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.

```
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 p
import math as m
from sklearn.model_selection import train_test_split
from matplotlib import gridspec
%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 0 and covariance matrix $\Sigma=\sigma^2 I_2$, with $\sigma=0.3$.
- This normalising flow uses bijectors that are parameterised by the following random variables:
  - $\theta \sim U ¿$
  - $a \sim N(3,1)$

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

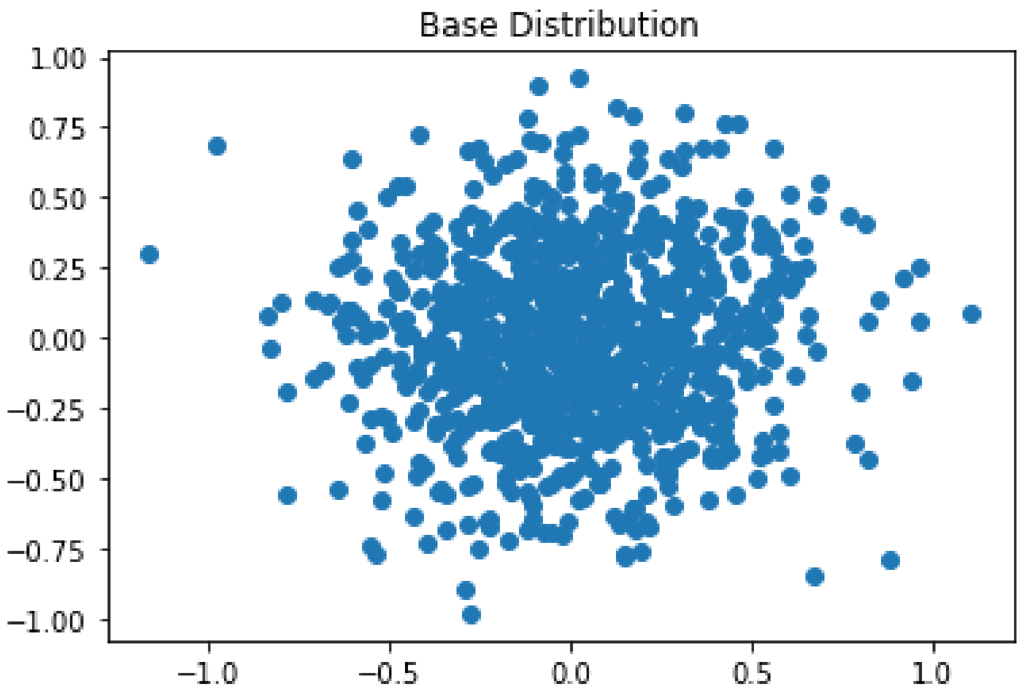
- $f_1(z)=(z_1, z_2-2)$,
- $f_2(z)=\left(z_1, \frac{z_2}{2}\right)$,
- $f_3(z)=(z_1, z_2+a z_1^2)$,
- $f_4(z)=R z$, where R is a rotation matrix with angle θ,
- $f_5(z)=\tanh(z)$, where the tanh function is applied elementwise.

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

- 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 $[-1,1]$.

```
theta_dist = tfd.Uniform(low = 0, high = 2*np.pi)
a_dist = tfd.Normal(loc = 3, scale = 1)

mu, sigma = 0, 0.3
base_dist = tfd.MultivariateNormalDiag(loc = [mu, mu],  scale_diag = 
[sigma, sigma])

def getBaseDist():
  sigma = 0.3
  mvn = tfd.MultivariateNormalDiag(loc=[0., 0.],
```

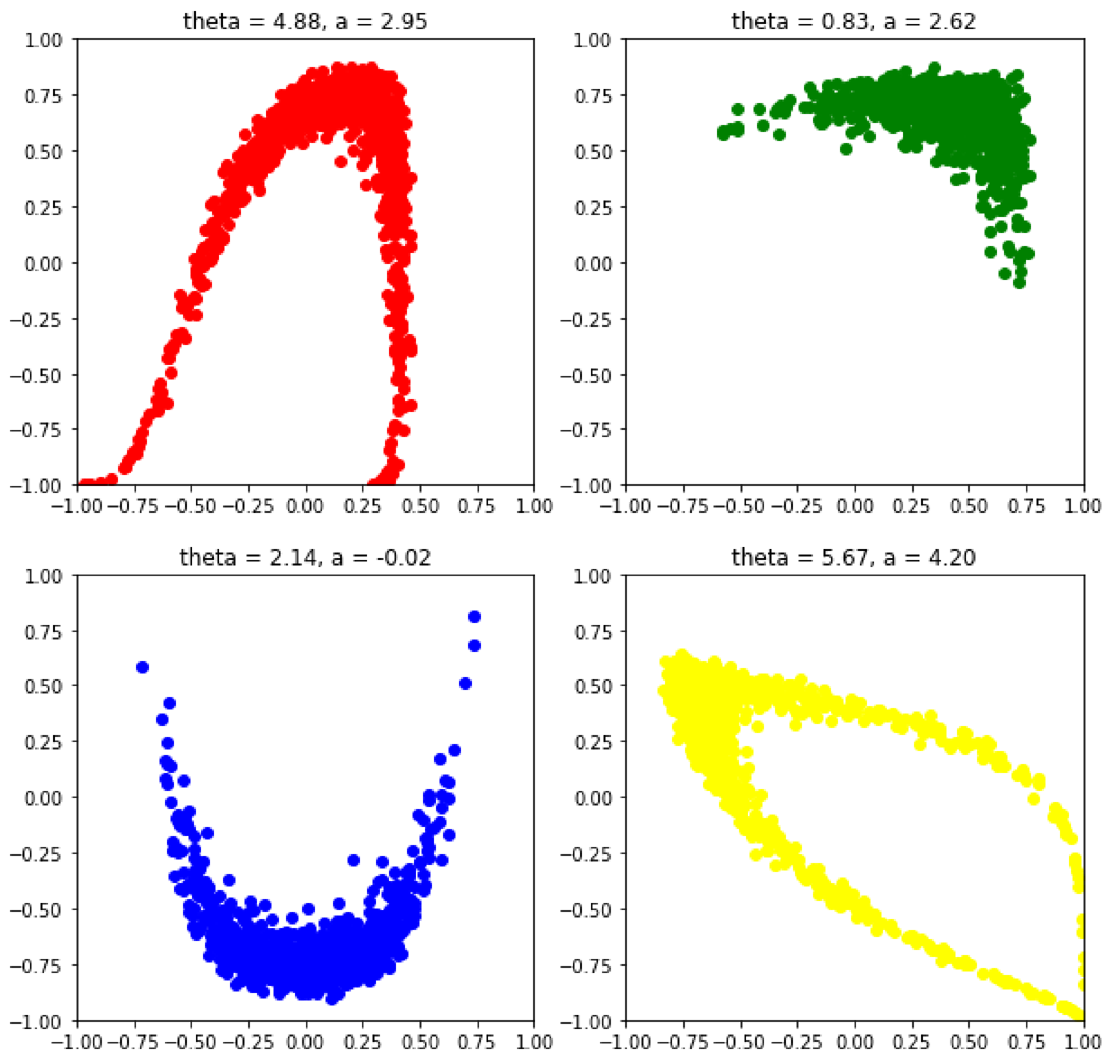
```
                                         scale_diag=[sigma, sigma])
  return mvn

n = 1000
mvn = getBaseDist()

z = mvn.sample(n).numpy().squeeze()
p.figure()
p.scatter(z[:, 0], z[:, 1])
p.title("Base Distribution")
p.show()
```




```
class Polynomial(tfb.Bijector):
    def __init__(self, a, name="Polynomial", **kwargs):
        super(Polynomial, self).__init__(forward_min_event_ndims=1,
                                         name=name,
                                         is_constant_jacobian=True,
                                         validate_args=False,
                                         **kwargs)

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

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

    def _inverse(self, y):
```

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

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



class Rotation(tfb.Bijector):
    def __init__(self, theta, name="Rotation", **kwargs):
        super(Rotation, self).__init__(name=name,
                                       forward_min_event_ndims=1,
                                       validate_args=False,
                                       **kwargs)

        self.rot_matrix = tf.convert_to_tensor([[tf.cos(theta), -
tf.sin(theta)],
                                                [tf.sin(theta),
tf.cos(theta)]], dtype=tf.float32)

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

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

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

def createFlow(a, theta):
  f1 = tfb.Shift([0, -2])
  f2 = tfb.Scale([1, 0.5])
  f3 = Polynomial(a)
  f4 = Rotation(theta)
  f5 = tfb.Tanh()

  return tfb.Chain([f5, f4, f3, f2, f1])

def createTransformedDist(theta, a, base_dist):
  return tfd.TransformedDistribution(distribution=base_dist,
                                     bijector=createFlow(theta, a))


p.figure(figsize = (10, 10))

base_dist = getBaseDist()
```

```python
colors = ['red', 'green', 'blue', 'yellow']
for i, col in enumerate(colors):
  theta = theta_dist.sample(1).numpy()[0]
  a = a_dist.sample(1).numpy()[0]
  transformed_dist = createTransformedDist(theta, a, base_dist)
  p.subplot(2, 2, i+1)
  samples = transformed_dist.sample(1000).numpy().squeeze()
  p.scatter(samples[:,0], samples[:, 1], color=col)
  p.title("theta = {:.2f}, a = {:.2f}".format(theta, a))
  p.xlim([-1,1])
  p.ylim([-1,1])
```



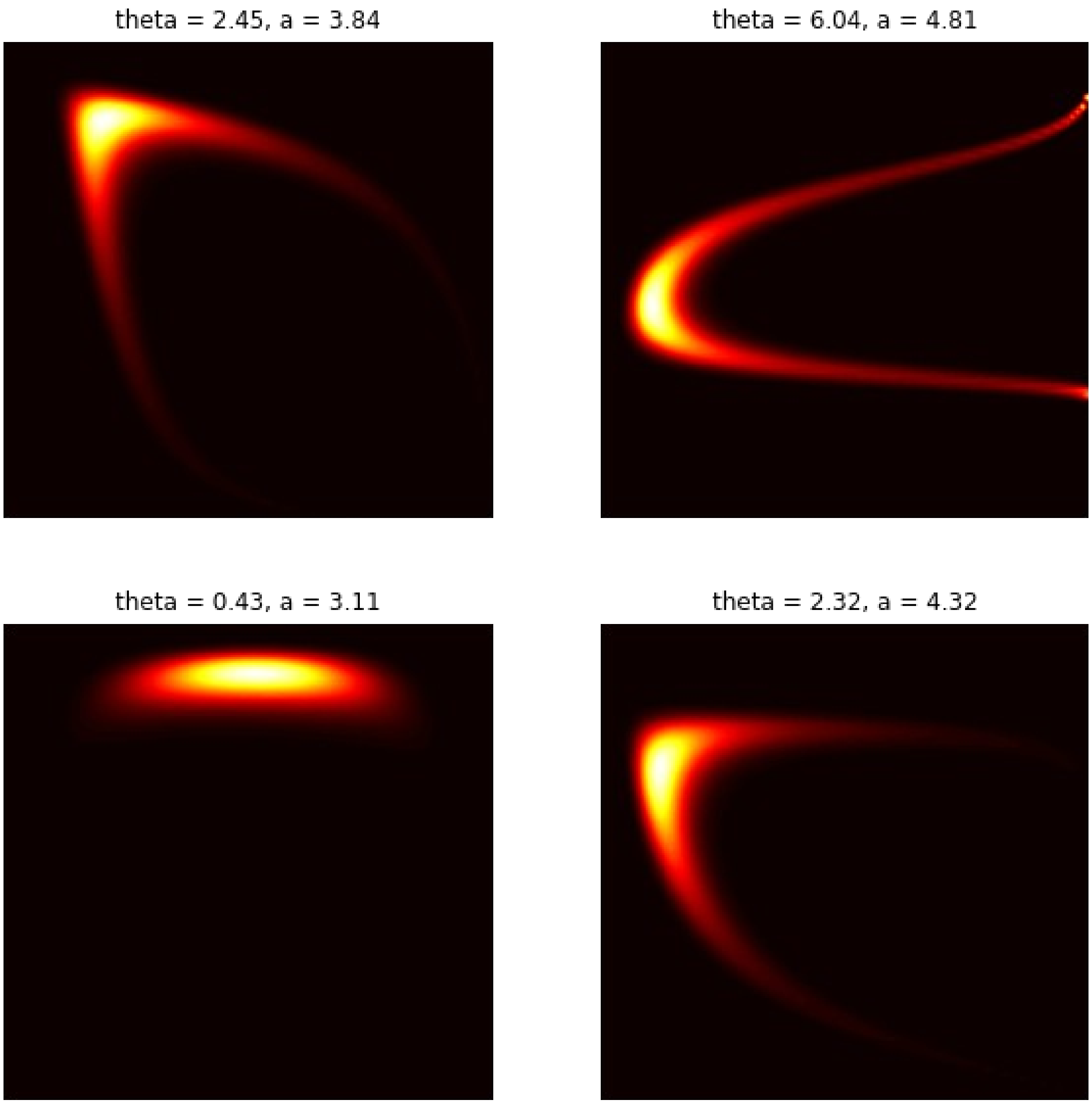
2. Create the image dataset

- 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.

```
# Helper function to compute transformed distribution densities

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

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

# 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

p.figure(figsize = (10, 10))
for i in range(4):
  theta = theta_dist.sample(1).numpy()[0]
  a = a_dist.sample(1).numpy()[0]
  transformed_dist = createTransformedDist(theta, a, base_dist)
  transformed_dist = tfd.BatchReshape(transformed_dist, [1])
  p.subplot(2, 2, i+1)

  p.contourf(X, Y, get_densities(transformed_dist).squeeze(),
cmap='hot', levels=100)
  p.title("theta = {:.2f}, a = {:.2f}".format(theta, a))
  p.xlim([-1,1])
  p.ylim([-1,1])
  p.axis('off')
p.show()
```

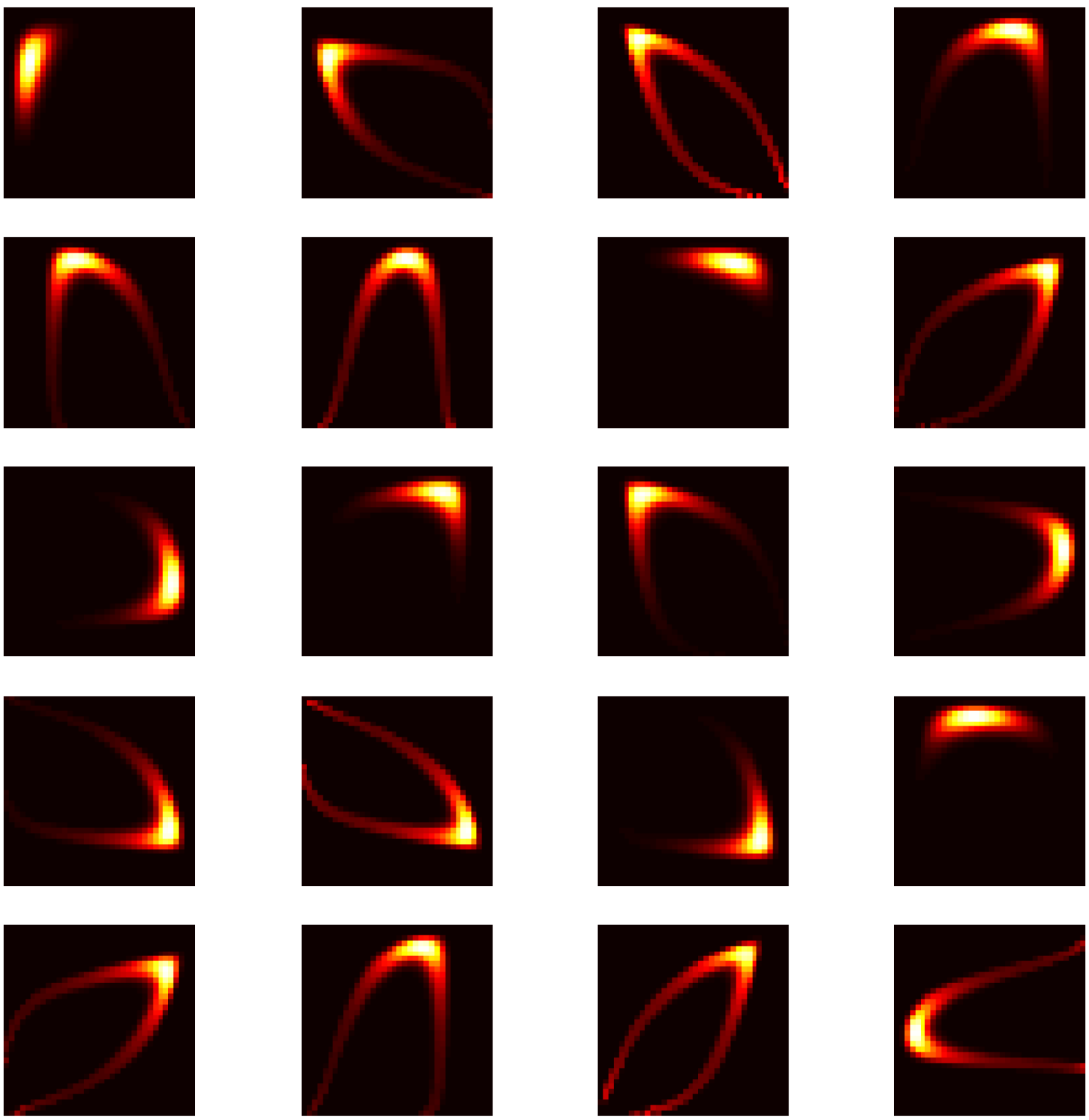



```
num_images = 1000
def generateDataset():
  images = []
  for i in range(num_images):
    theta = theta_dist.sample(1).numpy()[0]
    a = a_dist.sample(1).numpy()[0]
    transformed_dist = createTransformedDist(theta, a, base_dist)
    transformed_dist = tfd.BatchReshape(transformed_dist, [1])
    Z = get_densities(transformed_dist).squeeze()
    image = get_image_array_from_density_values(Z)
    images.append(image)

  return images

images = generateDataset()
```

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

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

```
p.figure(figsize = (15, 15))
for i in range(20):
  idx = np.random.randint(0, 1000)
  p.subplot(5, 4, i+1)
  image = images[i, ...]
  p.axis('off')
  p.imshow(image)
p.show()
```



3. Make `tf.data.Dataset` objects

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

```
data = images.astype(dtype=np.float32)
batch_size = 20
train, test = train_test_split(data, train_size=0.8, test_size=0.2)

ds_train = tf.data.Dataset.from_tensor_slices(train)
ds_train = ds_train.map(lambda t: t/255.0)
ds_train = ds_train.map(lambda x: (x, x))
ds_train = ds_train.batch(batch_size, drop_remainder=True)
ds_train = ds_train.shuffle(buffer_size=1001)

ds_test = tf.data.Dataset.from_tensor_slices(test)
ds_test = ds_test.map(lambda t: t/255.0)
ds_test = ds_test.map(lambda x: (x, x))
ds_test = ds_test.batch(batch_size, drop_remainder=True)

print(ds_train.element_spec)
```

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

4. Build the encoder and decoder networks

- You should now create the encoder and decoder for the variational autoencoder algorithm.
- You should design these networks yourself, subject to the following constraints:
  - The encoder and decoder networks should be built using the `Sequential` class.

  - The encoder and decoder networks should use probabilistic layers where necessary to represent distributions.
  - The prior distribution should be a zero-mean, isotropic Gaussian (identity covariance matrix).
  - The encoder network should add the KL divergence loss to the model.
- Print the model summary for the encoder and decoder networks.

```
from tensorflow.keras import Sequential, Model
from tensorflow.keras.layers import (Dense, Flatten, Reshape,
Concatenate, Conv2D,
                                     UpSampling2D, BatchNormalization)
tfd = tfp.distributions
tfb = tfp.bijectors
tfpl = tfp.layers

def get_prior(latent_dim):
    prior = tfd.MultivariateNormalDiag(loc =
tf.Variable(tf.zeros(latent_dim), dtype=tf.float32),
                                       scale_diag =
tfp.util.TransformedVariable(initial_value = tf.ones(latent_dim,
dtype=tf.float32),

bijector = tfb.Softplus(),

dtype = tf.float32)
                                       )

    return prior

def get_encoder(latent_dim):

    encoder = Sequential([
        Conv2D(32, 4, activation='relu', strides=2, padding='SAME',
input_shape=(36, 36, 3)),
        BatchNormalization(),
        Conv2D(64, 4, activation='relu', strides=2, padding='SAME'),
        BatchNormalization(),
        Conv2D(128, 4, activation='relu', strides=2, padding='SAME'),
        BatchNormalization(),
        Conv2D(256, 4, activation='relu', strides=2, padding='SAME'),
        BatchNormalization(),
        Flatten(),
        Dense(tfpl.MultivariateNormalTriL.params_size(latent_dim)),
        tfpl.MultivariateNormalTriL(latent_dim),
        tfpl.KLDivergenceAddLoss(get_prior(latent_dim),
                                 use_exact_kl = False,
                                 test_points_fn = lambda
q:q.sample(5),
                                 test_points_reduce_axis=(0,1))
    ])
```

```
    return encoder

latent_dim = 2
encoder = get_encoder(latent_dim)

encoder.summary()
```

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

 batch_normalization (BatchN  (None, 18, 18, 32)       128
 ormalization)

 conv2d_1 (Conv2D)           (None, 9, 9, 64)          32832

 batch_normalization_1 (Batc  (None, 9, 9, 64)         256
 hNormalization)

 conv2d_2 (Conv2D)           (None, 5, 5, 128)         131200

 batch_normalization_2 (Batc  (None, 5, 5, 128)        512
 hNormalization)

 conv2d_3 (Conv2D)           (None, 3, 3, 256)         524544

 batch_normalization_3 (Batc  (None, 3, 3, 256)        1024
 hNormalization)

 flatten (Flatten)           (None, 2304)              0

 dense (Dense)               (None, 5)                 11525

 multivariate_normal_tri_l (  ((None, 2),              0
 MultivariateNormalTriL)      (None, 2))

 kl_divergence_add_loss (KLD  (None, 2)                4
 ivergenceAddLoss)

=================================================================
Total params: 703,593
Trainable params: 702,633
Non-trainable params: 960
_________________________________________________________________
```

```
def get_decoder(latent_dim):
    """
```

```
    This function should build a CNN decoder model according to the
above specification.
    The function takes latent_dim as an argument, which should be used
to define the model.
    Your function should return the decoder model.
    """
    image_dim = (36, 36, 3)
    decoder = Sequential([
        Dense(4096, activation='relu', input_shape=(latent_dim,)),
        Reshape((4, 4, 256)),
        UpSampling2D(size=(2, 2)),
        Conv2D(128, 3, activation='relu', padding='SAME'),
        UpSampling2D(size=(2, 2)),
        Conv2D(64, 3, activation='relu', padding='SAME'),
        UpSampling2D(size=(2, 2)),
        Conv2D(32, 3, activation='relu', padding='SAME'),
        UpSampling2D(size=(2, 2)),
        Conv2D(128, 3, activation='relu', padding='SAME'),
        Conv2D(3, 3, padding='SAME'),
        Flatten(),
        Dense(tfpl.IndependentBernoulli.params_size(image_dim)),
        tfpl.IndependentBernoulli(event_shape=image_dim)
    ])

    return decoder

decoder = get_decoder(latent_dim)

decoder.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_1 (Dense)             (None, 4096)              12288

 reshape (Reshape)           (None, 4, 4, 256)         0

 up_sampling2d (UpSampling2D  (None, 8, 8, 256)        0
 )

 conv2d_4 (Conv2D)           (None, 8, 8, 128)         295040

 up_sampling2d_1 (UpSampling  (None, 16, 16, 128)      0
 2D)

 conv2d_5 (Conv2D)           (None, 16, 16, 64)        73792

 up_sampling2d_2 (UpSampling  (None, 32, 32, 64)       0
 2D)
```

```
 conv2d_6 (Conv2D)           (None, 32, 32, 32)        18464

 up_sampling2d_3 (UpSampling  (None, 64, 64, 32)       0
 2D)

 conv2d_7 (Conv2D)           (None, 64, 64, 128)       36992

 conv2d_8 (Conv2D)           (None, 64, 64, 3)         3459

 flatten_1 (Flatten)         (None, 12288)             0

 dense_2 (Dense)             (None, 3888)              47779632

 independent_bernoulli (Inde  ((None, 36, 36, 3),      0
 pendentBernoulli)            (None, 36, 36, 3))

=================================================================
Total params: 48,219,667
Trainable params: 48,219,667
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.

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

def reconstruction_loss(batch_of_images, decoding_dist):
    """
    This function should compute and return the average expected
reconstruction loss,
    as defined above.
    The function takes batch_of_images (Tensor containing a batch of
input images to
    the encoder) and decoding_dist (output distribution of decoder
after passing the
    image batch through the encoder and decoder) as arguments.
    The function should return the scalar average expected
reconstruction loss.
    """
    return -
```

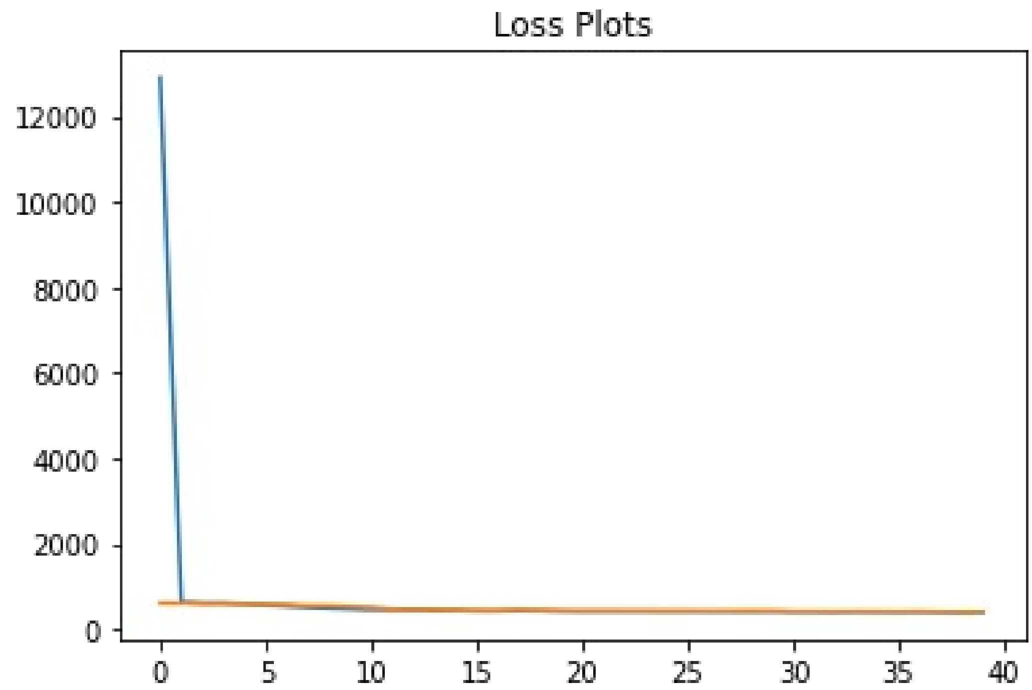
```
tf.reduce_sum(decoding_dist.log_prob(batch_of_images))/batch_of_images
.shape[0]

optimizer = tf.keras.optimizers.Adam(learning_rate=0.005)
vae.compile(optimizer=optimizer, loss=reconstruction_loss)

history = vae.fit(ds_train, validation_data=ds_test, epochs=40)
```

```
Epoch 1/40
40/40 [==============================] - 19s 90ms/step - loss:
12912.0527 - val_loss: 613.5903
Epoch 2/40
40/40 [==============================] - 3s 77ms/step - loss: 636.7983
- val_loss: 614.3257
Epoch 3/40
40/40 [==============================] - 3s 74ms/step - loss: 612.2168
- val_loss: 607.0528
Epoch 4/40
40/40 [==============================] - 3s 75ms/step - loss: 610.6771
- val_loss: 608.5170
Epoch 5/40
40/40 [==============================] - 3s 75ms/step - loss: 596.0806
- val_loss: 595.1943
Epoch 6/40
40/40 [==============================] - 3s 72ms/step - loss: 569.0111
- val_loss: 580.5905
Epoch 7/40
40/40 [==============================] - 3s 74ms/step - loss: 545.3578
- val_loss: 582.9617
Epoch 8/40
40/40 [==============================] - 3s 74ms/step - loss: 517.4138
- val_loss: 552.4846
Epoch 9/40
40/40 [==============================] - 3s 75ms/step - loss: 496.3533
- val_loss: 551.5870
Epoch 10/40
40/40 [==============================] - 3s 72ms/step - loss: 478.5088
- val_loss: 532.7495
Epoch 11/40
40/40 [==============================] - 3s 74ms/step - loss: 461.5001
- val_loss: 518.2845
Epoch 12/40
40/40 [==============================] - 3s 72ms/step - loss: 462.6373
- val_loss: 476.1368
Epoch 13/40
40/40 [==============================] - 3s 74ms/step - loss: 455.7077
- val_loss: 452.5573
Epoch 14/40
40/40 [==============================] - 3s 71ms/step - loss: 449.7452
- val_loss: 459.9346
Epoch 15/40
```

```
40/40 [==============================] - 3s 72ms/step - loss: 446.1797
- val_loss: 443.4171
Epoch 16/40
40/40 [==============================] - 3s 72ms/step - loss: 432.9568
- val_loss: 432.9738
Epoch 17/40
40/40 [==============================] - 3s 74ms/step - loss: 430.6093
- val_loss: 437.9270
Epoch 18/40
40/40 [==============================] - 3s 75ms/step - loss: 440.2687
- val_loss: 453.9005
Epoch 19/40
40/40 [==============================] - 3s 72ms/step - loss: 433.3456
- val_loss: 440.6649
Epoch 20/40
40/40 [==============================] - 3s 74ms/step - loss: 426.3310
- val_loss: 429.5184
Epoch 21/40
40/40 [==============================] - 3s 74ms/step - loss: 424.1861
- val_loss: 422.8842
Epoch 22/40
40/40 [==============================] - 3s 74ms/step - loss: 426.7986
- val_loss: 420.3523
Epoch 23/40
40/40 [==============================] - 3s 72ms/step - loss: 422.5291
- val_loss: 425.3409
Epoch 24/40
40/40 [==============================] - 3s 72ms/step - loss: 424.3690
- val_loss: 422.0417
Epoch 25/40
40/40 [==============================] - 3s 74ms/step - loss: 421.2646
- val_loss: 420.9660
Epoch 26/40
40/40 [==============================] - 3s 72ms/step - loss: 421.7173
- val_loss: 422.8844
Epoch 27/40
40/40 [==============================] - 3s 72ms/step - loss: 421.9485
- val_loss: 417.8807
Epoch 28/40
40/40 [==============================] - 3s 74ms/step - loss: 416.3178
- val_loss: 427.3566
Epoch 29/40
40/40 [==============================] - 3s 74ms/step - loss: 410.9361
- val_loss: 435.4379
Epoch 30/40
40/40 [==============================] - 3s 71ms/step - loss: 421.1538
- val_loss: 421.9115
Epoch 31/40
40/40 [==============================] - 3s 72ms/step - loss: 412.5978
- val_loss: 411.8120
```

```
Epoch 32/40
40/40 [==============================] - 3s 72ms/step - loss: 411.7654
- val_loss: 414.0316
Epoch 33/40
40/40 [==============================] - 3s 72ms/step - loss: 409.0431
- val_loss: 413.8081
Epoch 34/40
40/40 [==============================] - 3s 72ms/step - loss: 414.1893
- val_loss: 410.1511
Epoch 35/40
40/40 [==============================] - 3s 83ms/step - loss: 410.5901
- val_loss: 413.5927
Epoch 36/40
40/40 [==============================] - 3s 84ms/step - loss: 410.0312
- val_loss: 408.7293
Epoch 37/40
40/40 [==============================] - 3s 73ms/step - loss: 409.0816
- val_loss: 417.9182
Epoch 38/40
40/40 [==============================] - 3s 72ms/step - loss: 404.4967
- val_loss: 402.7227
Epoch 39/40
40/40 [==============================] - 3s 72ms/step - loss: 397.7386
- val_loss: 403.2838
Epoch 40/40
40/40 [==============================] - 3s 72ms/step - loss: 401.7977
- val_loss: 403.3502
```

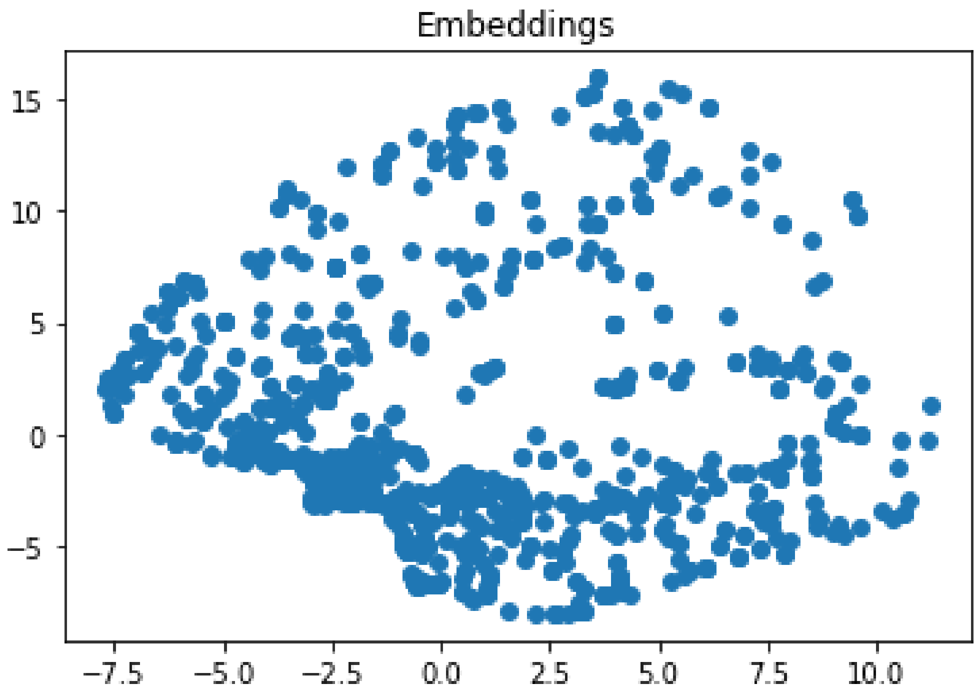
```
p.plot(history.history["loss"])
p.plot(history.history["val_loss"])
p.title("Loss Plots")

p.show()
```



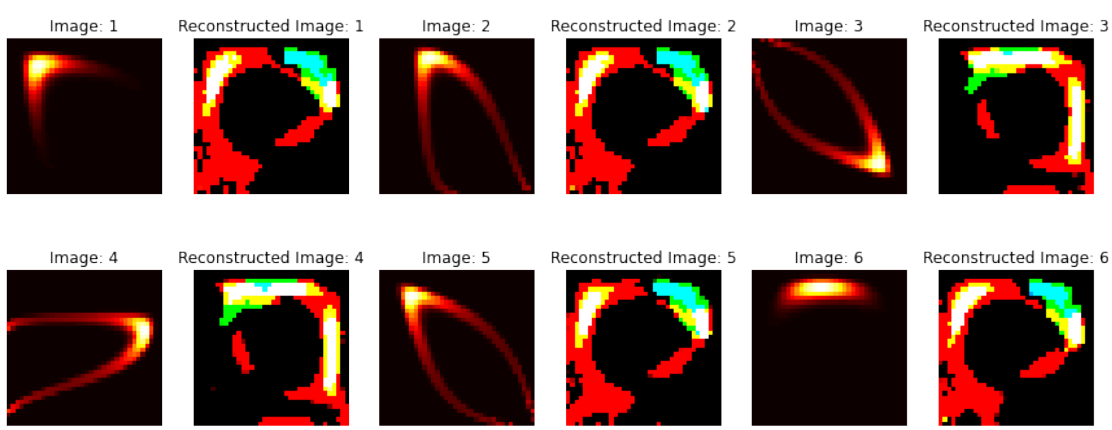
6. Use the encoder and decoder networks

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

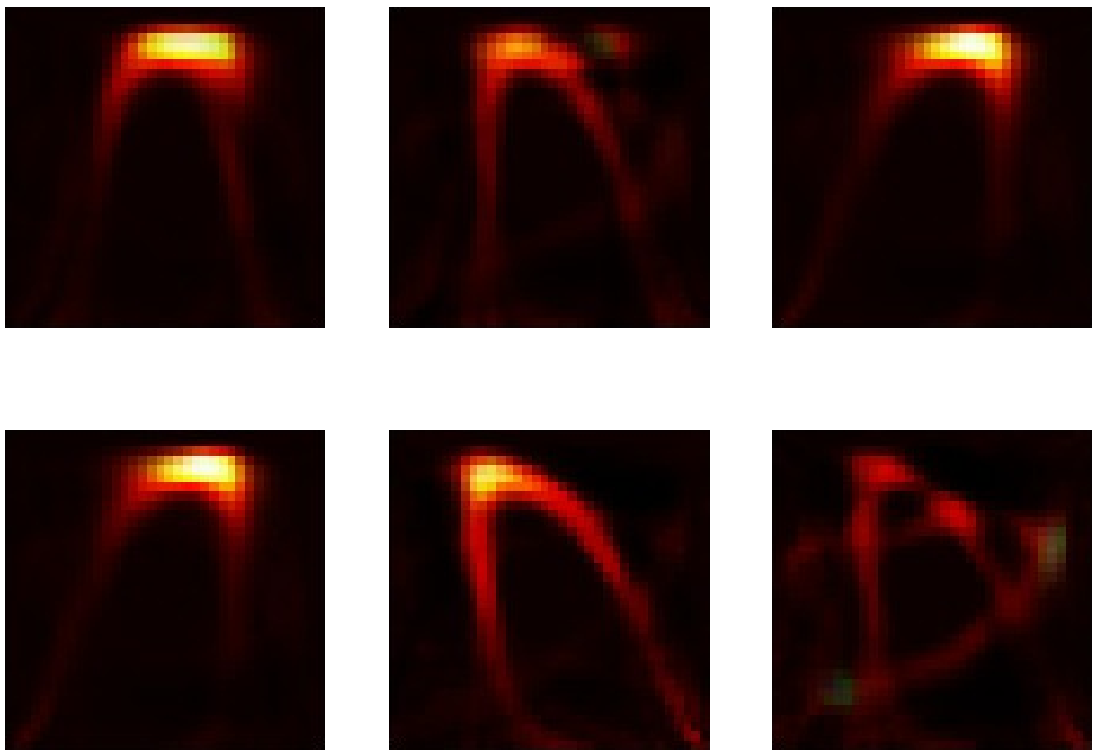
```
idx = np.random.choice(np.arange(images.shape[0]), 1000)
embeddings = encoder(images[idx]/255.0).mean()

p.figure()
p.scatter(embeddings[:, 0], embeddings[:, 1])
p.title("Embeddings")
p.show()
```




```
N = 6
idx = np.random.choice(np.arange(data.shape[0]), N)
reconstructed_images = vae(data[idx]).mean().numpy()
p.figure(figsize=(15, 6))
for i in range(N):
    p.subplot(2, 6, 2*i+1)
    p.imshow(data[idx[i]].astype(np.uint8))
    p.title("Image: {}".format(i+1))
    p.axis("off")

    p.subplot(2, 6, 2*i+2)
    p.imshow(reconstructed_images[i])
    p.title("Reconstructed Image: {}".format(i+1))
    p.axis("off")
p.show()
```



```
N = 6
embeddings = np.random.uniform(-2, 2, (N, latent_dim))
reconstructed_images = decoder(embeddings).mean()


fig = p.figure(figsize=(14, 6))
gs = gridspec.GridSpec(2, 5)

for i in range(2):
    for j in range(3):
        ax1 = p.subplot(gs[i, 2+j])
        ax1.imshow(reconstructed_images[2*i + j])
        ax1.set_axis_off()

p.show()
```



Make a video of latent space interpolation (not assessed)

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

```
# 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)

# Create the animation

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