Capstone Project

January 1, 2024

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.

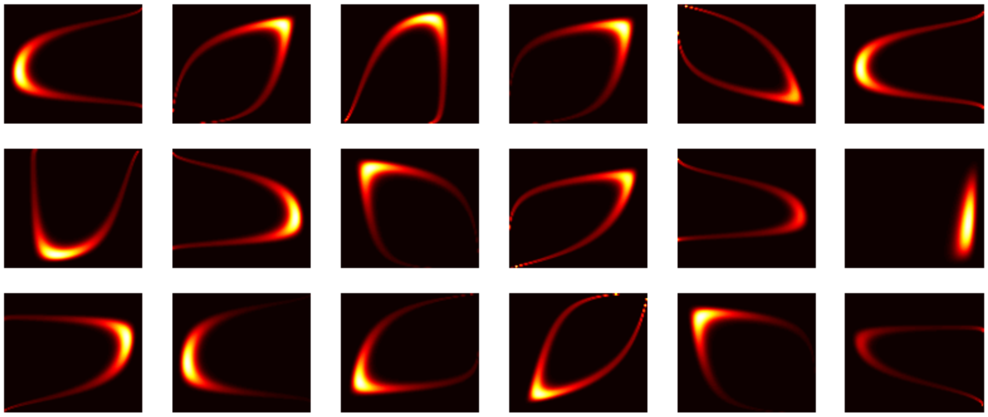
1.1.2 How to submit

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
```



Flags overview image

```
%matplotlib inline

from tqdm import tqdm

from tensorflow.keras.layers import InputLayer, Conv2D, Flatten, Dense, Reshape, BatchN
from tensorflow.keras.models import Sequential, Model
```

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

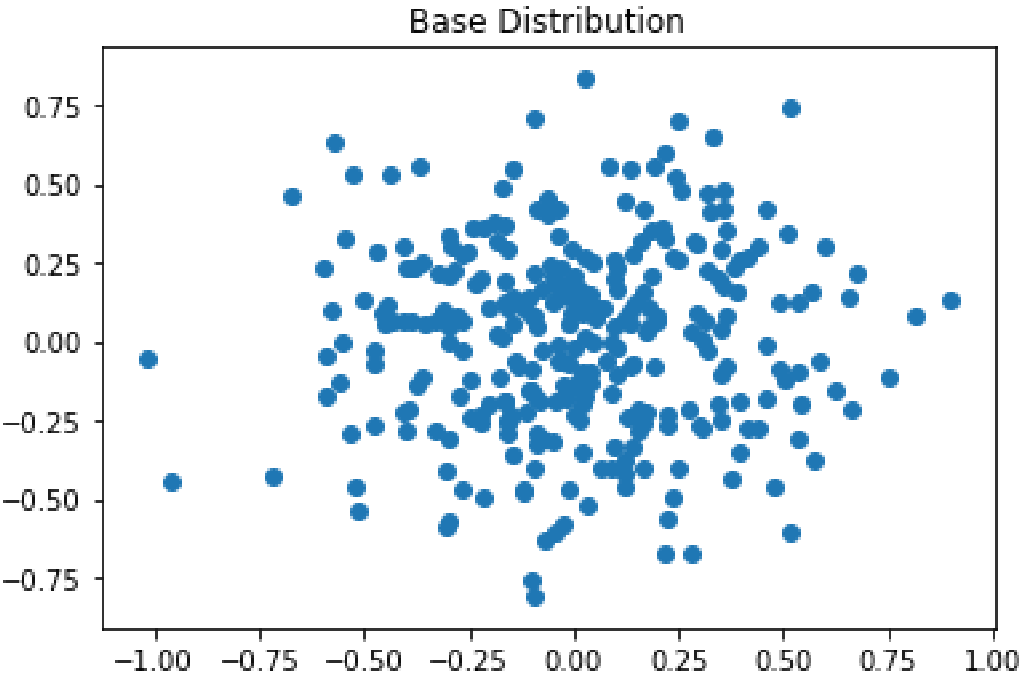
- To construct the image dataset, you will build a normalising flow to transform the 2-D Gaussian random variable $z = (z_1, z_2)$, which has mean $\mathbf{0}$ and covariance matrix $\Sigma = \sigma^2 \mathbf{I}_2$, with $\sigma = 0.3$.
- This normalising flow uses bijectors that are parameterised by 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]: # base distribution

        # To construct the image dataset, you will build a normalising flow to transform the 2
        # which has mean 0 and covariance matrix =2I2, with =0.3.
        mu, sigma = 0, 0.3
        base_dist = tfd.MultivariateNormalDiag(loc=[mu, mu], scale_diag=[sigma, sigma])

        # This normalising flow uses bijectors that are parameterised by the following random
        # * U[0,2)
        theta_dist = tfd.Uniform(low=0, high=2*np.pi)
        # * aN(3,1)
        a_dist = tfd.Normal(loc=3, scale=1)
```

```
In [3]: # Display a scatter plot of samples from the base distribution.

        dist_plot = base_dist.sample(300).numpy().squeeze()
        plt.figure()
        plt.scatter(dist_plot[:, 0], dist_plot[:, 1])
        plt.title("Base Distribution")
        plt.show()
```



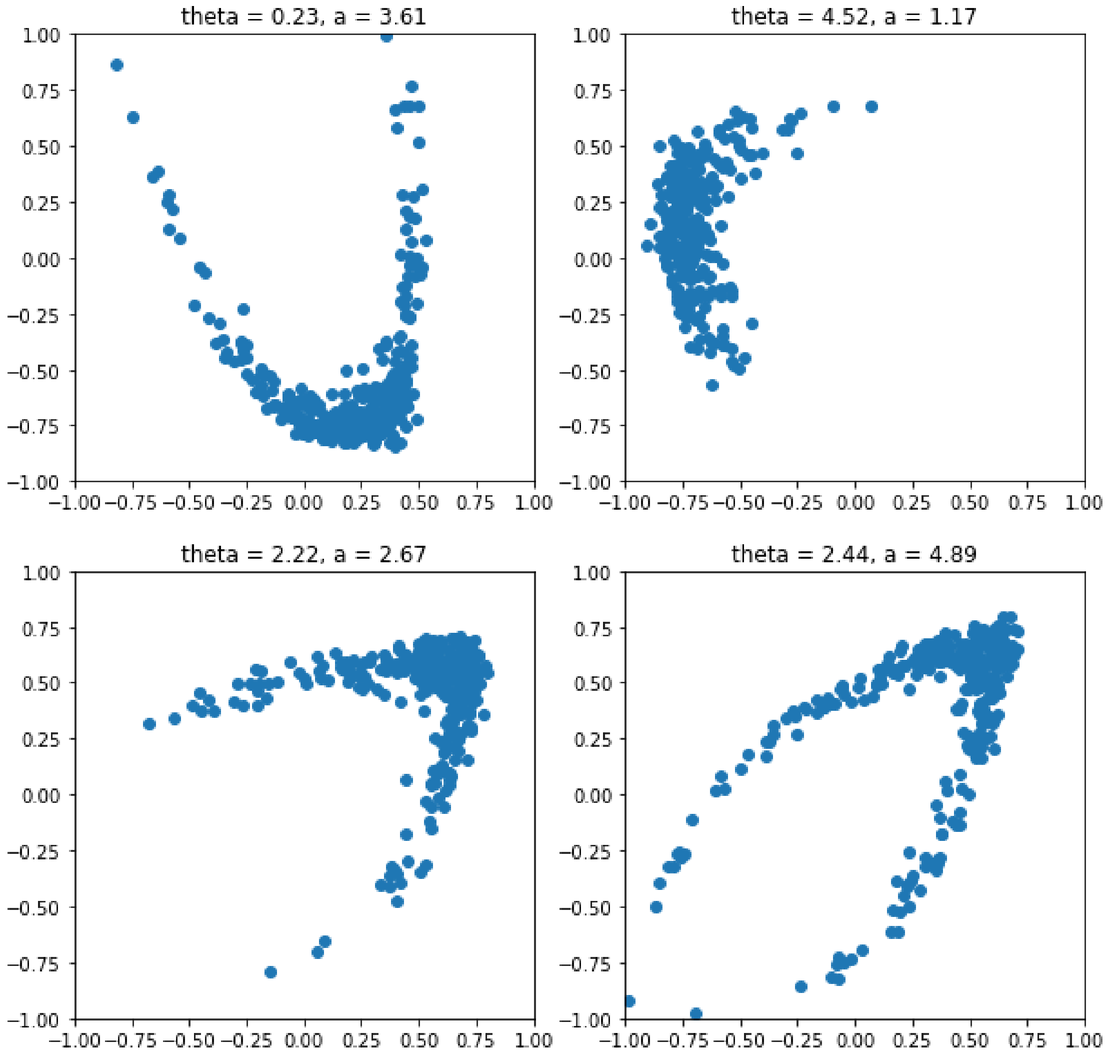
```
In [4]: # polynominal bijector (f3)

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

                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)

            # Ensure to implement the log_det_jacobian methods for any subclassed bijectors th
            def _forward_log_det_jacobian(self, x):
                return tf.constant(0., dtype=x.dtype)
```

```
In [5]: # rotation bijector (f4)

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

                self.rot_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, 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)

            # Ensure to implement the log_det_jacobian methods for any subclassed bijectors th
            def _forward_log_det_jacobian(self, x):
                return tf.constant(0., dtype=x.dtype)
```

```
In [6]: # chained bijectors
        def GetFlow(theta, a):
            # The complete normalising flow is given by the following chain of transformations
```

```
    # * f1(z)=(z1,z22),
    f1 = tfb.Shift([0, -2])
    # * f2(z)=(z1,z22),
    f2 = tfb.Scale([1, 0.5])
    # * f3(z)=(z1,z2+az21),
    f3 = Polynomial(a)
    # * f4(z)=Rz, where R is a rotation matrix with angle ,
    f4 = Rotation(theta)
    # * f5(z)=tanh(z), where the tanh function is applied elementwise.
    f5 = tfb.Tanh()

    # The transformed random variable x is given by x=f5(f4(f3(f2(f1(z))))).
    return tfb.Chain([f5, f4, f3, f2, f1])

# You should use or construct bijectors for each of the transformations fi, i=1,,5,
# and use tfb.Chain and tfb.TransformedDistribution to construct the final transformed
FlowDist = lambda theta, a, base_dist: tfd.TransformedDistribution(distribution=base_d
                                                                      bijector=GetFlow
```

```
In [7]: # Display 4 scatter plot images of the transformed distribution from your random norma
        # Fix the axes of these 4 plots to the range [1,1].

        plt.figure(figsize=(10, 10))
        for i in range(4):
            theta = theta_dist.sample().numpy()
            a = a_dist.sample().numpy()
            flow_dist = FlowDist(theta, a, base_dist)
            plt.subplot(2, 2, i+1)
            samples = flow_dist.sample(300).numpy().squeeze()
            plt.scatter(samples[:,0], samples[:, 1])
            plt.title("theta = {:.2f}, a = {:.2f}".format(theta, a))
            plt.xlim([-1,1])
            plt.ylim([-1,1])
        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.

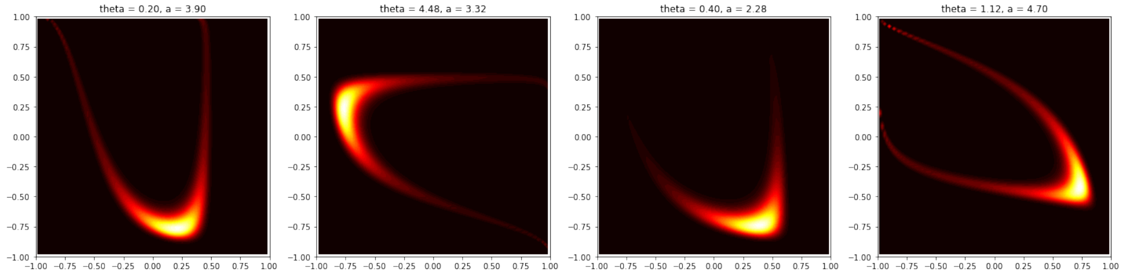
- As well as the `get_densities` function, the `get_image_array_from_density_values` function will help you to generate the dataset.
    - This function creates a numpy array for an image of the contour plot for a given set of density values Z. Feel free to choose your own options for the contour plots.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

```
In [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
```

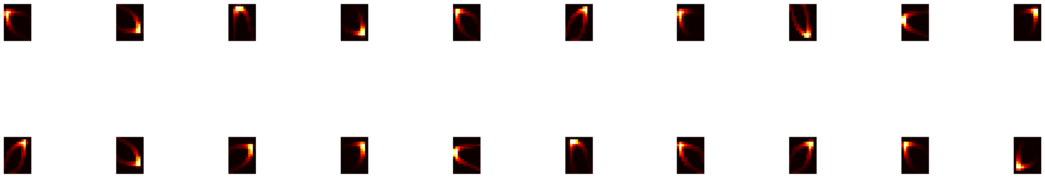
```
In [10]: # First, display a sample of 4 contour plot images from your normalising flow network

         plt.figure(figsize=(20,5))
         for i in range(4):
             theta = theta_dist.sample().numpy()
             a = a_dist.sample().numpy()
             flow_dist = FlowDist(theta, a, base_dist)
             flow_dist = tfd.BatchReshape(flow_dist, [1])
             plt.subplot(1, 4, i+1)
             # You may find the following get_densities function useful: this calculates densi
             plt.contourf(X, Y, get_densities(flow_dist).squeeze(), cmap='hot', levels=50)
             plt.title("theta = {:.2f}, a = {:.2f}".format(theta, a))
             plt.xlim([-1,1])
             plt.ylim([-1,1])
         plt.tight_layout()
         plt.show()
```



```
In [11]: # Your dataset should consist of at least 1000 images, stored in a numpy array of sha
         # Each image in the dataset should correspond to a contour plot of a transformed dist

         images = []
         img_params = []

         for _ in tqdm(range(1000)):
             theta = theta_dist.sample().numpy()
             a = a_dist.sample().numpy()
             flow_dist = FlowDist(theta, a, base_dist)
             flow_dist = tfd.BatchReshape(flow_dist, [1])
             # As well as the get_densities function, the get_image_array_from_density_values f
             densities = get_densities(flow_dist).squeeze()
             images.append(get_image_array_from_density_values(densities))

         images = np.array(images)
```

```
100%|| 1000/1000 [07:04<00:00,  2.36it/s]
```

```
In [12]: # Display a sample of 20 images from your generated dataset in a figure.

         plt.figure(figsize=(20,5))
         for i in range(20):
             plt.subplot(2, 10, i+1)
             plt.imshow(images[i])
             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 [13]: # You should now split your dataset to create tf.data.Dataset objects for training an

         dataset = tf.data.Dataset.from_tensor_slices(images)
         split_size = int(len(images)*0.75)
         print(split_size)

         train_set = dataset.take(split_size)
         val_set = dataset.skip(split_size)
```

```
750
```

```
In [14]: def prepare_data(dataset: tf.data.Dataset):

           # Using the map method, normalise the pixel values so that they lie between 0 and 1
             dataset = dataset.map(lambda x: tf.cast(x, tf.float32))
             dataset = dataset.map(lambda x: x/255.0)
           # These Datasets will be used to train a variational autoencoder (VAE). Use the map
             dataset = dataset.map(lambda x: (x,x))

             return dataset

         train_set = prepare_data(train_set)
         # Randomly shuffle the training Dataset.
         train_set = train_set.shuffle(split_size)
         val_set = prepare_data(val_set)

         # Batch both datasets with a batch size of 20, setting drop_remainder=True.
         train_set = train_set.batch(batch_size=20, drop_remainder=True)
         val_set = val_set.batch(batch_size=20, drop_remainder=True)

In [15]: # Print the element_spec property for one of the Dataset objects.
         train_set.element_spec

Out[15]: (TensorSpec(shape=(20, 36, 36, 3), dtype=tf.float32, name=None),
          TensorSpec(shape=(20, 36, 36, 3), dtype=tf.float32, name=None))
```

1.4 4. Build the encoder and decoder networks

- You should now create the encoder and decoder for the variational autoencoder algorithm.
- You should design these networks yourself, subject to the following constraints:
  - The encoder and decoder networks should be built using the `Sequential` class.
  - The encoder and decoder networks should use probabilistic layers where necessary to represent distributions.
  - The prior distribution should be a zero-mean, isotropic Gaussian (identity covariance matrix).
  - The encoder network should add the KL divergence loss to the model.
- Print the model summary for the encoder and decoder networks.

```
In [16]: # You should now create the encoder and decoder for the variational autoencoder algor
         # 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

         latent_dim = 2
         image_dim = images.shape[1:]

         # * The prior distribution should be a zero-mean, isotropic Gaussian (identity covari
         prior = tfd.MultivariateNormalDiag(loc=tf.Variable(tf.zeros(latent_dim), dtype=tf.floa
                                   scale_diag = tfp.util.TransformedVariable(initial_
                                                                             bijector
```

```
In [17]: # encoder part
         # inspired by https://www.tensorflow.org/tutorials/generative/cvae
         encoder = Sequential([
                               InputLayer(input_shape=image_dim),

                               Conv2D(filters=32, kernel_size=(3,3), activation='relu', padding
                               BatchNormalization(),
                               Conv2D(filters=64, kernel_size=(3,3), activation='relu', padding
                               BatchNormalization(),
                               Conv2D(filters=128, kernel_size=(3,3), activation='relu', paddin
                               BatchNormalization(),
                               Conv2D(filters=8, kernel_size=(1,1), activation='relu', padding=
                               BatchNormalization(),

                               Flatten(),
                               Dense(100),
                               BatchNormalization(),

                               Dense(tfpl.MultivariateNormalTriL.params_size(latent_dim), acti
                               tfpl.MultivariateNormalTriL(latent_dim),
                               # * The encoder network should add the KL divergence loss to th
                               tfpl.KLDivergenceAddLoss(prior,
                                                        use_exact_kl = False,
                                                        test_points_fn = lambda q:q.sample(5),
                                                        test_points_reduce_axis=(0,1))
         ], name='encoder')
         # Print the model summary for the encoder and decoder networks.
         encoder.summary()
```

```
WARNING:tensorflow:From /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/ops/lina
Instructions for updating:
Do not pass `graph_parents`.  They will  no longer be used.
Model: "encoder"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 36, 36, 32)        896
_________________________________________________________________
batch_normalization (BatchNo (None, 36, 36, 32)        128
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 36, 36, 64)        18496
_________________________________________________________________
batch_normalization_1 (Batch (None, 36, 36, 64)        256
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 36, 36, 128)       73856
```

```
_________________________________________________________________
batch_normalization_2 (Batch (None, 36, 36, 128)       512       
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 36, 36, 8)         1032      
_________________________________________________________________
batch_normalization_3 (Batch (None, 36, 36, 8)         32        
_________________________________________________________________
flatten (Flatten)            (None, 10368)             0         
_________________________________________________________________
dense (Dense)                (None, 100)               1036900   
_________________________________________________________________
batch_normalization_4 (Batch (None, 100)               400       
_________________________________________________________________
dense_1 (Dense)              (None, 5)                 505       
_________________________________________________________________
multivariate_normal_tri_l (M ((None, 2), (None, 2))    0         
_________________________________________________________________
kl_divergence_add_loss (KLDi (None, 2)                 4         
=================================================================
Total params: 1,133,017
Trainable params: 1,132,353
Non-trainable params: 664
_________________________________________________________________
```

In [18]:
```
# decoder part
# inspired by https://www.tensorflow.org/tutorials/generative/cvae
decoder = Sequential([
                      InputLayer(input_shape=(latent_dim,)),

                      Dense(8*8*8),
                      Reshape(target_shape=(8,8,8)),

                      Conv2DTranspose(filters=32, kernel_size=(3,3), strides=(2,2), a
                      BatchNormalization(),
                      Conv2DTranspose(filters=16, kernel_size=(3,3), strides=(2,2), a
                      BatchNormalization(),
                      Conv2DTranspose(filters=1, kernel_size=(3, 3), strides=(2,2), a
                      BatchNormalization(),

                      Flatten(),
                      Dense(tfpl.IndependentBernoulli.params_size(image_dim), activat
                      tfpl.IndependentBernoulli(event_shape=image_dim)
], name='decoder')
# Print the model summary for the encoder and decoder networks.
decoder.summary()
```

```
Model: "decoder"
_________________________________________________________________
```

```
Layout (type)                Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 512)               1536
_________________________________________________________________
reshape (Reshape)            (None, 8, 8, 8)           0
_________________________________________________________________
conv2d_transpose (Conv2DTran (None, 16, 16, 32)        2336
_________________________________________________________________
batch_normalization_5 (Batch (None, 16, 16, 32)        128
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 32, 32, 16)        4624
_________________________________________________________________
batch_normalization_6 (Batch (None, 32, 32, 16)        64
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 64, 64, 1)         145
_________________________________________________________________
batch_normalization_7 (Batch (None, 64, 64, 1)         4
_________________________________________________________________
flatten_1 (Flatten)          (None, 4096)              0
_________________________________________________________________
dense_3 (Dense)              (None, 3888)              15929136
_________________________________________________________________
independent_bernoulli (Indep ((None, 36, 36, 3), (None 0
=================================================================
Total params: 15,937,973
Trainable params: 15,937,875
Non-trainable params: 98
_________________________________________________________________
```

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 [19]: # Build the VAE using the Model class and the encoder and decoder models. Print the m
         vae = Model(inputs=encoder.inputs, outputs=decoder(encoder.outputs), name='vae')
```

```
In [20]: # Compile the VAE with the negative log likelihood loss
         def nll(y_true, y_pred):
             return -tf.reduce_mean(y_pred.log_prob(y_true))

         vae.compile(loss=nll, optimizer='adam')
         vae.summary()
```

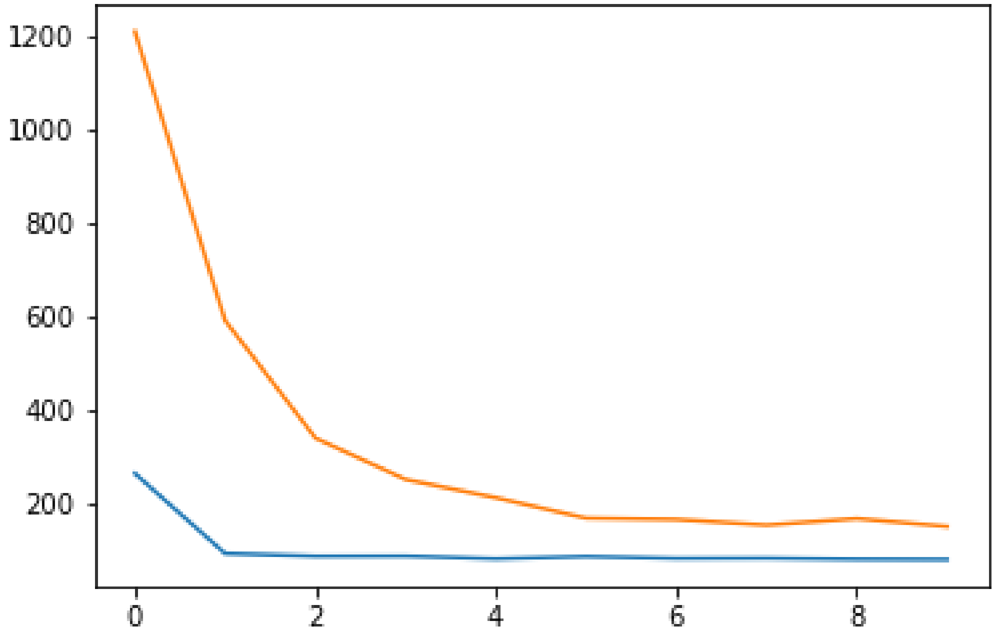
```
Model: "vae"
-----------------------------------------------------------------
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 36, 36, 3)]       0
-----------------------------------------------------------------
conv2d (Conv2D)              (None, 36, 36, 32)        896
-----------------------------------------------------------------
batch_normalization (BatchNo (None, 36, 36, 32)        128
-----------------------------------------------------------------
conv2d_1 (Conv2D)            (None, 36, 36, 64)        18496
-----------------------------------------------------------------
batch_normalization_1 (Batch (None, 36, 36, 64)        256
-----------------------------------------------------------------
conv2d_2 (Conv2D)            (None, 36, 36, 128)       73856
-----------------------------------------------------------------
batch_normalization_2 (Batch (None, 36, 36, 128)       512
-----------------------------------------------------------------
conv2d_3 (Conv2D)            (None, 36, 36, 8)         1032
-----------------------------------------------------------------
batch_normalization_3 (Batch (None, 36, 36, 8)         32
-----------------------------------------------------------------
flatten (Flatten)            (None, 10368)             0
-----------------------------------------------------------------
dense (Dense)                (None, 100)               1036900
-----------------------------------------------------------------
batch_normalization_4 (Batch (None, 100)               400
-----------------------------------------------------------------
dense_1 (Dense)              (None, 5)                 505
-----------------------------------------------------------------
multivariate_normal_tri_l (M ((None, 2), (None, 2))    0
-----------------------------------------------------------------
kl_divergence_add_loss (KLDi (None, 2)                 4
-----------------------------------------------------------------
decoder (Sequential)         (None, 36, 36, 3)         15937973
=================================================================
Total params: 17,070,990
Trainable params: 17,070,228
Non-trainable params: 762
-----------------------------------------------------------------
```

```
In [21]: # and train with the fit method, using the training and validation Datasets.
         early_stopping = tf.keras.callbacks.EarlyStopping(monitor="val_loss",
                                                           min_delta=0.1, patience=5,
                                                           restore_best_weights=True)
```

```
history = vae.fit(train_set,
                  validation_data=val_set,
                  epochs=10,
                  callbacks=[early_stopping],
                  verbose=2)
```

```
Train for 37 steps, validate for 12 steps
Epoch 1/10
37/37 - 100s - loss: 263.2699 - val_loss: 1208.6427
Epoch 2/10
37/37 - 99s - loss: 92.1871 - val_loss: 590.4367
Epoch 3/10
37/37 - 98s - loss: 86.4432 - val_loss: 339.0957
Epoch 4/10
37/37 - 99s - loss: 86.7019 - val_loss: 250.3676
Epoch 5/10
37/37 - 96s - loss: 80.8608 - val_loss: 211.6612
Epoch 6/10
37/37 - 94s - loss: 85.6911 - val_loss: 167.8786
Epoch 7/10
37/37 - 94s - loss: 81.7220 - val_loss: 165.1337
Epoch 8/10
37/37 - 94s - loss: 82.0641 - val_loss: 153.0014
Epoch 9/10
37/37 - 94s - loss: 79.9850 - val_loss: 166.9210
Epoch 10/10
37/37 - 95s - loss: 79.4047 - val_loss: 150.0522
```

```
In [22]: # Plot the learning curves for loss vs epoch for both training and validation sets.
         plt.plot(history.history["loss"])
         plt.plot(history.history["val_loss"])
         plt.show()
```

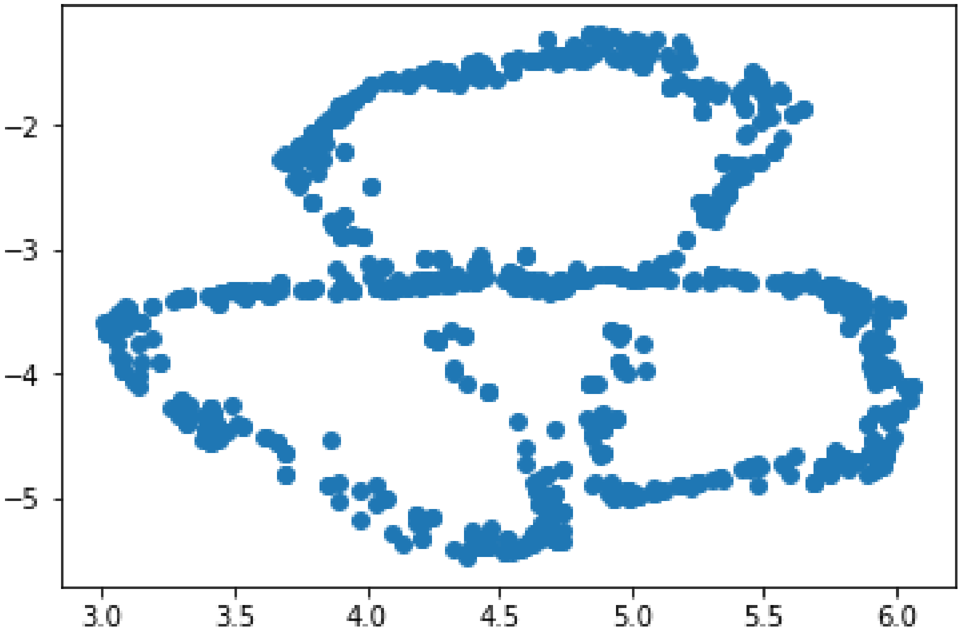


1.6 6. Use the encoder and decoder networks

- 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 [23]: # 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 spa

         idx = np.random.choice(np.arange(images.shape[0]), 1000)
         embeddings = encoder(images[idx]/255.0).mean()
         plt.scatter(embeddings[:,0], embeddings[:,1])
         plt.show()
```

```
WARNING:tensorflow:Layer conv2d is casting an input tensor from dtype float64 to the layer's d

If you intended to run this layer in float32, you can safely ignore this warning. If in doubt,
To change all layers to have dtype float64 by default, call `tf.keras.backend.set_floatx('floa
```



```
In [24]: # Randomly sample 4 images from the dataset and for each image, display the original
         # Use the mean of the output distribution to display the images.

         idx = np.random.choice(np.arange(images.shape[0]), 4)
         reconstructions = vae(images[idx]).mean().numpy()

         plt.figure(figsize=(15, 6))
         for i in range(4):
             plt.subplot(2, 4, i+1)
             plt.imshow(images[idx[i]])
             plt.title("Original: {}".format(i+1))
             plt.axis("off")

             plt.subplot(2, 4, i+5)
             plt.imshow(reconstructions[i])
             plt.title("Reconstruction: {}".format(i+1))
             plt.axis("off")
         plt.show()
```

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

```
    _FallbackException                        Traceback (most recent call last)

    /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/ops/gen_nn_ops.py in conv
    925         "explicit_paddings", explicit_paddings, "data_format", data_format,
--> 926         "dilations", dilations)
    927       return _result


    _FallbackException: Expecting int64_t value for attr strides, got numpy.int32


During handling of the above exception, another exception occurred:


    ValueError                                Traceback (most recent call last)

    <ipython-input-24-9a47393497c4> in <module>
      3
      4 idx = np.random.choice(np.arange(images.shape[0]), 4)
----> 5 reconstructions = vae(images[idx]).mean().numpy()
      6
      7 plt.figure(figsize=(15, 6))


    /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/keras/engine/base_layer.p
    820           with base_layer_utils.autocast_context_manager(
    821               self._compute_dtype):
--> 822             outputs = self.call(cast_inputs, *args, **kwargs)
    823           self._handle_activity_regularization(inputs, outputs)
    824           self._set_mask_metadata(inputs, outputs, input_masks)


    /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/keras/engine/network.py i
    715     return self._run_internal_graph(
    716         inputs, training=training, mask=mask,
--> 717         convert_kwargs_to_constants=base_layer_utils.call_context().saving)
    718
    719   def compute_output_shape(self, input_shape):


    /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/keras/engine/network.py i
    889
    890           # Compute outputs.
--> 891           output_tensors = layer(computed_tensors, **kwargs)
    892
    893           # Update tensor_dict.
    /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/keras/engine/base_layer.p
    820           with base_layer_utils.autocast_context_manager(
    821               self._compute_dtype):
--> 822             outputs = self.call(cast_inputs, *args, **kwargs)
    823           self._handle_activity_regularization(inputs, outputs)
    824           self._set_mask_metadata(inputs, outputs, input_masks)


    /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/keras/layers/convolutiona
    207       inputs = array_ops.pad(inputs, self._compute_causal_padding())
    208
--> 209     outputs = self._convolution_op(inputs, self.kernel)
    210
    211     if self.use_bias:


    /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/ops/nn_ops.py in __call_
   1133           call_from_convolution=False)
   1134     else:
-> 1135       return self.conv_op(inp, filter)
   1136     # copybara:strip_end
   1137     # copybara:insert return self.conv_op(inp, filter)


    /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/ops/nn_ops.py in __call_
    638
    639   def __call__(self, inp, filter):  # pylint: disable=redefined-builtin
--> 640     return self.call(inp, filter)
    641
    642


    /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/ops/nn_ops.py in __call_
    237         padding=self.padding,
    238         data_format=self.data_format,
--> 239         name=self.name)
    240
    241


    /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/ops/nn_ops.py in conv2d(
   2009                           data_format=data_format,
   2010                           dilations=dilations,
-> 2011                           name=name)
   2012
   2013
```

```
    /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/ops/gen_nn_ops.py in conv
    931             input, filter, strides=strides, use_cudnn_on_gpu=use_cudnn_on_gpu,
    932             padding=padding, explicit_paddings=explicit_paddings,
--> 933             data_format=data_format, dilations=dilations, name=name, ctx=_ctx)
    934       except _core._SymbolicException:
    935         pass  # Add nodes to the TensorFlow graph.


    /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/ops/gen_nn_ops.py in conv
   1013         "'conv2d' Op, not %r." % dilations)
   1014   dilations = [_execute.make_int(_i, "dilations") for _i in dilations]
-> 1015   _attr_T, _inputs_T = _execute.args_to_matching_eager([input, filter], ctx)
   1016   (input, filter) = _inputs_T
   1017   _inputs_flat = [input, filter]


    /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/eager/execute.py in args_
    265         dtype = ret[-1].dtype
    266   else:
--> 267     ret = [ops.convert_to_tensor(t, dtype, ctx=ctx) for t in l]
    268
    269   # TODO(slebedev): consider removing this as it leaks a Keras concept.


    /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/eager/execute.py in <list
    265         dtype = ret[-1].dtype
    266   else:
--> 267     ret = [ops.convert_to_tensor(t, dtype, ctx=ctx) for t in l]
    268
    269   # TODO(slebedev): consider removing this as it leaks a Keras concept.


    /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/framework/ops.py in conve
   1312
   1313     if ret is None:
-> 1314       ret = conversion_func(value, dtype=dtype, name=name, as_ref=as_ref)
   1315
   1316     if ret is NotImplemented:


    /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/ops/resource_variable_ops
   1792
   1793 def _dense_var_to_tensor(var, dtype=None, name=None, as_ref=False):
-> 1794   return var._dense_var_to_tensor(dtype=dtype, name=name, as_ref=as_ref)  # pylint
   1795
   1796
```

```
 /opt/conda/lib/python3.7/site-packages/tensorflow_core/python/ops/resource_variable_ops
1213        raise ValueError(
1214            "Incompatible type conversion requested to type {!r} for variable "
-> 1215            "of type {!r}".format(dtype.name, self.dtype.name))
1216      if as_ref:
1217        return self.read_value().op.inputs[0]


 ValueError: Incompatible type conversion requested to type 'uint8' for variable of typ
```

```
In [25]: # Randomly sample 6 latent variable realisations from the prior distribution, and dis
         # Again use the mean of the output distribution to display the images.

         latent_variables = np.random.uniform(-2, 2, (6, latent_dim))
         realisations = decoder(latent_variables).mean()

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

```
WARNING:tensorflow:Layer dense_2 is casting an input tensor from dtype float64 to the layer's c

If you intended to run this layer in float32, you can safely ignore this warning. If in doubt,

To change all layers to have dtype float64 by default, call `tf.keras.backend.set_floatx('floa
```

Realisation: 1
Realisation: 2
Realisation: 3
Realisation: 4
Realisation: 5
Realisation: 6

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

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

```
In [26]: # 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(n

             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 [27]: # Create the animation

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

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

```
    NameError                                 Traceback (most recent call last)

    <ipython-input-27-f8780fad386d> in <module>
      1 # Create the animation
      2
----> 3 a = get_animation(latent_size, decoder, interpolation_length=200)
      4 HTML(a.to_html5_video())


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

```
In [ ]:
```