjiectors.append(F3Bijector(a)) # f3
        bjiectors.append(RotationBijector(theta)) # f4
        bjiectors.append(tfb.Tanh()) # f5

        return tfb.Chain(list(reversed(bjiectors)))
```

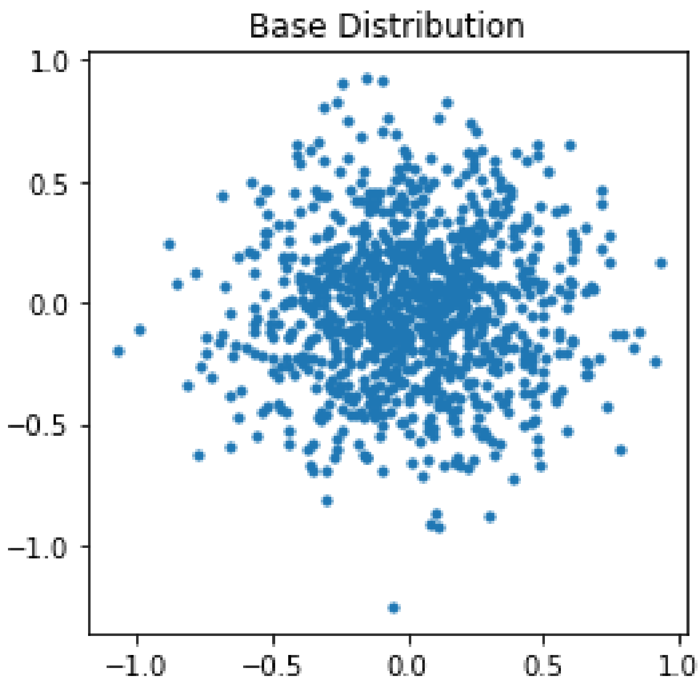
```
In [7]: def get_transformed_distribution(distribution, a, theta):
        return tfd.TransformedDistribution(distribution=distribution, bijector=get_bjiecto
```

```
In [40]: def display_samples_from_base_distribution(n_samples):
         z = base_distribution.sample(n_samples).numpy().squeeze()
         plt.figure(figsize=(4, 4))
         plt.scatter(z[:, 0], z[:, 1], s=10)
         plt.title("Base Distribution")
         plt.show()
```

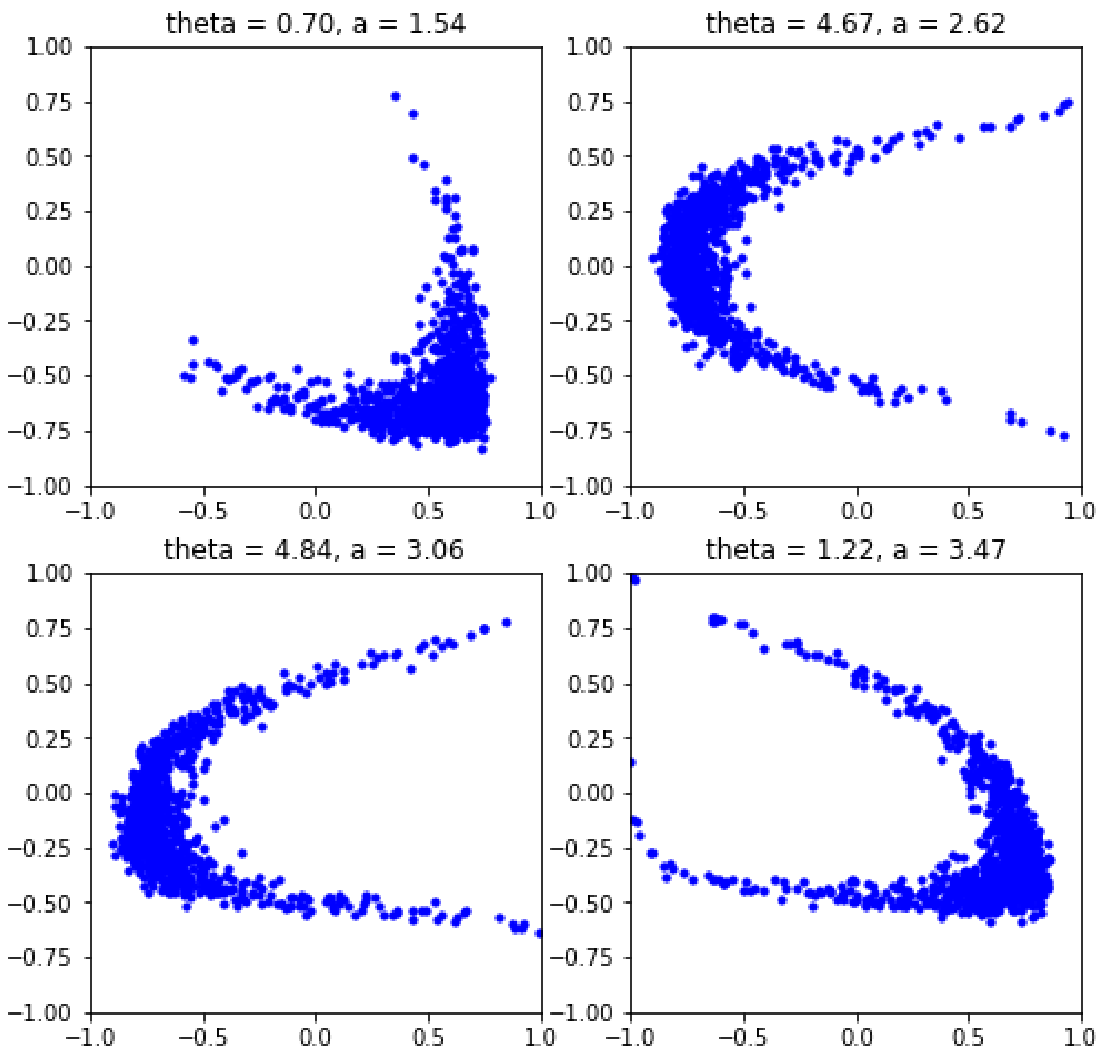
In [9]:
```
def display_samples_from_random_flow(flow, a, theta, n_samples):
    samples = flow.sample(n_samples).numpy().squeeze()
    plt.scatter(samples[:,0], samples[:, 1], s=10)
    plt.title("theta = {:.2f}, a = {:.2f}".format(theta, a))
    plt.xlim([-1,1])
    plt.ylim([-1,1])
```

In [10]:
```
n_samples = 1000
```

In [41]:
```
display_samples_from_base_distribution(n_samples)
```



In [42]:
```
plt.figure(figsize = (8, 8))
for i, col in enumerate(range(4)):
    theta = theta_dist.sample(1).numpy()[0]
    a = a_dist.sample(1).numpy()[0]

    flow = get_transformed_distribution(base_distribution, a, theta)

    plt.subplot(2, 2, i+1)
    display_samples_from_random_flow(flow, a, theta, n_samples)
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 dis-

tribution 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.

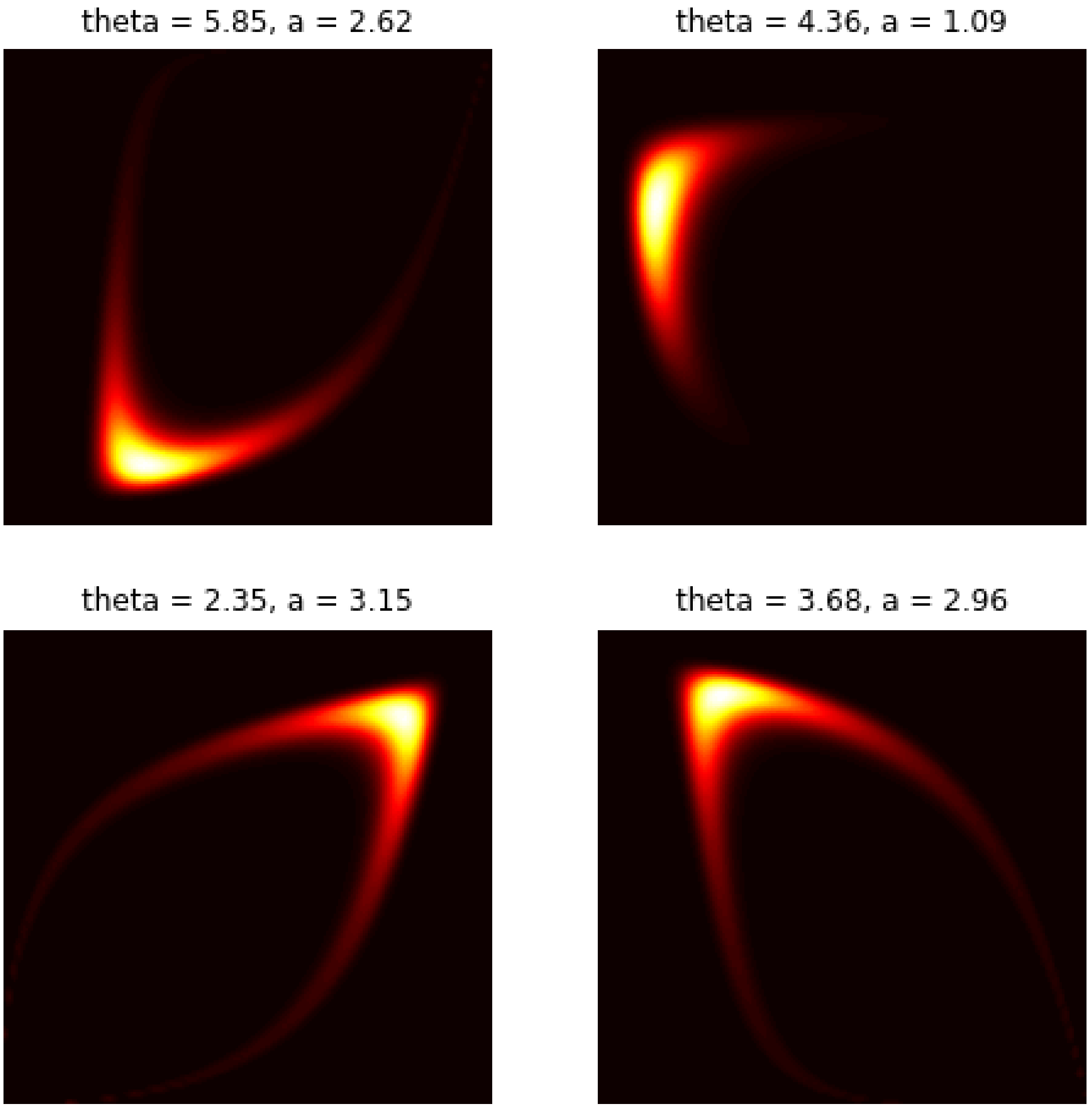
- As well as the get_densities function, the get_image_array_from_density_values function will help you to generate the dataset.
    - This function creates a numpy array for an image of the contour plot for a given set of density values Z. Feel free to choose your own options for the contour plots.
- Display a sample of 20 images from your generated dataset in a figure.

In [13]:
```
# 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
    array Z of shape (batch_shape, 100, 100) of density values, that can be used to m
    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 [14]:
```
# 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 im
    """
    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
```

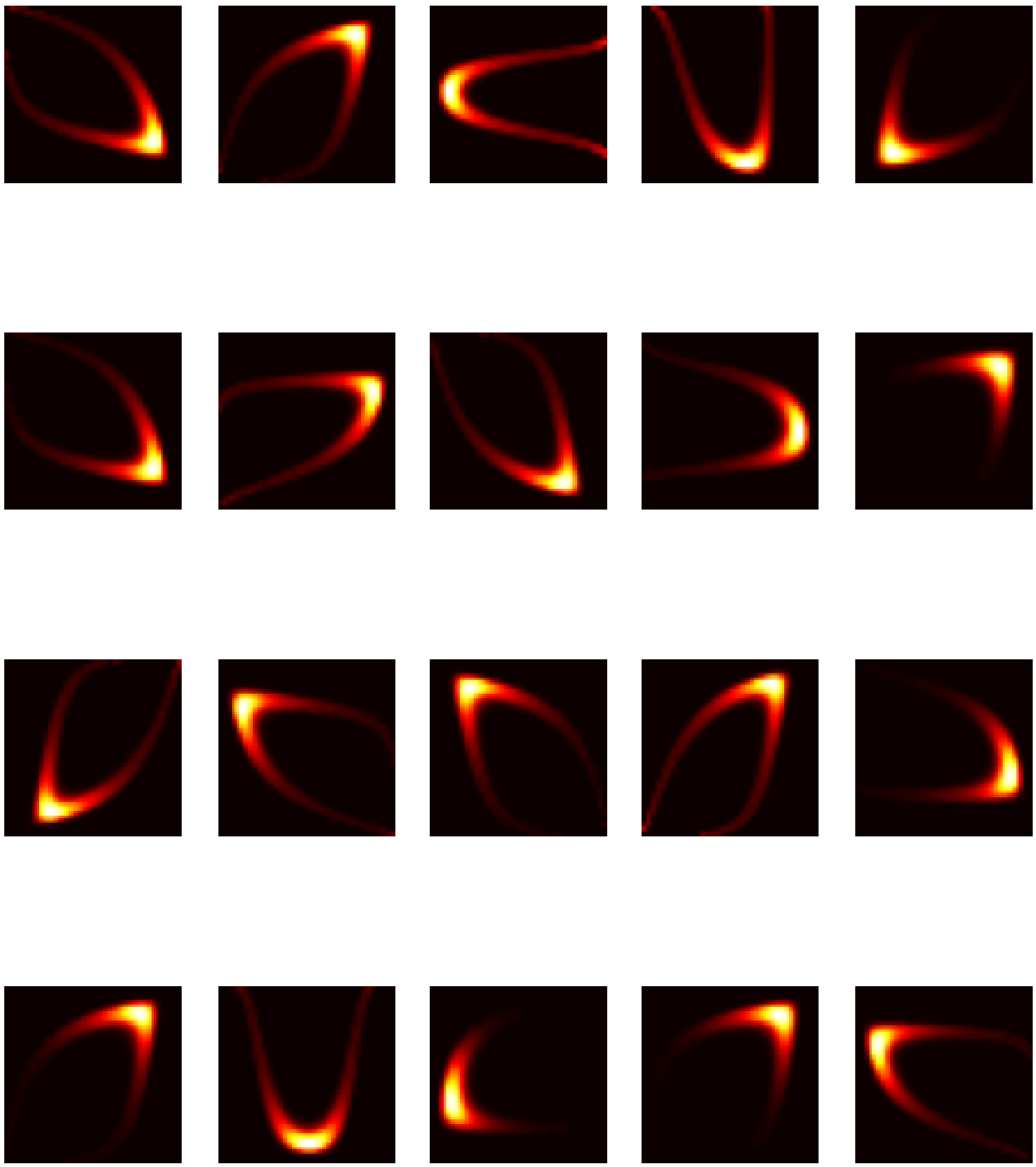
```
In [15]: plt.figure(figsize = (8, 8))
         for i in range(4):
             theta = theta_dist.sample(1).numpy()[0]
             a = a_dist.sample(1).numpy()[0]

             flow = get_transformed_distribution(base_distribution, a, theta)
             flow = tfd.BatchReshape(flow, [1])

             plt.subplot(2, 2, i+1)
             plt.contourf(X, Y, get_densities(flow).squeeze(), cmap='hot', levels=100)
             plt.title("theta = {:.2f}, a = {:.2f}".format(theta, a))
             plt.axis('off')
         plt.show()
```



In [16]:
```
images = []
N = 2000

for _ in range(N):
    theta = theta_dist.sample(1).numpy()[0]
    a = a_dist.sample(1).numpy()[0]

    flow = get_transformed_distribution(base_distribution, a, theta)
    flow = tfd.BatchReshape(flow, [1])

    Z = get_densities(flow).squeeze()

    images.append(get_image_array_from_density_values(Z))
```

```
        images = np.array(images)
```

```
In [17]: plt.figure(figsize=(8, 10))
         for i in range(20):
             plt.subplot(4, 5, i+1)
             idx = np.random.randint(0, N)
             plt.imshow(images[idx])
             plt.axis("off")
         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.

In [18]:
```
def get_tf_dataset(data, batch_size, shuffle=False):
    dataset = tf.data.Dataset.from_tensor_slices(data)
    dataset = dataset.map(lambda x: x/255.0)
    dataset = dataset.map(lambda x: (x, x))
    dataset = dataset.batch(batch_size, drop_remainder=True)
    if shuffle:
        dataset = dataset.shuffle(data.shape[0])
    return dataset
```

In [19]:
```
train_fraction = 0.8
train_len = int(train_fraction * images.shape[0])

train_data = images[0:train_len]
test_data = images[train_len:]

train_dataset = get_tf_dataset(train_data.astype(np.float32), 20, shuffle=True)
test_dataset = get_tf_dataset(test_data.astype(np.float32), 20)
```

In [20]:
```
print(train_dataset.element_spec)
```

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

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.

In [21]:
```
from tensorflow.keras import Sequential, Model
from tensorflow.keras.layers import (Dense, Flatten, Reshape, Concatenate, Conv2D,
                                     UpSampling2D, BatchNormalization)
```

In [22]:
```
latent_dim = 2
event_shape = images.shape[1:]
```

In [23]:
```
def get_prior(latent_dim):
    return tfd.MultivariateNormalDiag(loc=tf.Variable(tf.zeros(latent_dim)),
                                      scale_diag=tfp.util.TransformedVariable(tf.ones
                                                                          bijector=tfb.So
                                      )
prior = get_prior(latent_dim)
```

In [24]:
```
def get_encoder(latent_dim, prior):
    return Sequential([
            Conv2D(32, (4,4), activation='relu', strides=(1,1), padding='same', input_
            BatchNormalization(),
            Conv2D(64, (4,4), activation='relu', strides=(2,2), padding='same'),
            BatchNormalization(),
            Conv2D(128, (4,4), activation='relu', strides=(2,2), padding='same'),
            BatchNormalization(),
            Conv2D(256, (4,4), activation='relu', strides=(2,2), padding='same'),
            BatchNormalization(),
            Flatten(),
            Dense(units=tfpl.MultivariateNormalTriL.params_size(latent_dim)),
            tfpl.MultivariateNormalTriL(event_size=latent_dim),
            tfpl.KLDivergenceAddLoss(prior,
                                     use_exact_kl = False,
                                     test_points_fn = lambda q:q.sample(3),
                                     test_points_reduce_axis=(0,1))
    ])
```

In [25]:
```
encoder = get_encoder(latent_dim, prior)
encoder.summary()
```

```
Model: "sequential"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 36, 36, 32)        1568
_________________________________________________________________
batch_normalization (BatchNo (None, 36, 36, 32)        128
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 18, 18, 64)        32832
_________________________________________________________________
batch_normalization_1 (Batch (None, 18, 18, 64)        256
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 9, 9, 128)         131200
_________________________________________________________________
batch_normalization_2 (Batch (None, 9, 9, 128)         512
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 5, 5, 256)         524544
```

```
_________________________________________________________________
batch_normalization_3 (Batch (None, 5, 5, 256)         1024
_________________________________________________________________
flatten (Flatten)            (None, 6400)              0
_________________________________________________________________
dense (Dense)                (None, 5)                 32005
_________________________________________________________________
multivariate_normal_tri_l (M multiple                  0
_________________________________________________________________
kl_divergence_add_loss (KLDi multiple                  4
=================================================================
Total params: 724,073
Trainable params: 723,113
Non-trainable params: 960
_________________________________________________________________
```

In [26]:
```python
def get_decoder(latent_dim):
    return Sequential([
            Dense(6400, activation='relu', input_shape=(latent_dim,)),
            Reshape((5, 5, 256)),
            UpSampling2D((2,2)),
            Conv2D(128, (3,3), activation='relu', padding='same'),
            UpSampling2D((2,2)),
            Conv2D(64, (3,3), activation='relu', padding='same'),
            UpSampling2D((2,2)),
            Conv2D(32, (3,3), activation='relu', padding='valid'),
            # UpSampling2D((2,2)),
            # Conv2D(128, (3,3), activation='relu', padding='same'),
            Conv2D(3, (3,3), padding='valid'),
            Flatten(),
            tfpl.IndependentBernoulli(event_shape=event_shape)
    ])
```

In [27]:
```python
decoder = get_decoder(latent_dim)
decoder.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 6400)              19200
_________________________________________________________________
reshape (Reshape)            (None, 5, 5, 256)         0
_________________________________________________________________
up_sampling2d (UpSampling2D) (None, 10, 10, 256)       0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 10, 10, 128)       295040
```

```
_________________________________________________________________
up_sampling2d_1 (UpSampling2 (None, 20, 20, 128)       0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 20, 20, 64)        73792
_________________________________________________________________
up_sampling2d_2 (UpSampling2 (None, 40, 40, 64)        0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 38, 38, 32)        18464
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 36, 36, 3)         867
_________________________________________________________________
flatten_1 (Flatten)          (None, 3888)              0
_________________________________________________________________
independent_bernoulli (Indep multiple                  0
=================================================================
Total params: 407,363
Trainable params: 407,363
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.

```
In [28]: vae = Model(inputs=encoder.inputs, outputs=decoder(encoder.outputs))
```

```
In [39]: vae.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_input (InputLayer)    [(None, 36, 36, 3)]       0
_________________________________________________________________
conv2d (Conv2D)              (None, 36, 36, 32)        1568
_________________________________________________________________
batch_normalization (BatchNo (None, 36, 36, 32)        128
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 18, 18, 64)        32832
_________________________________________________________________
batch_normalization_1 (Batch (None, 18, 18, 64)        256
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 9, 9, 128)         131200
_________________________________________________________________
```

```
batch_normalization_2 (Batch (None, 9, 9, 128)         512
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 5, 5, 256)         524544
_________________________________________________________________
batch_normalization_3 (Batch (None, 5, 5, 256)         1024
_________________________________________________________________
flatten (Flatten)            (None, 6400)              0
_________________________________________________________________
dense (Dense)                (None, 5)                 32005
_________________________________________________________________
multivariate_normal_tri_l (M multiple                  0
_________________________________________________________________
kl_divergence_add_loss (KLDi multiple                  4
_________________________________________________________________
sequential_1 (Sequential)    multiple                  407363
=================================================================
Total params: 1,131,436
Trainable params: 1,130,476
Non-trainable params: 960
_________________________________________________________________
```

In [29]:
```python
def reconstruction_loss(batch_of_images, decoding_dist):
    return -tf.reduce_mean(decoding_dist.log_prob(batch_of_images))
```

In [31]:
```python
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
vae.compile(optimizer=optimizer, loss=reconstruction_loss)
```

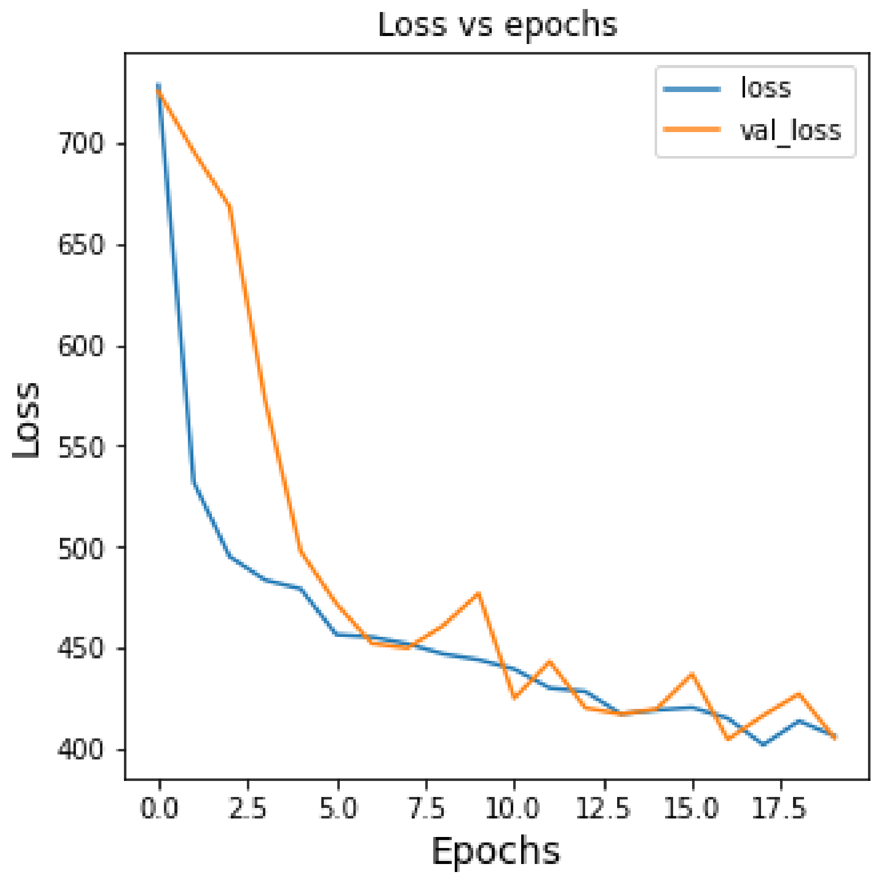
In [32]:
```python
history = vae.fit(train_dataset, validation_data=test_dataset, epochs=20)
```

```
Epoch 1/20
80/80 [==============================] - 46s 539ms/step - loss: 728.1896 - val_loss: 725.1144
Epoch 2/20
80/80 [==============================] - 42s 530ms/step - loss: 531.9446 - val_loss: 695.2062
Epoch 3/20
80/80 [==============================] - 43s 530ms/step - loss: 495.2586 - val_loss: 668.0511
Epoch 4/20
80/80 [==============================] - 43s 536ms/step - loss: 483.6056 - val_loss: 572.3756
Epoch 5/20
80/80 [==============================] - 43s 534ms/step - loss: 479.2614 - val_loss: 497.7943
Epoch 6/20
80/80 [==============================] - 43s 535ms/step - loss: 456.7144 - val_loss: 471.9682
Epoch 7/20
80/80 [==============================] - 43s 541ms/step - loss: 455.4901 - val_loss: 452.3803
Epoch 8/20
80/80 [==============================] - 43s 535ms/step - loss: 452.1444 - val_loss: 450.0581
Epoch 9/20
80/80 [==============================] - 43s 535ms/step - loss: 447.1755 - val_loss: 461.0799
Epoch 10/20
```

```
80/80 [==============================] - 42s 530ms/step - loss: 444.2690 - val_loss: 476.9537
Epoch 11/20
80/80 [==============================] - 43s 535ms/step - loss: 439.6835 - val_loss: 425.2427
Epoch 12/20
80/80 [==============================] - 43s 540ms/step - loss: 430.3006 - val_loss: 443.3583
Epoch 13/20
80/80 [==============================] - 43s 540ms/step - loss: 428.5930 - val_loss: 420.4771
Epoch 14/20
80/80 [==============================] - 43s 542ms/step - loss: 417.4929 - val_loss: 417.4368
Epoch 15/20
80/80 [==============================] - 43s 536ms/step - loss: 419.3777 - val_loss: 419.9648
Epoch 16/20
80/80 [==============================] - 43s 539ms/step - loss: 420.6538 - val_loss: 437.2065
Epoch 17/20
80/80 [==============================] - 44s 546ms/step - loss: 415.3503 - val_loss: 404.8840
Epoch 18/20
80/80 [==============================] - 44s 547ms/step - loss: 402.2370 - val_loss: 416.8009
Epoch 19/20
80/80 [==============================] - 45s 559ms/step - loss: 413.9939 - val_loss: 427.4025
Epoch 20/20
80/80 [==============================] - 45s 560ms/step - loss: 406.5812 - val_loss: 405.6213
```

In [43]:
```python
import matplotlib.pyplot as plt

fig, axes = plt.subplots(1, 1, figsize=(5, 5))

train_loss = history.history['loss']
val_loss = history.history['val_loss']

axes.set_xlabel("Epochs", fontsize=14)
axes.set_ylabel("Loss", fontsize=14)
axes.set_title('Loss vs epochs')
axes.plot(train_loss, label='loss')
axes.plot(val_loss, label='val_loss')
axes.legend()
```

Out[43]:
```
<matplotlib.legend.Legend at 0x7fe60e12c390>
```



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.

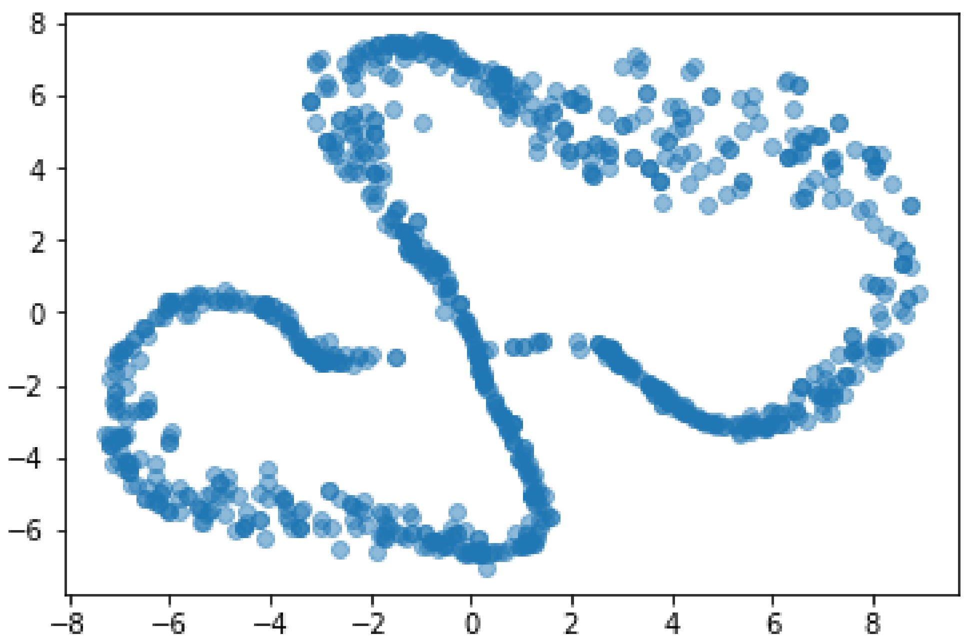
```
In [34]: def get_random_idx(pool_size, n_samples):
             return np.random.choice(np.arange(pool_size), n_samples)
```

```
In [35]: n_embeddings = 1000
         embeddings_idx = get_random_idx(images.shape[0], n_embeddings)
```

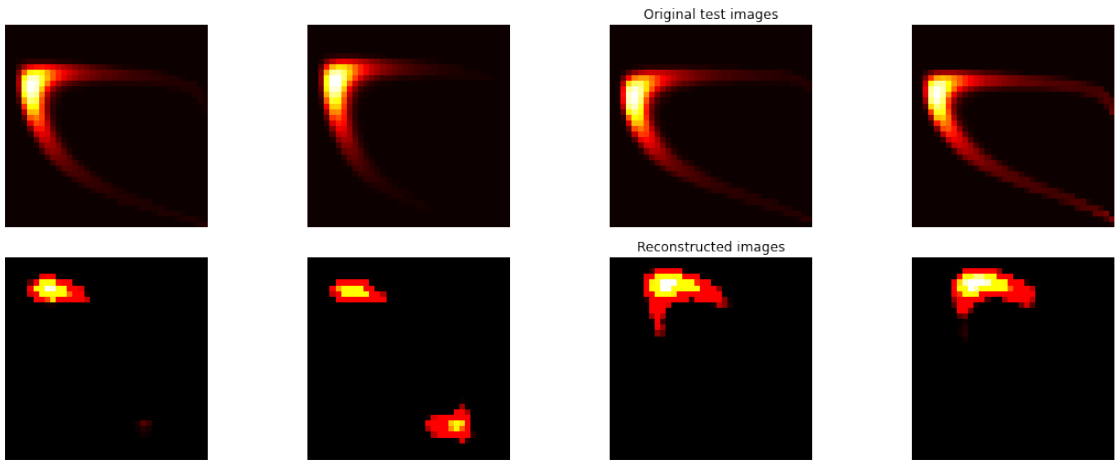
```
print(images[embeddings_idx].shape)
embeddings = encoder(images[embeddings_idx]/255.0).mean()
```

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

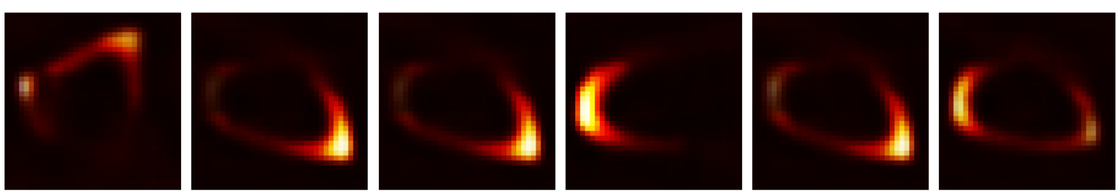
In [36]:
```
plt.scatter(embeddings[:,0], embeddings[:,1], alpha=0.5)
plt.show()
```



In [37]:
```
n_reconstructions = 4
reconstruction_idx = get_random_idx(images.shape[0], n_reconstructions)
reconstructions = vae(images[reconstruction_idx]).mean().numpy()

f, axs = plt.subplots(2, n_reconstructions, figsize=(16, 6))
axs[0, n_reconstructions // 2].set_title("Original test images")
axs[1, n_reconstructions // 2].set_title("Reconstructed images")
for j in range(n_reconstructions):
    axs[0, j].imshow(images[reconstruction_idx[j]])
    axs[1, j].imshow(reconstructions[j])
    axs[0, j].axis('off')
    axs[1, j].axis('off')

plt.tight_layout();
```




```
In [38]: n_latent = 6
         z = prior.sample(n_latent)
         generated_images = decoder(z).mean().numpy()

         f, axs = plt.subplots(1, n_latent, figsize=(16, 6))

         for j in range(n_latent):
             axs[j].imshow(generated_images[j])
             axs[j].axis('off')

         plt.tight_layout();
```



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.

```
In [ ]: # 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

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

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

In [ ]:
```
# Create the animation

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

In [ ]: