Capstone Project

January 3, 2023

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.

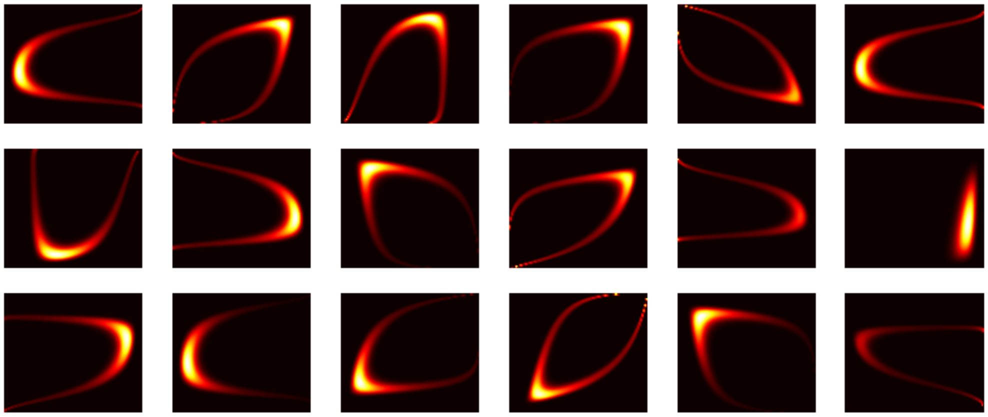
1.1.2 How to submit

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 [49]: 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
```



Flags overview image

```
from matplotlib import gridspec
%matplotlib inline

from tensorflow.keras import models, layers, callbacks
```

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

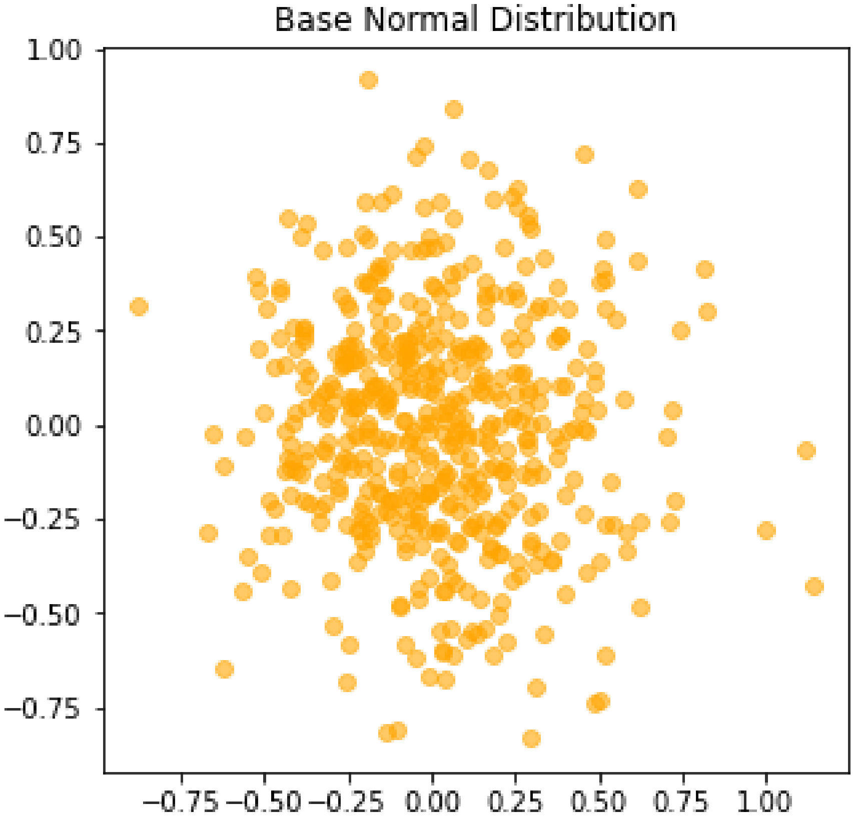
The complete normalising flow is given by the following chain of transformations: * $f_1(z) = (z_1, z_2 - 2)$, * $f_2(z) = (z_1, \frac{z_2}{2})$, * $f_3(z) = (z_1, z_2 + az_1^2)$, * $f_4(z) = Rz$, where R is a rotation matrix with angle θ, * $f_5(z) = \tanh(z)$, where the tanh function is applied elementwise.

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 [11]: 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])
```

```
In [12]: n = 500

         z = base_dist.sample(n).numpy().squeeze()
         plt.figure(figsize=(5,5))
         plt.scatter(z[:, 0], z[:, 1], color="orange", alpha=0.6)
         plt.title("Base Normal Distribution")
         plt.show()
```



```
In [13]: 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)
```

```
In [14]: 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)]], dtyp

             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)
```

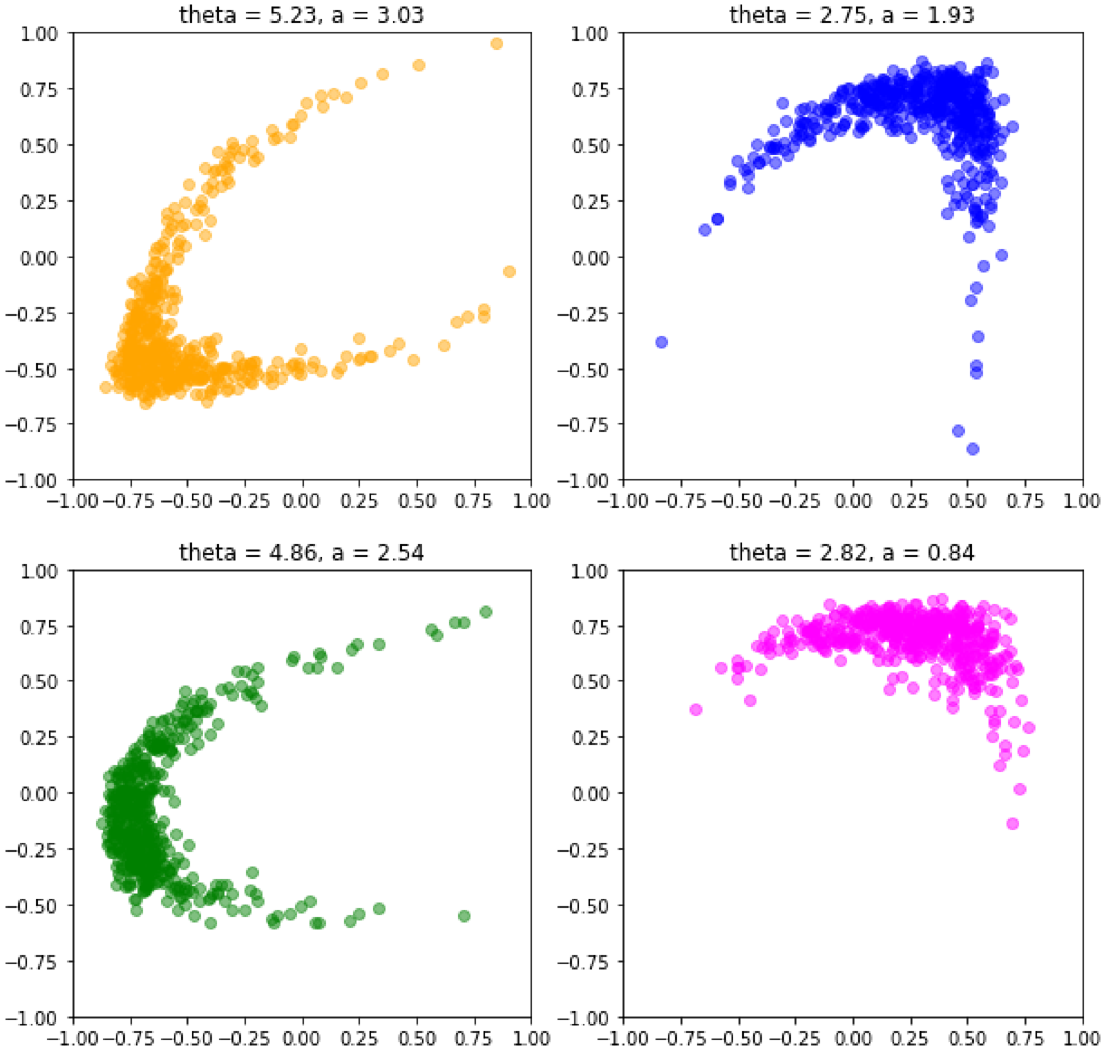
```
In [15]: def GetFlow(theta, a):
             f1 = tfb.Shift([0, -2])
             f2 = tfb.Scale([1, 0.5])
             f3 = Polynomial(a)
             f4 = Rotation(theta)
             f5 = tfb.Tanh()

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

```
In [16]: GetFlowDist = lambda theta, a, base_dist: tfd.TransformedDistribution(distribution=bas
                                                                               bijector=GetFlo
```

```
In [17]: def PlotFlow(theta, a, flow, n_samples, color="blue"):
             samples = flow.sample(n_samples).numpy().squeeze()
```

```
    plt.scatter(samples[:,0], samples[:, 1], color=color, alpha=0.5)
    plt.title("theta = {:.2f}, a = {:.2f}".format(theta, a))
    plt.xlim([-1,1])
    plt.ylim([-1,1])
```

```
In [18]: n = 500

         plt.figure(figsize = (10, 10))
         for i, col in enumerate(["orange", "blue", "green", "magenta"]):
             # Parameter Sampling
             theta = theta_dist.sample(1).numpy()[0]
             a = a_dist.sample(1).numpy()[0]
             # Building a Normalizing Flow Distrubtion
             flow_dist = GetFlowDist(theta, a, base_dist)
             # Plotting the Samples.
             plt.subplot(2, 2, i+1)
             PlotFlow(theta, a, flow_dist, n, col)
         plt.show()
```



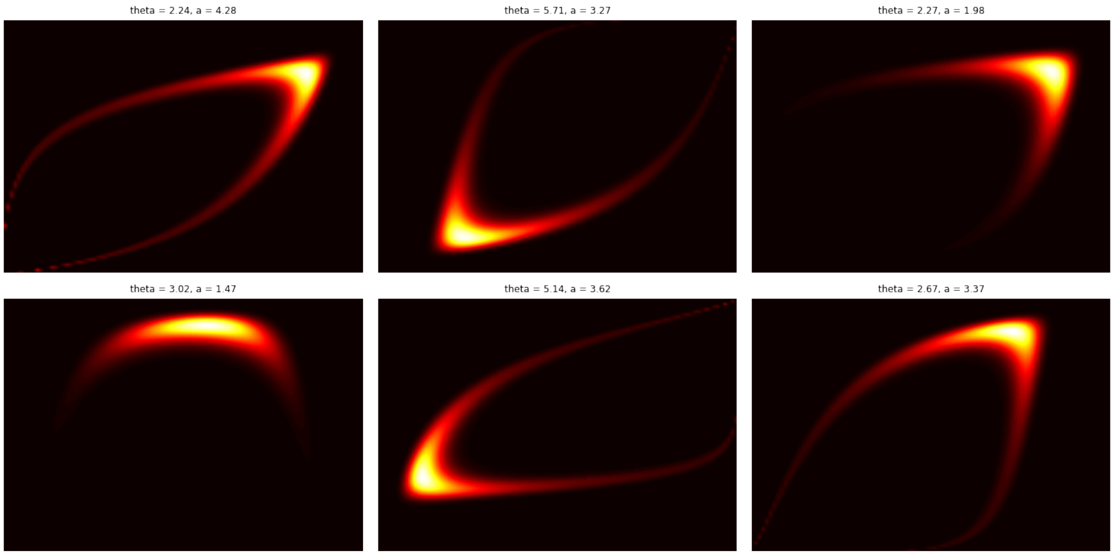
1.2 2. Create the image dataset

- 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.

In [19]:
```
# 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
```

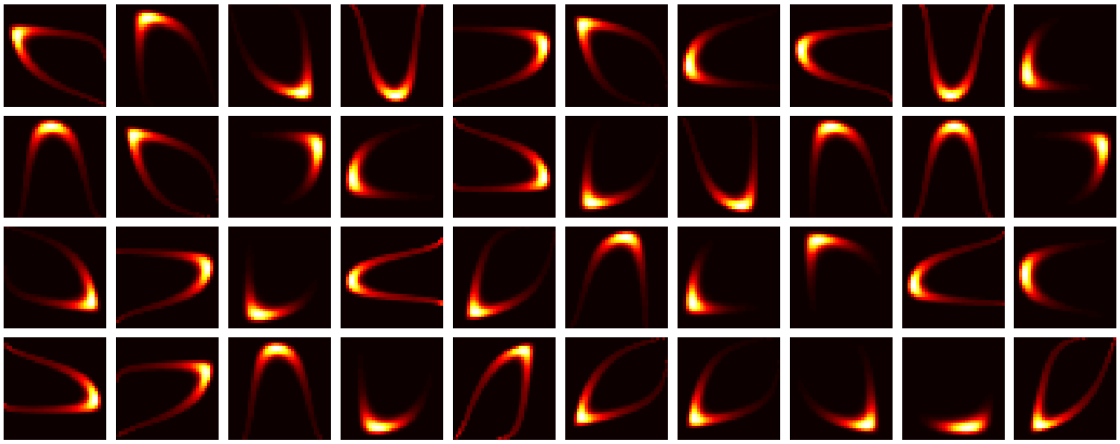
In [20]:
```
# Helper function to convert contour plots to numpy arrays

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] + (
            return image_from_plot

In [21]: plt.figure(figsize = (20, 10))
        for i in range(6):
            # Parameter Sampling
            theta = theta_dist.sample(1).numpy()[0]
            a = a_dist.sample(1).numpy()[0]
            # Building a Normalizing Flow Distrubtion
            flow_dist = GetFlowDist(theta, a, base_dist)
            flow_dist = tfd.BatchReshape(flow_dist, [1])
            # Contour Plot
            plt.subplot(2, 3, i+1)
            plt.contourf(X, Y, get_densities(flow_dist).squeeze(), cmap='hot', levels=100)
            plt.title("theta = {:.2f}, a = {:.2f}".format(theta, a))
            plt.axis('off')
        plt.tight_layout()
        plt.show()
```



In [22]:
```
images = []
img_params = []
N = 2500

for _ in range(N):
    # Parameter Sampling
    theta = theta_dist.sample(1).numpy()[0]
    a = a_dist.sample(1).numpy()[0]
    # Building a Normalizing Flow Distrubtion
    flow_dist = GetFlowDist(theta, a, base_dist)
    flow_dist = tfd.BatchReshape(flow_dist, [1])
    # Getting Density
    Z = get_densities(flow_dist).squeeze()
    #Saving Images
    images.append(get_image_array_from_density_values(Z))

images = np.array(images)
```

In [23]:
```
plt.figure(figsize=(20, 8))
for i in range(40):
    plt.subplot(4, 10, i+1)
    idx = np.random.randint(0, N)
    plt.imshow(images[idx])
    plt.axis("off")
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.

```
In [24]: def test_train_split(data, test_fraction):
             data = data.astype(dtype=np.float32)
             N = data.shape[0]
             test_idx = np.random.choice(np.arange(N), int(test_fraction*N), replace=False)
             train_idx = np.setdiff1d(np.arange(N), test_idx)

             return data[train_idx], data[test_idx]
```

```
In [25]: def GetDataset(data, test_fraction, batch_size=20):
             train, test = test_train_split(data, test_fraction)

             train = tf.data.Dataset.from_tensor_slices(train)
             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(test)
             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
```

```
In [26]: batch_size = 20
         train_data, test_data = GetDataset(images, 0.2)

         print(train_data.element_spec)
         print(test_data.element_spec)
```

```
WARNING:tensorflow:From c:\Python38\lib\site-packages\tensorflow\python\autograph\pyct\static_a
Instructions for updating:
Lambda fuctions will be no more assumed to be used in the statement where they are used, or at
(TensorSpec(shape=(20, 36, 36, 3), dtype=tf.float32, name=None), TensorSpec(shape=(20, 36, 36,
(TensorSpec(shape=(20, 36, 36, 3), dtype=tf.float32, name=None), TensorSpec(shape=(20, 36, 36,
```

```
In [ ]:

In [ ]:

In [ ]:
```

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 [27]: image_dims = images.shape[1:]
         image_dims
```

```
Out[27]: (36, 36, 3)
```

```
In [28]: latent_dim = 2
```

```
In [29]: def get_prior(latent_dim):
             prior = tfd.MultivariateNormalDiag(loc = tf.Variable(tf.zeros(latent_dim), dtype=
                                                scale_diag = tfp.util.TransformedVariable(initi
                                                                                          bije
                                                                                          dtype
                                                )

             return prior
```

```
In [30]: def get_encoder(latent_dim):
             prior = get_prior(latent_dim)

             model = models.Sequential([
                                         layers.InputLayer(input_shape=image_dims),

                                         layers.Conv2D(filters=32, kernel_size=(3,3)),
                                         layers.BatchNormalization(),
                                         layers.LeakyReLU(0.2),

                                         layers.Conv2D(filters=64, kernel_size=(3,3)),
                                         layers.BatchNormalization(),
                                         layers.LeakyReLU(0.2),

                                         layers.Conv2D(filters=128, kernel_size=(3,3)),
                                         layers.BatchNormalization(),
                                         layers.LeakyReLU(0.2),

                                         layers.Conv2D(filters=256, kernel_size=(3,3)),
                                         layers.BatchNormalization(),
                                         layers.LeakyReLU(0.2),

                                         layers.Conv2D(filters=8, kernel_size=(1,1)),
                                         layers.BatchNormalization(),

                                         layers.Flatten(),

                                         layers.Dense(100),
                                         layers.BatchNormalization(),
                                         layers.LeakyReLU(0.2),

                                         layers.Dense(tfpl.MultivariateNormalTriL.params_size(la
                                         tfpl.MultivariateNormalTriL(latent_dim),

                                         tfpl.KLDivergenceAddLoss(prior,
                                                                  use_exact_kl = False,
                                                                  test_points_fn = lambda q:q.sa
                                                                  test_points_reduce_axis=(0,1)
             ])

             return model

In [31]: enc_model = get_encoder(latent_dim)

         enc_model.summary()

Model: "sequential"
_________________________________________________________________
```

```
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 34, 34, 32)        896

 batch_normalization (BatchN  (None, 34, 34, 32)       128
 ormalization)

 leaky_re_lu (LeakyReLU)     (None, 34, 34, 32)        0

 conv2d_1 (Conv2D)           (None, 32, 32, 64)        18496

 batch_normalization_1 (Batc  (None, 32, 32, 64)       256
 hNormalization)

 leaky_re_lu_1 (LeakyReLU)   (None, 32, 32, 64)        0

 conv2d_2 (Conv2D)           (None, 30, 30, 128)       73856

 batch_normalization_2 (Batc  (None, 30, 30, 128)      512
 hNormalization)

 leaky_re_lu_2 (LeakyReLU)   (None, 30, 30, 128)       0

 conv2d_3 (Conv2D)           (None, 28, 28, 256)       295168

 batch_normalization_3 (Batc  (None, 28, 28, 256)      1024
 hNormalization)

 leaky_re_lu_3 (LeakyReLU)   (None, 28, 28, 256)       0

 conv2d_4 (Conv2D)           (None, 28, 28, 8)         2056

 batch_normalization_4 (Batc  (None, 28, 28, 8)        32
 hNormalization)

 flatten (Flatten)           (None, 6272)              0

 dense (Dense)               (None, 100)               627300

 batch_normalization_5 (Batc  (None, 100)              400
 hNormalization)

 leaky_re_lu_4 (LeakyReLU)   (None, 100)               0

 dense_1 (Dense)             (None, 5)                 505

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

```
 kl_divergence_add_loss (KLD  (None, 2)                4
 ivergenceAddLoss)

=================================================================
Total params: 1,020,633
Trainable params: 1,019,457
Non-trainable params: 1,176
_________________________________________________________________
```

```
In [32]: def get_decoder(latent_dim, image_dim):
             model = models.Sequential([
                                         layers.InputLayer(input_shape=(latent_dim,)),

                                         layers.Dense(64),
                                         layers.BatchNormalization(),
                                         layers.LeakyReLU(0.2),

                                         layers.Dense(128),
                                         layers.BatchNormalization(),
                                         layers.LeakyReLU(0.2),

                                         layers.Dense(256),
                                         layers.BatchNormalization(),
                                         layers.LeakyReLU(0.2),

                                         layers.Reshape(target_shape=(8, 8, 4)),

                                         layers.UpSampling2D(size=(3,3)),
                                         layers.Conv2D(filters=128, kernel_size=(3,3)),
                                         layers.BatchNormalization(),
                                         layers.LeakyReLU(0.2),

                                         layers.UpSampling2D(size=(2, 2)),
                                         layers.Conv2D(filters=64, kernel_size=(3,3)),
                                         layers.BatchNormalization(),
                                         layers.LeakyReLU(0.2),

                                         layers.Conv2D(filters=32, kernel_size=(3,3)),
                                         layers.BatchNormalization(),
                                         layers.LeakyReLU(0.2),

                                         layers.Conv2D(filters=16, kernel_size=(3,3)),
                                         layers.BatchNormalization(),
                                         layers.LeakyReLU(0.2),

                                         layers.Conv2D(filters=1, kernel_size=(3, 3), strides=(
```

```
                                        layers.BatchNormalization(),
                                        layers.LeakyReLU(0.2),

                                        layers.Flatten(),

                                        layers.Dense(tfpl.IndependentBernoulli.params_size(imag

                                        tfpl.IndependentBernoulli(event_shape=image_dim)
                    ])

                    return model
```

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 [34]: nll = lambda real, estimated: -tf.reduce_mean(estimated.log_prob(real))
```

```
In [35]: def get_vae(encoder, decoder):
             vae = models.Model(inputs = encoder.inputs,
                                outputs= decoder(encoder.outputs))

             vae.compile(optimizer="adam", loss=nll)

             return vae
```

```
In [36]: vae_model = get_vae(enc_model, dec_model)
         vae_model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 36, 36, 3)]       0

 conv2d (Conv2D)             (None, 34, 34, 32)        896

 batch_normalization (BatchN  (None, 34, 34, 32)       128
 ormalization)

 leaky_re_lu (LeakyReLU)     (None, 34, 34, 32)        0

 conv2d_1 (Conv2D)           (None, 32, 32, 64)        18496

 batch_normalization_1 (Batc  (None, 32, 32, 64)       256
```

```
 hNormalization)

 leaky_re_lu_1 (LeakyReLU)   (None, 32, 32, 64)        0

 conv2d_2 (Conv2D)           (None, 30, 30, 128)       73856

 batch_normalization_2 (Batc  (None, 30, 30, 128)      512
 hNormalization)

 leaky_re_lu_2 (LeakyReLU)   (None, 30, 30, 128)       0

 conv2d_3 (Conv2D)           (None, 28, 28, 256)       295168

 batch_normalization_3 (Batc  (None, 28, 28, 256)      1024
 hNormalization)

 leaky_re_lu_3 (LeakyReLU)   (None, 28, 28, 256)       0

 conv2d_4 (Conv2D)           (None, 28, 28, 8)         2056

 batch_normalization_4 (Batc  (None, 28, 28, 8)        32
 hNormalization)

 flatten (Flatten)           (None, 6272)              0

 dense (Dense)               (None, 100)               627300

 batch_normalization_5 (Batc  (None, 100)              400
 hNormalization)

 leaky_re_lu_4 (LeakyReLU)   (None, 100)               0

 dense_1 (Dense)             (None, 5)                 505

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

 kl_divergence_add_loss (KLD  (None, 2)                4
 ivergenceAddLoss)

 sequential_1 (Sequential)   (None, 36, 36, 3)         1409653

=================================================================
Total params: 2,430,286
Trainable params: 2,427,732
Non-trainable params: 2,554
_________________________________________________________________
```

```
In [38]: es_callback = callbacks.EarlyStopping(monitor="val_loss",
                                               min_delta=0.1,
                                               patience=8,
                                               restore_best_weights=True)


         history = vae_model.fit(train_data,
                                 validation_data=test_data,
                                 epochs=30,
                                 callbacks=[es_callback])
```

```
Epoch 1/30
100/100 [==============================] - 68s 680ms/step - loss: 383.6437 - val_loss: 379.4981
Epoch 2/30
100/100 [==============================] - 66s 656ms/step - loss: 379.1909 - val_loss: 384.6248
Epoch 3/30
100/100 [==============================] - 67s 673ms/step - loss: 378.6607 - val_loss: 383.0833
Epoch 4/30
100/100 [==============================] - 65s 651ms/step - loss: 378.7240 - val_loss: 379.6505
Epoch 5/30
100/100 [==============================] - 64s 645ms/step - loss: 376.7932 - val_loss: 377.6109
Epoch 6/30
100/100 [==============================] - 66s 658ms/step - loss: 384.6761 - val_loss: 391.2424
Epoch 7/30
100/100 [==============================] - 65s 654ms/step - loss: 383.2501 - val_loss: 393.8499
Epoch 8/30
100/100 [==============================] - 63s 634ms/step - loss: 386.9408 - val_loss: 445.8101
Epoch 9/30
100/100 [==============================] - 63s 634ms/step - loss: 380.3320 - val_loss: 381.6845
Epoch 10/30
100/100 [==============================] - 66s 661ms/step - loss: 379.3434 - val_loss: 383.1099
Epoch 11/30
100/100 [==============================] - 64s 638ms/step - loss: 378.7409 - val_loss: 375.5200
Epoch 12/30
100/100 [==============================] - 63s 635ms/step - loss: 377.8260 - val_loss: 384.5381
Epoch 13/30
100/100 [==============================] - 64s 638ms/step - loss: 377.2528 - val_loss: 381.3002
Epoch 14/30
100/100 [==============================] - 63s 630ms/step - loss: 377.0789 - val_loss: 390.2468
Epoch 15/30
100/100 [==============================] - 64s 640ms/step - loss: 378.9029 - val_loss: 378.6939
Epoch 16/30
100/100 [==============================] - 66s 664ms/step - loss: 375.5562 - val_loss: 378.9418
Epoch 17/30
100/100 [==============================] - 67s 671ms/step - loss: 377.6570 - val_loss: 377.2133
Epoch 18/30
100/100 [==============================] - 66s 663ms/step - loss: 379.8917 - val_loss: 393.4899
Epoch 19/30
```

```
100/100 [==============================] - 66s 665ms/step - loss: 375.5200 - val_loss: 376.830
```

```
In [39]: vae_model.save_weights('D:\\GitHub\\Coursera\\Tensorflow2forDeepLearning\\Programming
```

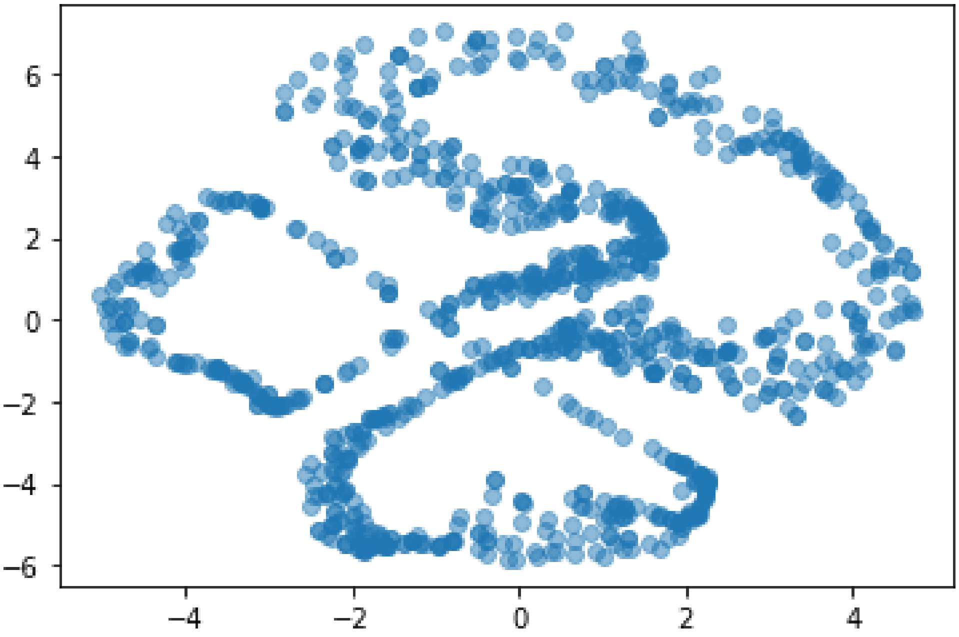
```
In [40]: plt.plot(history.history["loss"])
         plt.plot(history.history["val_loss"])
         plt.title("Loss Plots")
         plt.show()
```

1.6 6. Use the encoder and decoder networks

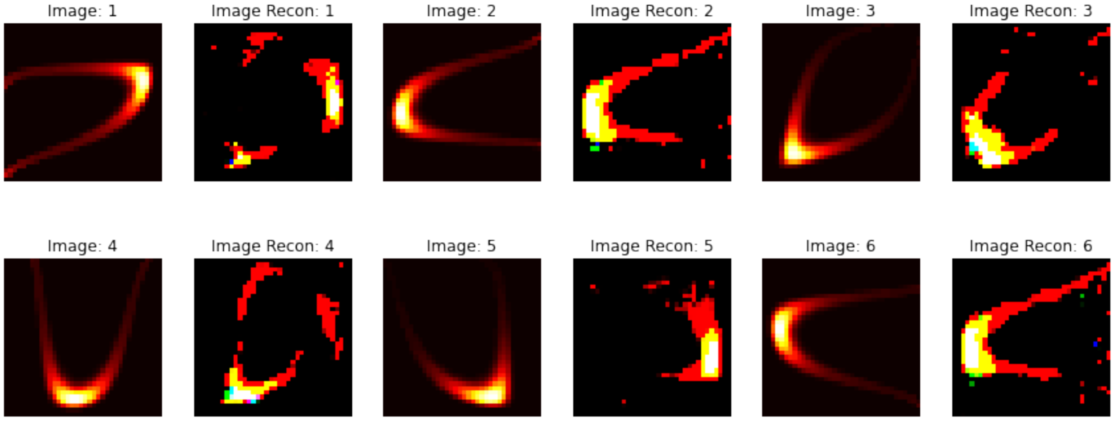
- You can now put your encoder and decoder networks into practice!
- Randomly sample 1000 images from the dataset, and pass them through the encoder. Display the embeddings in a scatter plot (project to 2 dimensions if the latent space has dimension higher than two).
- Randomly sample 4 images from the 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.

```
In [41]: N = 1000
         idx = np.random.choice(np.arange(images.shape[0]), N)

         embeddings = enc_model(images[idx]/255.0).mean()
```

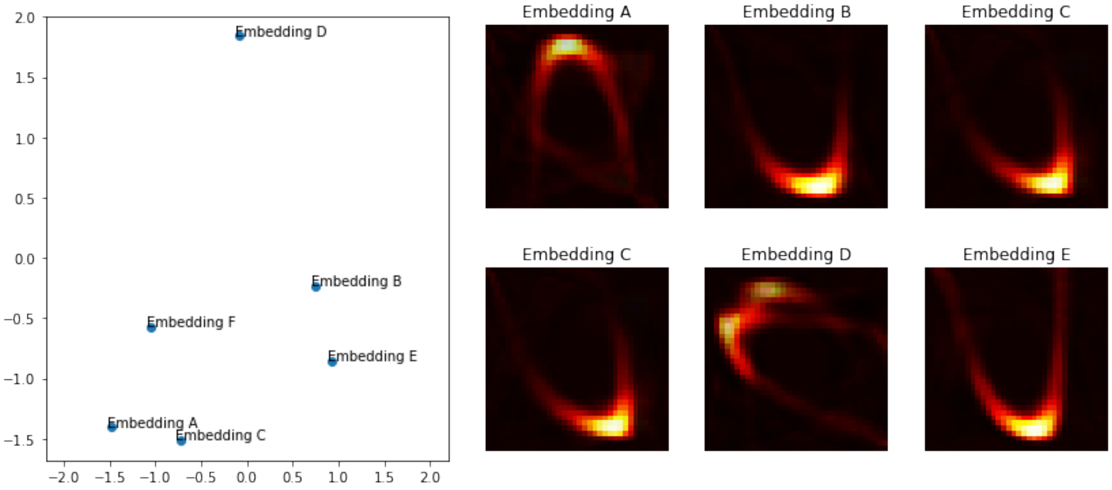
```
In [42]: plt.scatter(embeddings[:,0], embeddings[:,1], alpha=0.5)
         plt.show()
```



```
In [43]: N = 6
         idx = np.random.choice(np.arange(images.shape[0]), N)
         rec_images = vae_model(images[idx]).mean().numpy()

         plt.figure(figsize=(15, 6))
         for i in range(N):
             plt.subplot(2, 6, 2*i+1)
             plt.imshow(images[idx[i]])
             plt.title("Image: {}".format(i+1))
             plt.axis("off")

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



```
In [51]: N = 6
         embeddings = np.random.uniform(-2, 2, (N, latent_dim))
         rec_images = dec_model(embeddings).mean()


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

         ax0 = plt.subplot(gs[:, 0:2])
         ax0.scatter(embeddings[:, 0], embeddings[:, 1])
         for i in range(N):
             ax0.annotate("Embedding "+chr(ord("A")+i), (embeddings[i, 0]-0.05, embeddings[i, 
         ax0.set_xlim(-2.2, 2.2)

         for i in range(2):
             for j in range(3):
                 ax1 = plt.subplot(gs[i, 2+j])
                 ax1.imshow(rec_images[2*i + j])
                 ax1.set_axis_off()
                 ax1.set_title("Embedding "+chr(ord("A")+(2*i + j)))

         plt.show()
```



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 [52]:
```
# 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 plottin
    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 [53]:
```
# Create the animation

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

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

d:\GitHub\Coursera\Tensorflow2forDeepLearning\Programming Assignments\prob_models\Caps
  <a href='vscode-notebook-cell:/d%3A/GitHub/Coursera/Tensorflow2forDeepLearning/Progra
----> <a href='vscode-notebook-cell:/d%3A/GitHub/Coursera/Tensorflow2forDeepLearning/Progra
  <a href='vscode-notebook-cell:/d%3A/GitHub/Coursera/Tensorflow2forDeepLearning/Progra


NameError: name 'latent_size' is not defined
```

In [ ]: