Capstone Project

August 23, 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

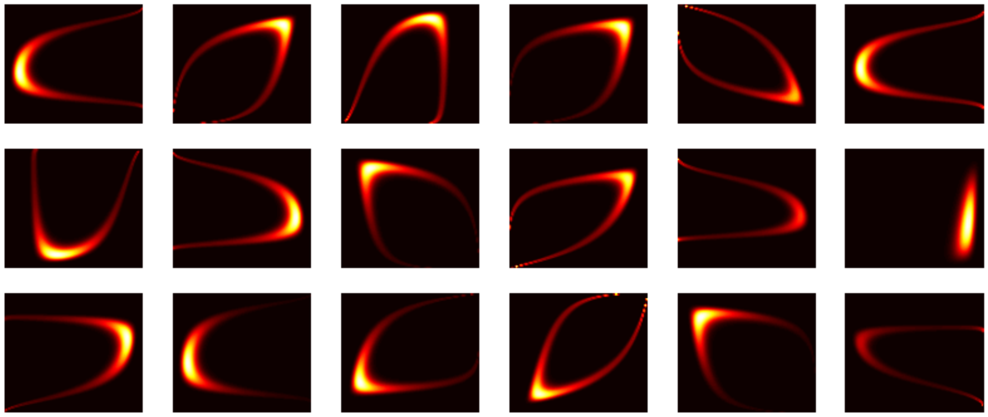
When you have completed the Capstone project notebook, you will submit a pdf of the notebook for peer review. First ensure that the notebook has been fully executed from beginning to end, and all of the cell outputs are visible. This is important, as the grading rubric depends on the reviewer being able 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]:
```
# I install this version of matplotlib to avoid some visualization issues that me and o
# see on the forum:
# https://www.coursera.org/learn/probabilistic-deep-learning-with-tensorflow2/discussio
!pip install matplotlib==3.2.2
```

```
Collecting matplotlib==3.2.2
  Downloading matplotlib-3.2.2-cp37-cp37m-manylinux1_x86_64.whl (12.4 MB)
     || 12.4 MB 16.3 MB/s eta 0:00:01
```



Flags overview image

```
Requirement already satisfied: kiwisolver>=1.0.1 in /opt/conda/lib/python3.7/site-packages (fro
Requirement already satisfied: cycler>=0.10 in /opt/conda/lib/python3.7/site-packages (from ma
Requirement already satisfied: python-dateutil>=2.1 in /opt/conda/lib/python3.7/site-packages
Requirement already satisfied: numpy>=1.11 in /opt/conda/lib/python3.7/site-packages (from mat
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /opt/conda/lib/pytho
Requirement already satisfied: setuptools in /opt/conda/lib/python3.7/site-packages (from kiwi
Requirement already satisfied: six in /opt/conda/lib/python3.7/site-packages (from cycler>=0.10
Installing collected packages: matplotlib
  Attempting uninstall: matplotlib
    Found existing installation: matplotlib 3.0.3
    Uninstalling matplotlib-3.0.3:
      Successfully uninstalled matplotlib-3.0.3
Successfully installed matplotlib-3.2.2
WARNING: You are using pip version 20.1; however, version 21.2.4 is available.You should consid
```

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

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

```
In [3]: # define the base distribution
        dim = 2
        mean = np.zeros(dim)
        sigma = 0.6
        scale = np.ones_like(mean) * sigma**2
        base_distribution = tfd.MultivariateNormalDiag(loc=mean.astype('float32'),scale_diag=sc
```

```
In [4]: # subclass bijection that involves mixing components
        class F3(tfb.Bijector):

            def __init__(self, a):
                self.a = a
                super(F3, self).__init__(forward_min_event_ndims=1)

            def _forward(self,z):

                z_0, z_1 = tf.split(z,num_or_size_splits=2,axis=-1)

                zt_0 = z_0

                zt_1 = z_1 + self.a * (z_0**2)

                ret = tf.concat([zt_0, zt_1],axis=-1)

                return ret

            def _inverse(self,z):

                z_0, z_1 = tf.split(z,num_or_size_splits=2,axis=-1)
```

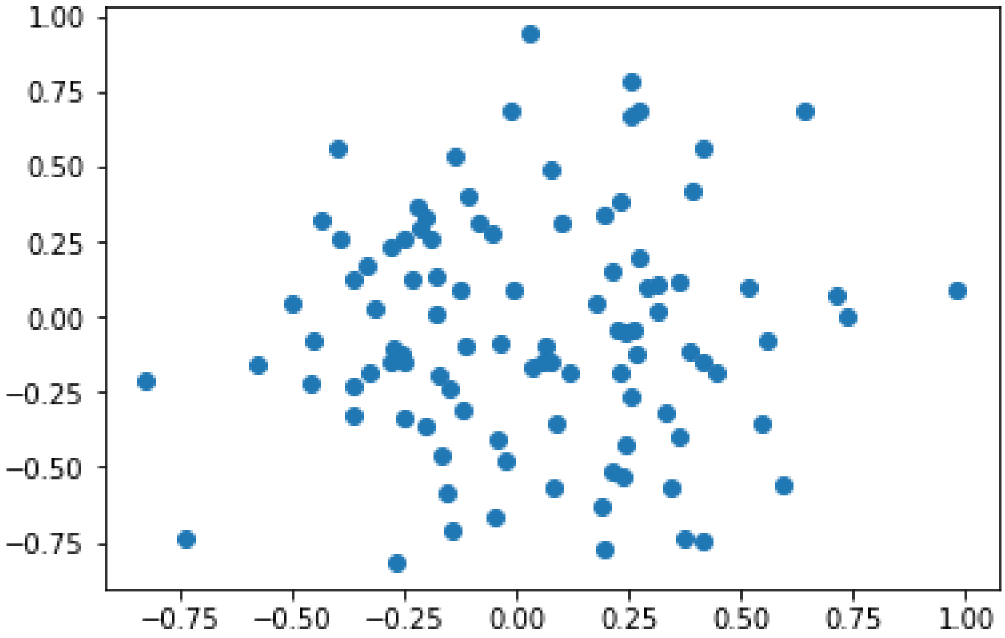
```
            zt_0 = z_0

            zt_1 = z_1 - self.a * (z_0**2)

            return tf.concat([zt_0, zt_1],axis=-1)

        def _forward_log_det_jacobian(self,z):

            return tf.constant([0.0],dtype='float32')

        def _inverse_log_det_jacobian(self,z):

            return -self._forward_log_det_jacobian(self._forward(z))
```

In [6]:
```
# build final distribution

def get_transformed_distribution(base_distribution,theta,a):


    # use tfb.Blockwise to apply bijection by components
    f1 = tfb.Blockwise(bijectors=[tfb.Identity(),tfb.Shift(-2)])

    f2 = tfb.Blockwise(bijectors=[tfb.Identity(),tfb.Scale(0.5)])

    # get instance of bijection
    f3 = F3(a)

    # for rotation use a lineal operator
    scale = tf.linalg.LinearOperatorFullMatrix([[np.cos(theta).astype('float32'), -np.s
                                                [np.sin(theta).astype('float32'), np.co
    f4 = tfb.ScaleMatvecLinearOperator(scale)

    f5 = tfb.Tanh()

    # get the final transormed distribution

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

    return tfd.TransformedDistribution(base_distribution, bijector)
```

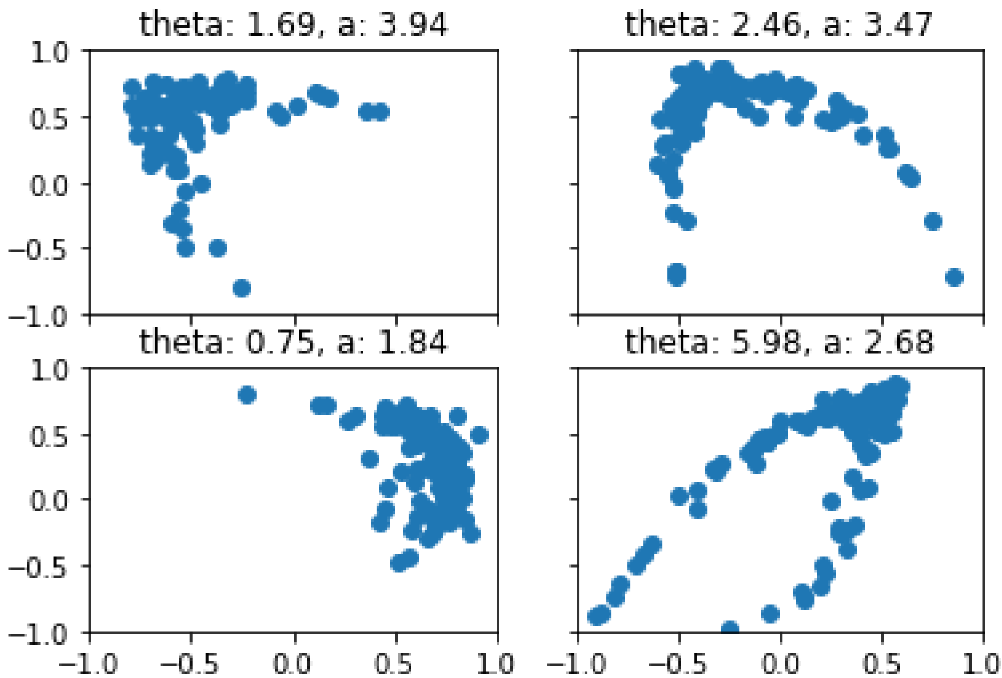
In [5]:
```
# show an scatterplot from the base distribution

num_samples = 100
samples_base = base_distribution.sample(num_samples)
samples_base_np = np.array(samples_base)
```

```
plt.figure()
plt.scatter(samples_base_np[:,0],samples_base_np[:,1])
```

Out[5]: <matplotlib.collections.PathCollection at 0x7f02d9052128>



In [7]:
```
# show scatter plots of the transformed distribution for 4 sets of randomly picked para

num_samples = 100
fig, ax = plt.subplots(2, 2)

for i in range(2):
    for j in range(2):

        theta = tfd.Uniform(0,2*np.pi).sample()
        a = tfd.Normal(3,1).sample()

        transformed_ditribution = get_transformed_distribution(base_distribution,a,thet

        samples = transformed_ditribution.sample(num_samples)
        samples_np = np.array(samples)

        ax[i, j].scatter(samples_np[:,0], samples_np[:,1])
        ax[i, j].set_title(f'theta: {theta:.2f}, a: {a:.2f}')
        ax[i, j].set(xlim=(-1, 1), ylim=(-1, 1))
        ax[i, j].label_outer()
```



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.

```
In [8]: # 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 ma
    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 [9]:
```
# 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 ima
    """
    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
```

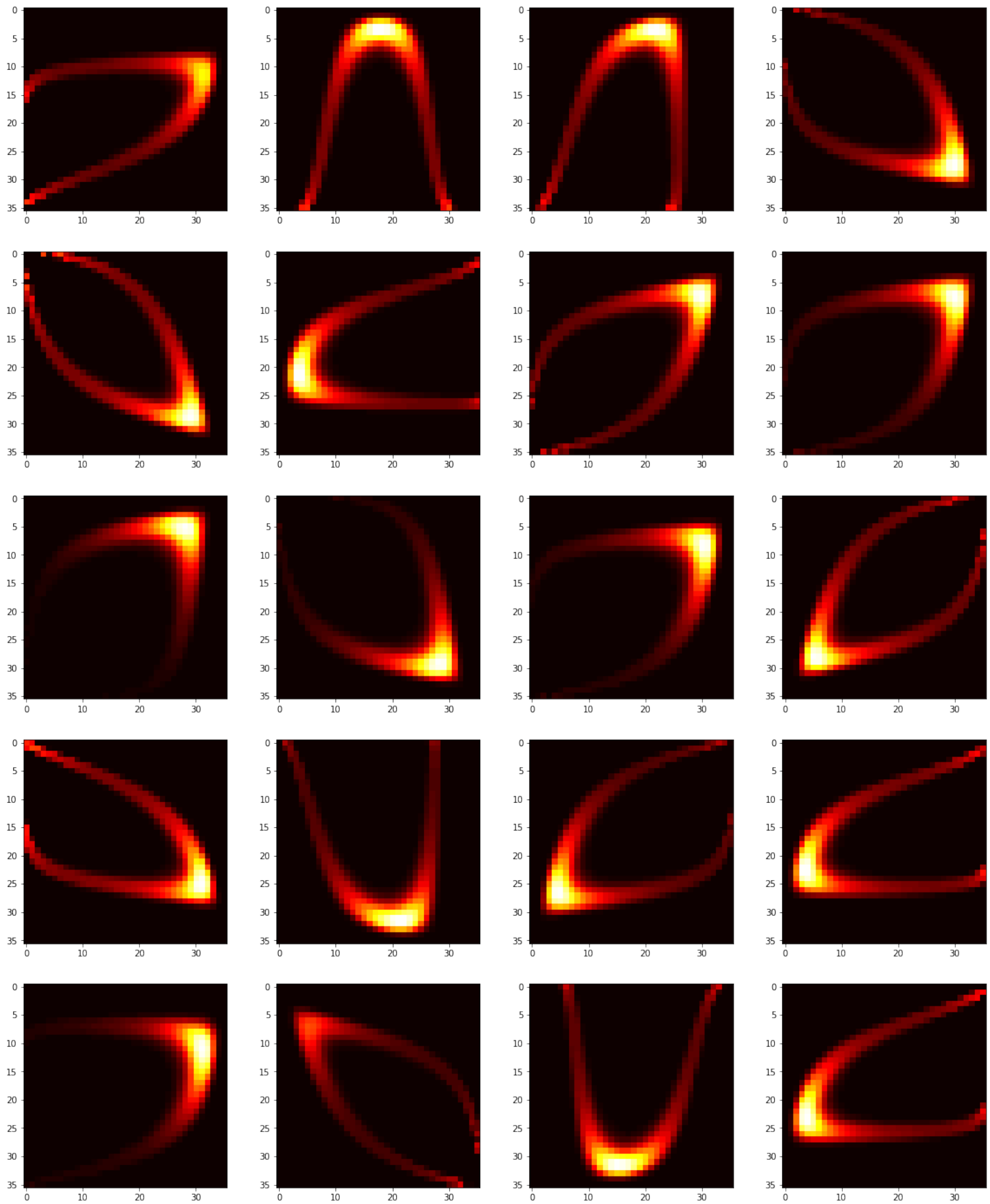
In [10]:
```
# create batched base distribution for training images
mean = np.zeros([1,dim])
sigma = .6
scale = np.ones_like(mean) * sigma**2

base_distribution_batched = tfd.MultivariateNormalDiag(loc=mean.astype('float32'),scal
```

In [ ]:
```
# display 4 images
```

```python
# notice that tensorflow will give some warnings that I ignore
fig, ax = plt.subplots(1, 4,figsize=(20,5))

for i in range(4):

    theta = tfd.Uniform(0,2*np.pi).sample()
    a = tfd.Normal(3,1).sample()

    dist = get_transformed_distribution(base_distribution_batched,theta,a)

    Z = get_densities(dist)

    ax[i].contourf(X, Y, Z[0, ...], cmap='hot', levels=100)
```

In [ ]:
```python
# create dataset (1000 images)
# again, tensorflow will give some warnings that I ignore
dataset = []
N = 1000

for i in range(N):

    theta = tfd.Uniform(0,2*np.pi).sample()
    a = tfd.Normal(3,1).sample()

    dist = get_transformed_distribution(base_distribution_batched,theta,a)

    Z = get_densities(dist)

    img = get_image_array_from_density_values(Z[0,...])

    dataset.append(img)

dataset = np.array(dataset)
```

In [15]:
```python
# display 20 images from my dataset
rows = 5
cols = 4

fig, ax = plt.subplots(rows, cols,figsize=(20,25))

import random
indexes = [i for i in range(dataset.shape[0])]
random.shuffle(indexes)

for i in range(rows):
    for j in range(cols):
        idx = indexes[j*rows+i]
        ax[i,j].imshow(dataset[idx,...])
```



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 [16]: train_fraction = 0.7
         num = dataset.shape[0]

         ds_train = tf.data.Dataset.from_tensor_slices(dataset[:int(train_fraction*num),...])
         ds_train = ds_train.map(lambda x: x/255)
         ds_train = ds_train.map(lambda x: (x,x))
         ds_train = ds_train.shuffle(10000)
         ds_train = ds_train.batch(20, drop_remainder=True)
         print(ds_train.element_spec)
```

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

```
In [17]: ds_val = tf.data.Dataset.from_tensor_slices(dataset[int(train_fraction*num):,...])
         ds_val = ds_val.map(lambda x: x/255)
         ds_val = ds_val.map(lambda x: (x,x))
         ds_val = ds_val.shuffle(10000)
         ds_val = ds_val.batch(20, drop_remainder=True)
         print(ds_train.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 [18]: # import tensorflow classes and layers
         from tensorflow.keras import Sequential, Model
         from tensorflow.keras.layers import (InputLayer, Dense, Flatten, Reshape, Concatenate
                                              Conv2DTranspose ,UpSampling2D, BatchNormalizatio


         tfd = tfp.distributions
```

```
tfb = tfp.bijectors
tfpl = tfp.layers
```

In [19]:
```
# define prior
def get_prior(latent_dim):
    """
    This function should create an instance of a MixtureSameFamily distribution
    according to the above specification.
    The function takes the num_modes and latent_dim as arguments, which should
    be used to define the distribution.
    Your function should then return the distribution instance.
    """

    prior = tfd.Independent(tfd.Normal(loc=tf.zeros(latent_dim), scale=1.),
                            reinterpreted_batch_ndims=1)

    return prior
```

In [20]:
```
def get_kl_regularizer(prior_distribution):
    """
    This function should create an instance of the KLDivergenceRegularizer
    according to the above specification.
    The function takes the prior_distribution, which should be used to define
    the distribution.
    Your function should then return the KLDivergenceRegularizer instance.
    """

    return tfpl.KLDivergenceRegularizer(prior_distribution,
                                        test_points_fn=lambda q: q.sample(5),
                                        test_points_reduce_axis=0)
```

In [22]:
```
# define encoder
def get_encoder(kl_regularizer,latent_dim):

    model = Sequential()

    model.add(InputLayer(input_shape=(36,36,3)))

    model.add(Conv2D(filters=32, kernel_size=3, strides=(2,2),
                                 padding='valid', activation='relu')) # 32

    model.add(Conv2D(filters=64, kernel_size=3, strides=(2,2),
                                 padding='valid', activation='relu')) # 64

    model.add(Flatten())

    model.add(Dense(tfpl.IndependentNormal.params_size(latent_dim),
                               activation=None))
```

```
    model.add(tfpl.IndependentNormal(latent_dim,
            convert_to_tensor_fn=tfd.Distribution.sample,
            activity_regularizer=tfpl.KLDivergenceRegularizer(prior, weight=0.1)))


    return model
```

In [23]:
```
# set parameters for encoder
latent_dim = 10
prior = get_prior(latent_dim)
kl_regularizer = get_kl_regularizer(prior)

encoder = get_encoder(kl_regularizer,latent_dim)
encoder.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 17, 17, 32)        896
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 8, 8, 64)          18496
_________________________________________________________________
flatten (Flatten)            (None, 4096)              0
_________________________________________________________________
dense (Dense)                (None, 20)                81940
_________________________________________________________________
independent_normal (Independ ((None, 10), (None, 10))  0
=================================================================
Total params: 101,332
Trainable params: 101,332
Non-trainable params: 0
_________________________________________________________________
```

In [24]:
```
# define decoder
def get_decoder(latent_dim):

    model = Sequential()

    model.add(InputLayer(input_shape=(latent_dim,)))

    model.add(Dense(9*9*32, activation=None))

    model.add(Reshape((9,9,32)))
```

```
        model.add(Conv2DTranspose(filters=64, kernel_size=3, strides=2,
                                  padding='same', activation='relu')) # 64

        model.add(Conv2DTranspose(filters=32, kernel_size=3, strides=2,
                                  padding='same', activation='relu')) # 32


        model.add(Conv2DTranspose(filters=3, kernel_size=3, strides=1,
                                  padding='same'))


        model.add(Flatten())

        model.add(tfpl.IndependentBernoulli((36,36,3)))

        return model
```

In [25]:
```
# set decoder
decoder = get_decoder(latent_dim)
decoder.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 2592)              28512
_________________________________________________________________
reshape (Reshape)            (None, 9, 9, 32)          0
_________________________________________________________________
conv2d_transpose (Conv2DTran (None, 18, 18, 64)        18496
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 36, 36, 32)        18464
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 36, 36, 3)         867
_________________________________________________________________
flatten_1 (Flatten)          (None, 3888)              0
_________________________________________________________________
independent_bernoulli (Indep ((None, 36, 36, 3), (None 0
=================================================================
Total params: 66,339
Trainable params: 66,339
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 [26]:
```
# build 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, 17, 17, 32)        896
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 8, 8, 64)          18496
_________________________________________________________________
flatten (Flatten)            (None, 4096)              0
_________________________________________________________________
dense (Dense)                (None, 20)                81940
_________________________________________________________________
independent_normal (Independ ((None, 10), (None, 10))  0
_________________________________________________________________
sequential_1 (Sequential)    (None, 36, 36, 3)         66339
=================================================================
Total params: 167,671
Trainable params: 167,671
Non-trainable params: 0
_________________________________________________________________
```

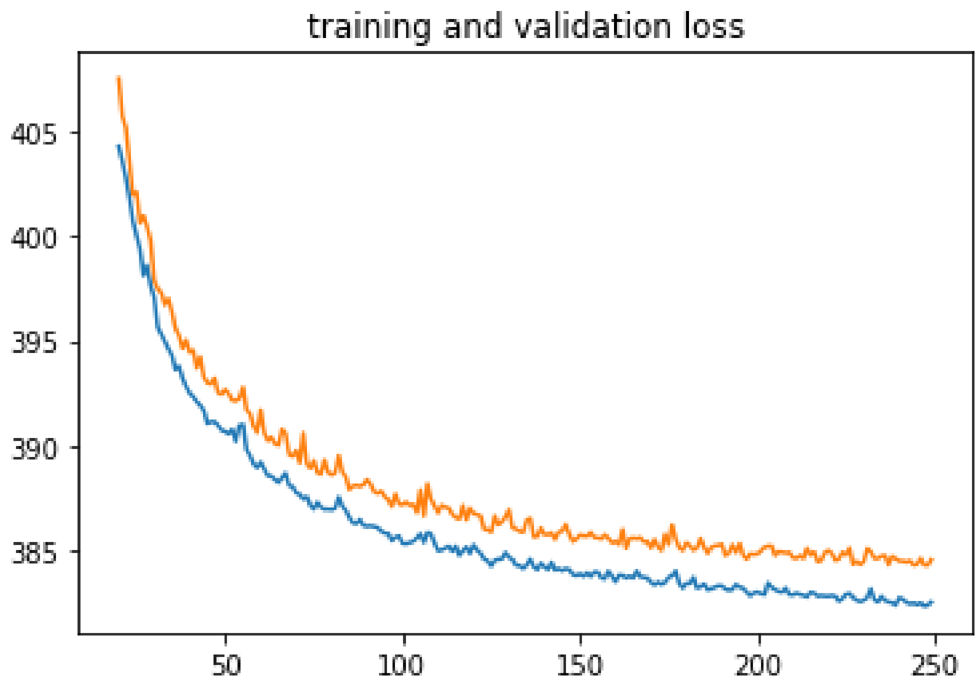
In [27]:
```
def reconstruction_loss(batch_of_images, decoding_dist):
    """
    This function should compute and return the average expected reconstruction loss,
    as defined above.
    The function takes batch_of_images (Tensor containing a batch of input images to
    the encoder) and decoding_dist (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 -decoding_dist.log_prob(batch_of_images)
```

In [28]:
```
# set optimizer and compile
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
vae.compile(optimizer=optimizer, loss=reconstruction_loss)
```

In [ ]:
```
# fit
history = vae.fit(ds_train,validation_data=ds_val, epochs=250)
```

In [33]:
```
# plot learning curves

plt.figure()
plt.plot(history.epoch[20:],history.history['loss'][20:])
plt.plot(history.epoch[20:],history.history['val_loss'][20:])
plt.title('training and validation loss')
```

Out[33]:
```
Text(0.5, 1.0, 'training and validation loss')
```



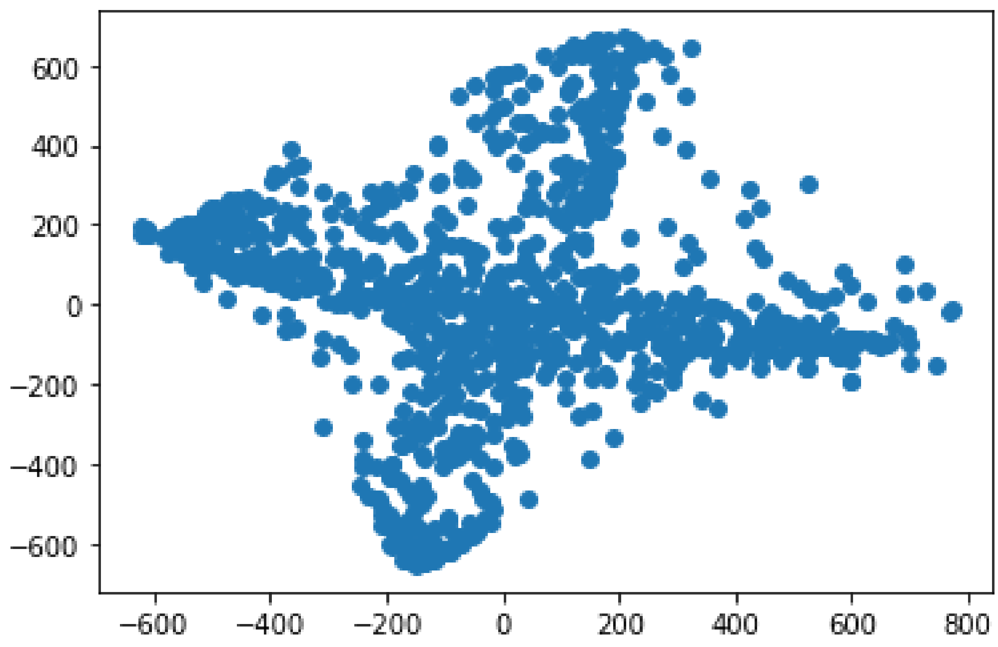
1.6 6. Use the encoder and decoder networks

- 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 [51]:
```
# pass all images from my dataset through encoder
encoded_imgs = encoder(dataset.astype('float32'))
```

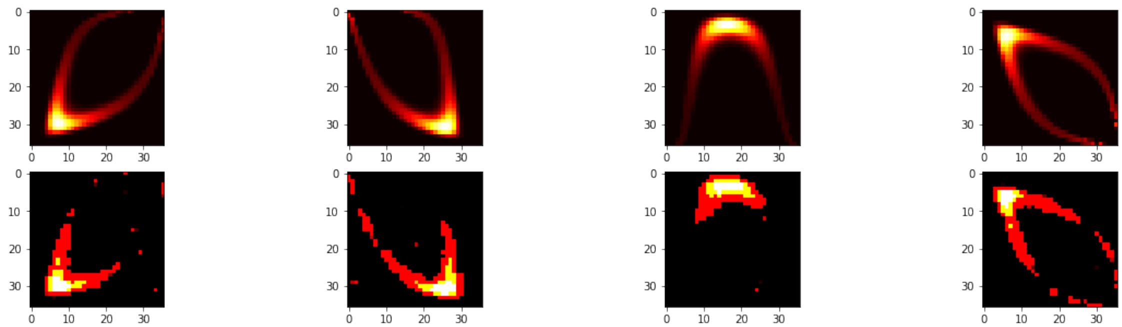
```
# display embeddings in scatter plot
# since is required to display the latent space in 2D whatever the latent space dimen
# then PCA can be used for dimensionality reduction

from sklearn.decomposition import PCA

# transform into numpy array
encoded_imgs = np.array(encoded_imgs[...])

# initialize an fit a PCA transorm
pca = PCA(n_components=2)
pca.fit(encoded_imgs)

pca_encoded_imgs = pca.transform(encoded_imgs)

plt.figure()
plt.scatter(pca_encoded_imgs[:,0],pca_encoded_imgs[:,1])
```

Out[51]: <matplotlib.collections.PathCollection at 0x7f00bc092828>



In [63]:
```
# sample 4 images, display original and reconstructed by the autoencoder (use mean)

fig, ax = plt.subplots(2, 4,figsize=(20,5))

indexes = [i for i in range(dataset.shape[0])]
```

```
random.shuffle(indexes)


for i in range(cols):

    idx = indexes[i]

    img = dataset[idx]
    img_r = vae(img[None,...].astype('float32'))

    ax[0,i].imshow(img)
    ax[1,i].imshow(np.squeeze(img_r.mean().numpy()))
```



1.7 Make a video of latent space interpolation (not assessed)

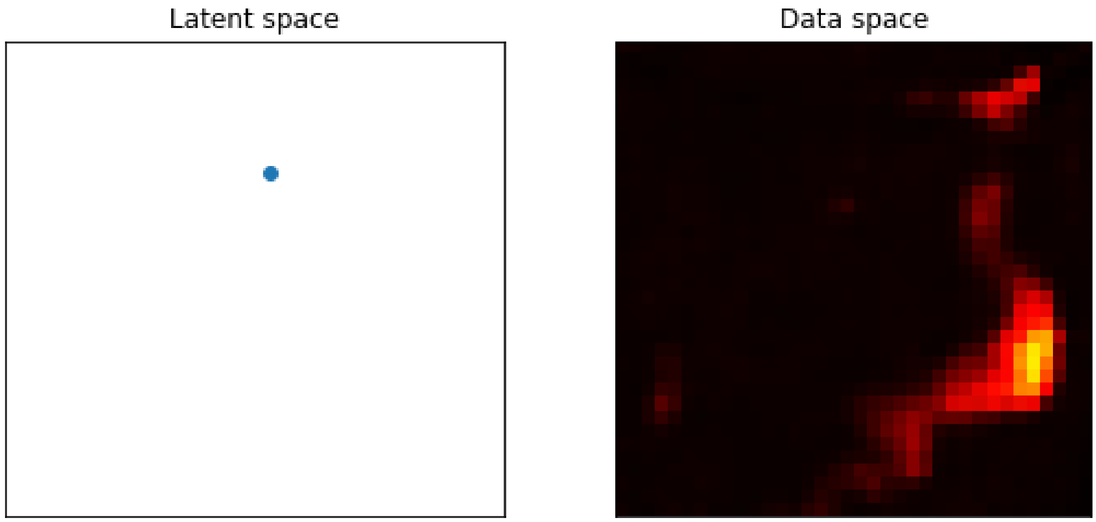
- Just for fun, you can run the code below to create a video of your decoder's generations, depending on the latent space.

In [56]:
```
# 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")
```

```python
    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 [57]:
```python
# Create the animation
latent_size = latent_dim
a = get_animation(latent_size, decoder, interpolation_length=200)
HTML(a.to_html5_video())
```

Out[57]: <IPython.core.display.HTML object>



```
In [ ]:
```