Probabilistic Deep Learning with TensorFlow 2

Multivariate Gaussian with full covariance
Broadcasting rules
Tensorflow Distributions
Maximum likelihood estimation
Bayes by backprop
Probabilistic Layers And Bayesian Neural Networks
Scale bijectors and LinearOperator
Change of variables
Autoregressive flows and RealNVP
Bijectors And Normalising Flows
Variational autoencoders
Kullback-Leibler divergence
Full covariance Gaussian approximation
Variational Autoencoders

Multivariate Gaussian with full covariance

In this reading you will learn how you can use TensorFlow to specify any multivariate Gaussian distribution.

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

print('TF version:', tf.__version__)
print('TFP version:', tfp.__version__)
```

```
TF version: 2.3.0
TFP version: 0.11.0
```

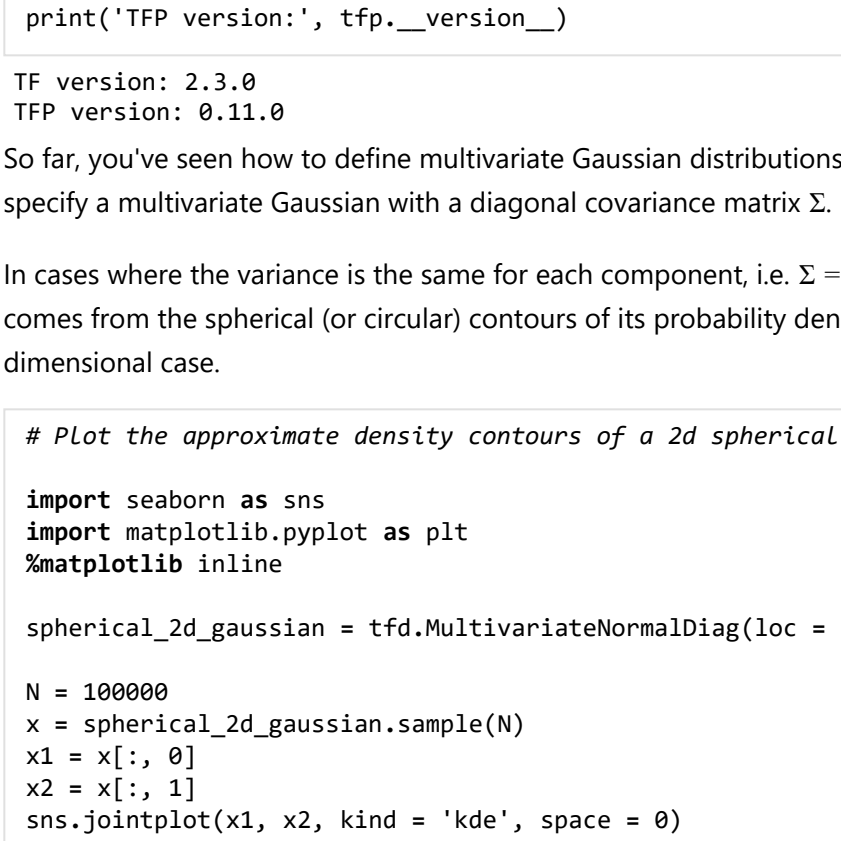
So far, you've seen how to define multivariate Gaussian distributions using `tfd.MultivariateNormalDiag` . This class allows you to specify a multivariate Gaussian with a diagonal covariance matrix Σ.

In cases where the variance is the same for each component, i.e. $\Sigma = \sigma^2 I$, this is known as a *spherical* or *isotropic* Gaussian. This name comes from the spherical (or circular) contours of its probability density function, as you can see from the plot below for the two-dimensional case.

In [2]:
```
# Plot the approximate density contours of a 2d spherical Gaussian

import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline

spherical_2d_gaussian = tfd.MultivariateNormalDiag(loc = [0., 0.])

N = 100000
x = spherical_2d_gaussian.sample(N)
x1 = x[:, 0]
x2 = x[:, 1]
sns.jointplot(x1, x2, kind = 'kde', space = 0)
```

```
/home/bacti/anaconda3/envs/tensor/lib/python3.7/site-packages/seaborn/_decorators.py:43: FutureWarning: Pass the following variables as keyword args: x, y. From version 0.12, the only valid positional argument will be `data`, and passing other arguments without an explicit keyword will result in an error or misinterpretation.
  FutureWarning
```

Out[2]: <seaborn.axisgrid.JointGrid at 0x7f55a5889210>



As you know, a diagonal covariance matrix results in the components of the random vector being independent.

Full covariance with `MultivariateNormalFullTriL`

You can define a full covariance Gaussian distribution in TensorFlow using the Distribution `tfd.MultivariateNormalTriL`.

Mathematically, the parameters of a multivariate Gaussian are a mean μ and a covariance matrix Σ, and so the `tfd.MultivariateNormalTriL` constructor requires two arguments:

- `loc`, a Tensor of floats corresponding to μ,
- `scale_tril`, a a lower-triangular matrix L such that $LL^T = \Sigma$.

For a d-dimensional random variable, the lower-triangular matrix L looks like this:

$$L = \begin{bmatrix} l_{1,1} & 0 & 0 & \cdots & 0 \\ l_{2,1} & l_{2,2} & 0 & \cdots & 0 \\ l_{3,1} & l_{3,2} & l_{3,3} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{d,1} & l_{d,2} & l_{d,3} & \cdots & l_{d,d} \end{bmatrix},$$

where the diagonal entries are positive: $l_{i,i} > 0$ for $i = 1, \ldots, d$.

Here is an example of creating a two-dimensional Gaussian with non-diagonal covariance:

In [3]:
```
# Set the mean and covariance parameters
mu = [0., 0.]  # mean
scale_tril = [[1.,  0.], [0.6, 0.8]]

sigma = tf.matmul(tf.constant(scale_tril), tf.transpose(tf.constant(scale_tril)))  # covariance matrix
print(sigma)
```

```
tf.Tensor(
[[1.  0.6]
 [0.6 1. ]], shape=(2, 2), dtype=float32)
```

In [4]:
```
# Create the 2D Gaussian with full covariance
nonspherical_2d_gaussian = tfd.MultivariateNormalTriL(loc = mu, scale_tril = scale_tril)
nonspherical_2d_gaussian
```

Out[4]:
```
<tfp.distributions.MultivariateNormalTriL 'MultivariateNormalTriL' batch_shape=[] event_shape=[2] dtype=float32>
```

In [5]:
```
# Check the Distribution mean
nonspherical_2d_gaussian.mean()
```

Out[5]:
```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([0., 0.], dtype=float32)>
```

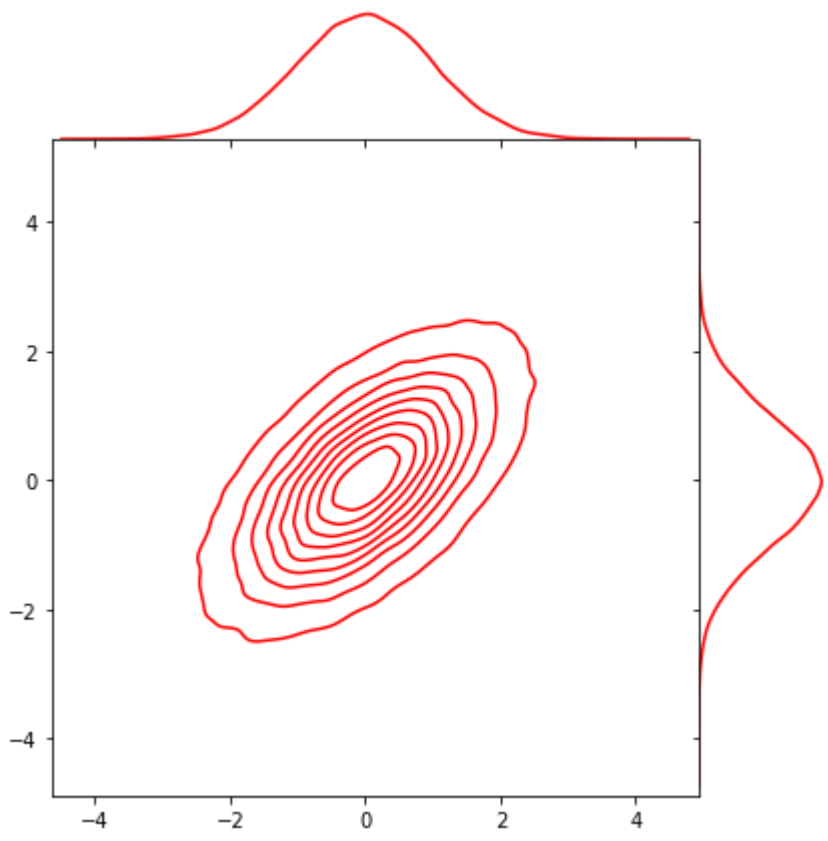
In [6]:
```
# Check the Distribution covariance
nonspherical_2d_gaussian.covariance()
```

Out[6]:
```
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[1. , 0.6],
       [0.6, 1. ]], dtype=float32)>
```

In [7]:
```
# Plot its approximate density contours
x = nonspherical_2d_gaussian.sample(N)
x1 = x[:, 0]
x2 = x[:, 1]
sns.jointplot(x1, x2, kind = 'kde', space = 0, color = 'r')
```

```
/home/bacti/anaconda3/envs/tensor/lib/python3.7/site-packages/seaborn/_decorators.py:43: FutureWarning: Pass the follo
wing variables as keyword args: x, y. From version 0.12, the only valid positional argument will be `data`, and passin
g other arguments without an explicit keyword will result in an error or misinterpretation.
  FutureWarning
```

Out[7]:
```
<seaborn.axisgrid.JointGrid at 0x7f55341cdc50>
```



As you can see, the approximate density contours are now elliptical rather than circular. This is because the components of the Gaussian are correlated.

Also note that the marginal distributions (shown on the sides of the plot) are both univariate Gaussian distributions.

The Cholesky decomposition

In the above example, we defined the lower triangular matrix L and used that to build the multivariate Gaussian distribution. The covariance matrix is easily computed from L as $\Sigma = LL^T$.

The reason that we define the multivariate Gaussian distribution in this way - as opposed to directly passing in the covariance matrix - is that not every matrix is a valid covariance matrix. The covariance matrix must have the following properties:

1. It is symmetric
2. It is positive (semi-)definite

NB: A symmetric matrix $M \in R^{d \times d}$ is positive semi-definite if it satisfies $b^T Mb \geq 0$ for all nonzero $b \in R^d$. If, in addition, we have $b^T Mb = 0 \Rightarrow b = 0$ then M is positive definite.

The Cholesky decomposition is a useful way of writing a covariance matrix. The decomposition is described by this result:

> For every real-valued symmetric positive-definite matrix M, there is a unique lower-diagonal matrix L that has positive diagonal entries for which
>
> $$LL^T = M$$
>
> This is called the *Cholesky decomposition* of M.

This result shows us why Gaussian distributions with full covariance are completely represented by the `MultivariateNormalTriL` Distribution.

`tf.linalg.cholesky`

In case you have a valid covariance matrix Σ and would like to compute the lower triangular matrix L above to instantiate a `MultivariateNormalTriL` object, this can be done with the `tf.linalg.cholesky` function.

In [8]:
```
# Define a symmetric positive-definite matrix
sigma = [[10., 5.], [5., 10.]]
```

In [9]:
```
# Compute the lower triangular matrix L from the Cholesky decomposition
scale_tril = tf.linalg.cholesky(sigma)
scale_tril
```

Out[9]:
```
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[3.1622777, 0.       ],
       [1.5811388, 2.7386127]], dtype=float32)>
```

In [10]:
```
# Check that LL^T = Sigma
tf.linalg.matmul(scale_tril, tf.transpose(scale_tril))
```

Out[10]:
```
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[10.      ,  5.      ],
       [ 5.      ,  9.999999]], dtype=float32)>
```

If the argument to the `tf.linalg.cholesky` is not positive definite, then it will fail:

In [11]:
```
# Try to compute the Cholesky decomposition for a matrix with negative eigenvalues
bad_sigma = [[10., 11.], [11., 10.]]

try:
    scale_tril = tf.linalg.cholesky(bad_sigma)
except Exception as e:
    print(e)
```

```
Cholesky decomposition was not successful. The input might not be valid. [Op:Cholesky]
```

What about positive semi-definite matrices?

In cases where the matrix is only positive semi-definite, the Cholesky decomposition exists (if the diagonal entries of L can be zero) but it is not unique.

For covariance matrices, this corresponds to the degenerate case where the probability density function collapses to a subspace of the event space. This is demonstrated in the following example:

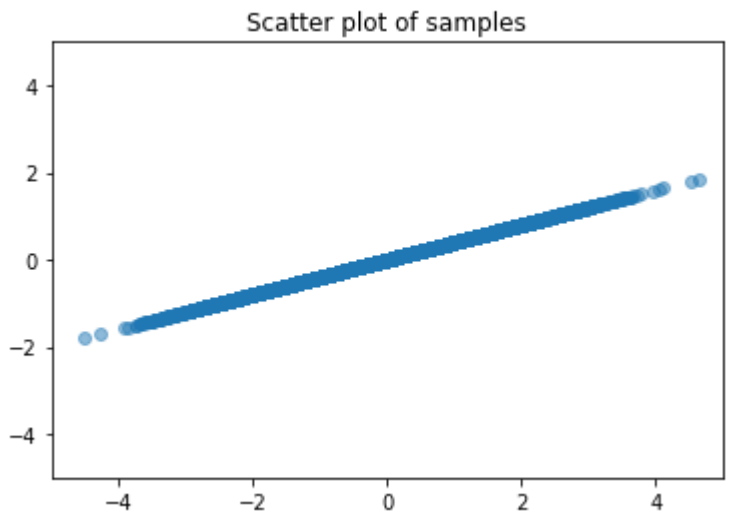
In [12]:
```
# Create a multivariate Gaussian with a positive semi-definite covariance matrix
psd_mvn = tfd.MultivariateNormalTriL(loc = [0., 0.], scale_tril = [[1., 0.], [0.4, 0.]])
psd_mvn
```

Out[12]:
```
<tfp.distributions.MultivariateNormalTriL 'MultivariateNormalTriL' batch_shape=[] event_shape=[2] dtype=float32>
```

In [13]:
```
# Plot samples from this distribution
x = psd_mvn.sample(N)
x1 = x[:, 0]
x2 = x[:, 1]
```

```
plt.xlim(-5, 5)
plt.ylim(-5, 5)
plt.title('Scatter plot of samples')
plt.scatter(x1, x2, alpha = 0.5)
```

Out[13]:
```
<matplotlib.collections.PathCollection at 0x7f55a5704e50>
```



If the input to the function `tf.linalg.cholesky` is positive semi-definite but not positive definite, it will also fail:

In [14]:
```
# Try to compute the Cholesky decomposition for a positive semi-definite matrix
another_bad_sigma = [[10., 0.], [0., 0.]]

try:
    scale_tril = tf.linalg.cholesky(another_bad_sigma)
except Exception as e:
    print(e)
```

```
Cholesky decomposition was not successful. The input might not be valid. [Op:Cholesky]
```

In summary: if the covariance matrix Σ for your multivariate Gaussian distribution is positive-definite, then an algorithm that computes the Cholesky decomposition of Σ returns a lower-triangular matrix L such that $LL^T = \Sigma$. This L can then be passed as the `scale_tril` of `MultivariateNormalTriL` .

Putting it all together

You are now ready to put everything that you have learned in this reading together.

To create a multivariate Gaussian distribution with full covariance you need to:

1. Specify parameters μ and either Σ (a symmetric positive definite matrix) or L (a lower triangular matrix with positive diagonal elements), such that $\Sigma = LL^T$.
2. If only Σ is specified, compute `scale_tril = tf.linalg.cholesky(sigma)` .
3. Create the distribution: `multivariate_normal = tfd.MultivariateNormalTriL(loc=mu, scale_tril=scale_tril)` .

In [15]:
```
# Create a multivariate Gaussian distribution
mu = [1., 2., 3.]
sigma = [
    [0.5, 0.1, 0.1],
    [0.1,  1., 0.6],
    [0.1, 0.6, 2.]
]

scale_tril = tf.linalg.cholesky(sigma)
multivariate_normal = tfd.MultivariateNormalTriL(loc = mu, scale_tril = scale_tril)
```

In [16]:
```
# Check the covariance matrix
multivariate_normal.covariance()
```

Out[16]:
```
<tf.Tensor: shape=(3, 3), dtype=float32, numpy=
array([[0.4999999 , 0.09999999, 0.09999999],
       [0.09999999, 1.0000001 , 0.6000001 ],
       [0.09999999, 0.6000001 , 2.0000002 ]], dtype=float32)>
```

In [17]:
```
# Check the mean
multivariate_normal.mean()
```

Out[17]:
```
<tf.Tensor: shape=(3,), dtype=float32, numpy=array([1., 2., 3.], dtype=float32)>
```

Deprecated: `MultivariateNormalFullCovariance`

There was previously a class called `tfd.MultivariateNormalFullCovariance` which takes the full covariance matrix in its constructor, but this is being deprecated. Two reasons for this are:

- covariance matrices are symmetric, so specifying one directly involves passing redundant information, which involves writing unnecessary code.
- it is easier to enforce positive-definiteness through constraints on the elements of a decomposition than through a covariance matrix itself. The decomposition's only constraint is that its diagonal elements are positive, a condition that is easy to parameterize for.

Further reading and resources

- https://www.tensorflow.org/probability/api_docs/python/tfp/distributions/MultivariateNormalTriL (https://www.tensorflow.org/probability/api_docs/python/tfp/distributions/MultivariateNormalTriL)
- https://www.tensorflow.org/api_docs/python/tf/linalg/cholesky (https://www.tensorflow.org/api_docs/python/tf/linalg/cholesky)

Broadcasting rules

This reading will introduce you to numpy's broadcasting rules and show how you can use broadcasting when specifying batches of distributions in TensorFlow, as well as with the `prob` and `log_prob` methods.

Broadcasting will also be discussed and demonstrated in the following videos.

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

print('TF version:', tf.__version__)
print('TFP version:', tfp.__version__)
```

```
TF version: 2.3.0
TFP version: 0.11.0
```

In [2]:
```
import numpy as np
```

Operations on arrays of different sizes in numpy

Numpy operations can be applied to arrays that are not of the same shape, but only if the shapes satisfy certain conditions.

As a demonstration of this, let us add together two arrays of different shapes:

In [3]:
```
# Add two arrays with different shapes
a = np.array([[1.], [2.], [3.], [4.]])  # shape (4, 1)
b = np.array([0., 1., 2.])  # shape (3,)
a + b
```

Out[3]:
```
array([[1., 2., 3.],
       [2., 3., 4.],
       [3., 4., 5.],
       [4., 5., 6.]])
```

This is the addition

```
[ [1.],    +  [0., 1., 2.]
  [2.],
  [3.],
  [4.] ]
```

To execute it, numpy:

1. Aligned the shapes of `a` and `b` on the last axis and prepended 1s to the shape with fewer axes:
   ```
   a: 4 x 1     --->    a: 4 x 1
   b:     3     --->    b: 1 x 3
   ```

1. Checked that the sizes of the axes matched or were equal to 1:
   ```
   a: 4 x 1
   b: 1 x 3
   ```
   `a` and `b` satisfied this criterion.

1. Stretched both arrays on their 1-valued axes so that their shapes matched, then added them together.
   `a` was replicated 3 times in the second axis, while `b` was replicated 4 times in the first axis.

This meant that the addition in the final step was

```
[ [1., 1., 1.],    +  [ [0., 1., 2.],
  [2., 2., 2.],         [0., 1., 2.],
  [3., 3., 3.],         [0., 1., 2.],
  [4., 4., 4.] ]        [0., 1., 2.] ]
```

Addition was then carried out element-by-element, as you can verify by referring back to the output of the code cell above.
This resulted in an output with shape 4 x 3.

Numpy's broadcasting rule

Broadcasting rules describe how values should be transmitted when the inputs to an operation do not match.
In numpy, the broadcasting rule is very simple:

> Prepend 1s to the smaller shape,
> check that the axes of both arrays have sizes that are equal or 1,
> then stretch the arrays in their size-1 axes.

A crucial aspect of this rule is that it does not require the input arrays have the same number of axes.
Another consequence of it is that a broadcasting output will have the largest size of its inputs in each axis.
Take the following multiplication as an example:

```
    a: 3 x 7 x 1
    b:     1 x 5
a * b: 3 x 7 x 5
```

You can see that the output shape is the maximum of the sizes in each axis.

Numpy's broadcasting rule also does not require that one of the arrays has to be bigger in all axes.
This is seen in the following example, where `a` is smaller than `b` in its third axis but is bigger in its second axis.

In [4]:
```
# Multiply two arrays with different shapes
a = np.array([[[0.01], [0.1]], [[1.00], [10.]]]) # shape (2, 2, 1)
b = np.array([[[2., 2.]], [[3., 3.]]]) # shape (2, 1, 2)
a * b # shape (2, 2, 2)
```

Out[4]:
```
array([[[2.e-02, 2.e-02],
        [2.e-01, 2.e-01]],

       [[3.e+00, 3.e+00],
        [3.e+01, 3.e+01]]])
```

Broadcasting behaviour also points to an efficient way to compute an outer product in numpy:

In [5]:
```
# Use broadcasting to compute an outer product
a = np.array([-1., 0., 1.])
b = np.array([0., 1., 2., 3.])
a[:, np.newaxis] * b  # outer product ab^T, where a and b are column vectors
```

Out[5]:
```
array([[-0., -1., -2., -3.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  1.,  2.,  3.]])
```

The idea of numpy stretching the arrays in their size-1 axes is useful and is functionally correct. But this is not what numpy literally does behind the scenes, since that would be an inefficient use of memory. Instead, numpy carries out the operation by looping over singleton (size-1) dimensions.

To give you some practise with broadcasting, try predicting the output shapes for the following operations:

In [6]:
```
# Define three arrays with different shapes
a = [[1.], [2.], [3.]]
b = np.zeros(shape = [10, 1, 1])
c = np.ones(shape = [4])
```

In [7]:
```
# Predict the shape before executing this cell
(a + b).shape
```

Out[7]:
```
(10, 3, 1)
```

In [8]:
```
# Predict the shape before executing this cell
(a * c).shape
```

Out[8]:
```
(3, 4)
```

In [9]:
```
# Predict the shape before executing this cell
(a * b + c).shape
```

Out[9]:
```
(10, 3, 4)
```

Broadcasting for univariate TensorFlow Distributions

The broadcasting rule for TensorFlow is the same as that for numpy. For example, TensorFlow also allows you to specify the parameters of Distribution objects using broadcasting.

What is meant by this can be understood through an example with the univariate normal distribution. Say that we wish to specify a parameter grid for six Gaussians. The parameter combinations to be used, `(loc, scale)`, are:

```
(0, 1)
(0, 10)
(0, 100)
(1, 1)
(1, 10)
(1, 100)
```

A laborious way of doing this is to explicitly pass each parameter to `tfd.Normal`:

In [10]:
```
# Define a batch of Normal distributions without broadcasting
batch_of_normals = tfd.Normal(loc = [0., 0., 0., 1., 1., 1.], scale = [1., 10., 100., 1., 10., 100.])
```

In [11]:
```
# Print the distribution and notice the batch and event shapes
batch_of_normals
```

Out[11]:
```
<tfp.distributions.Normal 'Normal' batch_shape=[6] event_shape=[] dtype=float32>
```

In [12]:
```
# Check the parameter values for loc
batch_of_normals.loc
```

Out[12]:
```
<tf.Tensor: shape=(6,), dtype=float32, numpy=array([0., 0., 0., 1., 1., 1.], dtype=float32)>
```

In [13]:
```
# Check the parameter values for scale
batch_of_normals.scale
```

Out[13]:
```
<tf.Tensor: shape=(6,), dtype=float32, numpy=array([  1.,  10., 100.,   1.,  10., 100.], dtype=float32)>
```

A more succinct way to create a batch of distributions for this parameter grid is to use broadcasting.
Consider what would happen if we were to broadcast these arrays according the rule discussed earlier:

```
loc = [ [0.],
        [1.] ]
scale = [1., 10., 100.]
```

The shapes would be stretched according to

```
loc:   2 x 1 ---> 2 x 3
scale: 1 x 3 ---> 2 x 3
```

resulting in

```
loc = [ [0., 0., 0.],
        [1., 1., 1.] ]
scale = [ [1., 10., 100.],
          [1., 10., 100.] ]
```

which are compatible with the `loc` and `scale` arguments of `tfd.Normal`.
Sure enough, this is precisely what TensorFlow does:

In [14]:
```
# Define a batch of Normal distributions with broadcasting
loc = [[0.], [1.]]
scale = [1., 10., 100.]
another_batch_of_normals = tfd.Normal(loc = loc, scale = scale)
```

In [15]:
```
# Print the distribution and notice the batch and event shapes
another_batch_of_normals
```

Out[15]:
```
<tfp.distributions.Normal 'Normal' batch_shape=[2, 3] event_shape=[] dtype=float32>
```

In [16]:
```
# The stored loc parameter values are what you pass in, not what is used after broadcasting
another_batch_of_normals.loc
```

Out[16]:
```
<tf.Tensor: shape=(2, 1), dtype=float32, numpy=
array([[0.],
       [1.]], dtype=float32)>
```

In [17]:
```
# The stored scale parameter values are what you pass in, not what is used after broadcasting
another_batch_of_normals.scale
```

Out[17]:
```
<tf.Tensor: shape=(3,), dtype=float32, numpy=array([  1.,  10., 100.], dtype=float32)>
```

In summary, TensorFlow broadcasts parameter arrays: it stretches them according to the broadcasting rule, then creates a distribution on an element-by-element basis.

Broadcasting with `prob` and `log_prob` methods

When using `prob` and `log_prob` with broadcasting, we follow the same principles as before. Let's make a new batch of normals as before but with means which are centered at different locations to help distinguish the results we get.

In [18]:
```
# Define a batch of Normal distributions with broadcasting
loc = [[0.], [10.]]
scale = [1., 1., 1.]
another_batch_of_normals = tfd.Normal(loc = loc, scale = scale)
another_batch_of_normals
```

Out[18]:
```
<tfp.distributions.Normal 'Normal' batch_shape=[2, 3] event_shape=[] dtype=float32>
```

We can feed in samples of any shape as long as it can be broadcast agasint our batch shape for this example.

In [19]:
```
# Use broadcasting along the second axis with the prob method
sample = tf.random.uniform((2, 1))
```

```
another_batch_of_normals.prob(sample)
```

Out[19]:
```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[3.6183167e-01, 3.6183167e-01, 3.6183167e-01],
       [4.8273273e-19, 4.8273273e-19, 4.8273273e-19]], dtype=float32)>
```

Or boradcasting along the first axis instead:

In [20]:
```
# Use broadcasting along the first axis with the prob method
sample = tf.random.uniform((1, 3))
another_batch_of_normals.prob(sample)
```

Out[20]:
```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[3.4341156e-01, 3.7987795e-01, 3.7401137e-01],
       [1.5809656e-20, 1.6750055e-21, 2.6204080e-21]], dtype=float32)>
```

Or even both axes:

In [21]:
```
# Use broadcasting along both axes with the prob method
sample = tf.random.uniform((1, 1))
another_batch_of_normals.prob(sample)
```

Out[21]:
```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[3.6780354e-01, 3.6780354e-01, 3.6780354e-01],
       [3.9974444e-21, 3.9974444e-21, 3.9974444e-21]], dtype=float32)>
```

`log_prob` works in the exact same way with broadcasting. We can replace `prob` with `log_prob` in any of the previous examples:

In [22]:
```
# Use broadcasting along the first axis with the log_prob method
sample = tf.random.uniform((1, 3))
another_batch_of_normals.log_prob(sample)
```

Out[22]:
```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[ -0.95117575,  -0.9875492 ,  -0.96976864],
       [-48.411987  , -47.28321   , -47.78135   ]], dtype=float32)>
```

Broadcasting for multivariate TensorFlow distributions

Broadcasting behaviour for multivariate distributions is only a little more sophisticated than it is for univariate distributions.

Recall that `MultivariateNormalDiag` has two parameter arguments: `loc` and `scale_diag`. When specifying a single distribution, these arguments are vectors of the same length:

In [23]:
```
# Define a multivariate Gaussian distribution without broadcasting
single_mvt_normal = tfd.MultivariateNormalDiag(loc = [0., 0.], scale_diag = [1., 0.5])
single_mvt_normal
```

Out[23]:
```
<tfp.distributions.MultivariateNormalDiag 'MultivariateNormalDiag' batch_shape=[] event_shape=[2] dtype=float32>
```

In [24]:
```
# Print the loc parameter
single_mvt_normal.loc
```

Out[24]:
```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([0., 0.], dtype=float32)>
```

In [25]:
```
# Print the covariance matrix - the diagonal is scale_diag^2
single_mvt_normal.covariance()
```

Out[25]:
```
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[1.  , 0.  ],
       [0.  , 0.25]], dtype=float32)>
```

The size of the final axis of the inputs determines the event shape for each distribution in the batch. This means that if we pass

```
loc = [ [0., 0.],
        [1., 1.] ]
scale_diag = [1., 0.5]
```

such that

```
loc:        2 x 2
scale_diag: 1 x 2
                ^ final dimension is interpreted as event dimension
            ^ other dimensions are interpreted as batch dimensions
```

then a batch of two bivariate normal distributions will be created.

In [26]:
```
# Define a multivariate Gaussian distribution with broadcasting
loc = [[0., 0.], [1., 1.]]
scale_diag = [1., 0.5]
batch_of_mvt_normals = tfd.MultivariateNormalDiag(loc = loc, scale_diag = scale_diag)
```

In [27]:
```
# Print the distribution - note the event_shape and batch_shape
batch_of_mvt_normals
```

Out[27]:
```
<tfp.distributions.MultivariateNormalDiag 'MultivariateNormalDiag' batch_shape=[2] event_shape=[2] dtype=float32>
```

In [28]:
```
# Print the distribution parameters
# There is a batch of two distributions with different means and same covariance
```

```
batch_of_mvt_normals.parameters
```

Out[28]:
```
{'loc': ListWrapper([ListWrapper([0.0, 0.0]), ListWrapper([1.0, 1.0])]),
 'scale_diag': ListWrapper([1.0, 0.5]),
 'scale_identity_multiplier': None,
 'validate_args': False,
 'allow_nan_stats': True,
 'name': 'MultivariateNormalDiag'}
```

Knowing that, for multivariate distributions, TensorFlow

- interprets the final axis of an array of parameters as the event shape,
- and broadcasts over the remaining axes,

can you predict what the batch and event shapes will if we pass the arguments

```
loc = [ [ 1.,  1.,  1.],
        [-1., -1., -1.] ] # shape (2, 3)
scale_diag = [ [[0.1, 0.1, 0.1]],
               [[10., 10., 10.]] ] # shape (2, 1, 3)
```

to `MultivariateNormalDiag` ?

Solution:

Align the parameter array shapes on their last axis, prepending 1s where necessary:

```
          loc: 1 x 2 x 3
scale_diag: 2 x 1 x 3
```

The final axis has size 3, so `event_shape = (3)` . The remaining axes are broadcast over to yield

```
          loc: 2 x 2 x 3
scale_diag: 2 x 2 x 3
```

so `batch_shape = (2, 2)` .

Let's see if this is correct!

In [29]:
```
# Define a multivariate Gaussian distribution with broadcasting
loc = [[1.,  1.,  1.], [-1., -1., -1.]] # shape (2, 3)
scale_diag = [[[0.1, 0.1, 0.1]], [[10., 10., 10.]]] # shape (2, 1, 3)
another_batch_of_mvt_normals = tfd.MultivariateNormalDiag(loc = loc, scale_diag = scale_diag)
```

In [30]:
```
# Print the distribution and note batch and event shapes - bingo!
another_batch_of_mvt_normals
```

Out[30]:
```
<tfp.distributions.MultivariateNormalDiag 'MultivariateNormalDiag' batch_shape=[2, 2] event_shape=[3] dtype=float32>
```

In [31]:
```
# Print the distribution parameters
another_batch_of_mvt_normals.parameters
```

Out[31]:
```
{'loc': ListWrapper([ListWrapper([1.0, 1.0, 1.0]), ListWrapper([-1.0, -1.0, -1.0])]),
 'scale_diag': ListWrapper([ListWrapper([ListWrapper([0.1, 0.1, 0.1])]), ListWrapper([ListWrapper([10.0, 10.0, 10.
0])])]),
 'scale_identity_multiplier': None,
 'validate_args': False,
 'allow_nan_stats': True,
 'name': 'MultivariateNormalDiag'}
```

As we did before lets also look at broadcasting when we have batches of multivariate distributions.

In [32]:
```
# Define a batch of Normal distributions with broadcasting
loc = [[0.], [1.], [0.]]
scale = [1., 10., 100., 1., 10, 100.]
another_batch_of_normals = tfd.Normal(loc = loc, scale = scale)
another_batch_of_normals
```

Out[32]:
```
<tfp.distributions.Normal 'Normal' batch_shape=[3, 6] event_shape=[] dtype=float32>
```

And to refresh our memory of `Independent` we'll use it below to roll the rightmost batch shape into the event shape.

In [33]:
```
# Create a multivariate Independent distribution
another_batch_of_mvt_normals = tfd.Independent(another_batch_of_normals)
another_batch_of_mvt_normals
```

Out[33]:
```
<tfp.distributions.Independent 'IndependentNormal' batch_shape=[3] event_shape=[6] dtype=float32>
```

Now, onto the broadcasting:

In [34]:
```
# Use broadcasting with the prob method
# B shaped input (broadcast over event)
sample = tf.random.uniform((3, 1))
another_batch_of_mvt_normals.prob(sample)
```

Out[34]:
```
<tf.Tensor: shape=(3,), dtype=float32, numpy=array([1.5887796e-09, 3.7002286e-09, 2.5683733e-09], dtype=float32)>
```

In [35]:
```
# Use broadcasting with the prob method
# E shaped input (broadcast over batch)
sample = tf.random.uniform((1, 6))
another_batch_of_mvt_normals.prob(sample)
```

Out[35]:
```
<tf.Tensor: shape=(3,), dtype=float32, numpy=array([2.850026e-09, 2.595904e-09, 2.850026e-09], dtype=float32)>
```

In [36]:
```
# Use broadcasting with the prob method
# [S,B,E] shaped input (broadcast over samples)
sample = tf.random.uniform((2, 3, 6))
another_batch_of_mvt_normals.prob(sample)
```

Out[36]:
```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[3.3494645e-09, 2.0184825e-09, 2.4749744e-09],
       [3.0989775e-09, 2.5836933e-09, 3.9658077e-09]], dtype=float32)>
```

In [37]:
```
# [S,b,e] shaped input where [b,e] can be broadcast agaisnt [B,E]
sample = tf.random.uniform((2, 1, 6))
another_batch_of_mvt_normals.prob(sample)
```

Out[37]:
```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[3.4162173e-09, 2.2252702e-09, 3.4162173e-09],
       [2.4587419e-09, 3.3255294e-09, 2.4587419e-09]], dtype=float32)>
```

As a final example with `log_prob` instead of `prob`

In [38]:
```
# Use broadcasting with the log_prob method
# [S,b,e] shaped input where [b,e] can be broadcast agaisnt [B,E]
sample = tf.random.uniform((2, 3, 1))
another_batch_of_mvt_normals.prob(sample)
```

Out[38]:
```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[4.0049937e-09, 3.6004075e-09, 2.5192635e-09],
       [4.0302814e-09, 2.1555266e-09, 2.2629485e-09]], dtype=float32)>
```

You should now feel confident specifying batches of distributions using broadcasting. As you may have already guessed, broadcasting is especially useful when specifying grids of hyperparameters.

If you don't feel entirely comfortable with broadcasting quite yet, don't worry: re-read this notebook, go through the further reading provided below, and experiment with broadcasting in both numpy and TensorFlow, and you'll be broadcasting in no time.

Further reading and resources

- Numpy documentation on broadcasting: https://numpy.org/devdocs/user/theory.broadcasting.html (https://numpy.org/devdocs/user/theory.broadcasting.html)
- https://www.tensorflow.org/xla/broadcasting (https://www.tensorflow.org/xla/broadcasting)

Tensorflow Distributions

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

print('TF version:', tf.__version__)
print('TFP version:', tfp.__version__)
```

```
TF version: 2.3.0
TFP version: 0.11.0
```

In [2]:
```
# Additional imports and setting fixed random seed to have reproducibility
import matplotlib.pyplot as plt
import numpy as np
tf.random.set_seed(123)
```

1. Univariate Distributions

2. Multivariate Distributions

3. The Independent Distribution

4. Sampling and log probs

5. Trainable Distributions

Univariate distributions

In [3]:
```
# Create a normal distribution from Tensorflow Distributions
normal = tfd.Normal(loc = 0, scale = 1)
normal
```

Out[3]:
```
<tfp.distributions.Normal 'Normal' batch_shape=[] event_shape=[] dtype=float32>
```

In [4]:
```
# Sample from the chosen distribution...
```

```
normal.sample()
```

Out[4]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=-0.8980837>
```

In [5]:
```
# ... or sample multiple times
normal.sample(10)
```

Out[5]:
```
<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([ 0.33875433,  0.3449861 , -0.6605785 , -0.28549942,  0.43852386,
        0.8288566 , -0.53591555, -0.53534836, -1.0324249 , -2.942705  ],
      dtype=float32)>
```

In [6]:
```
# Obtain value of probability's density
normal.prob(0)
```

Out[6]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=0.3989423>
```

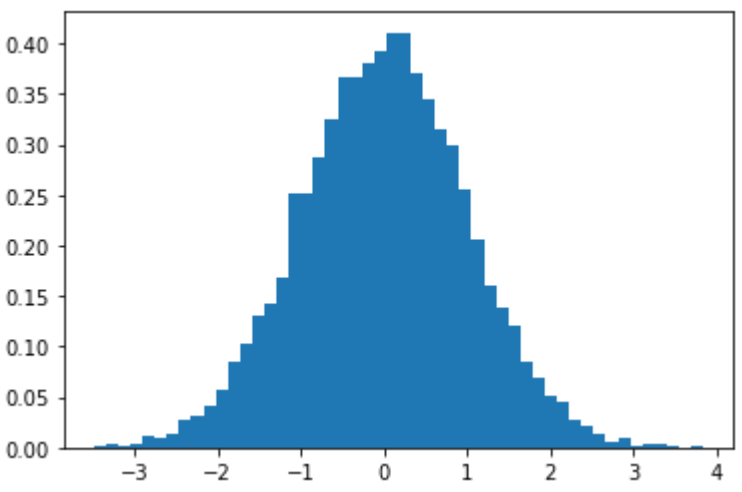
In [7]:
```
# Obtain value of logprobability
normal.log_prob(0)
```

Out[7]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=-0.9189385>
```

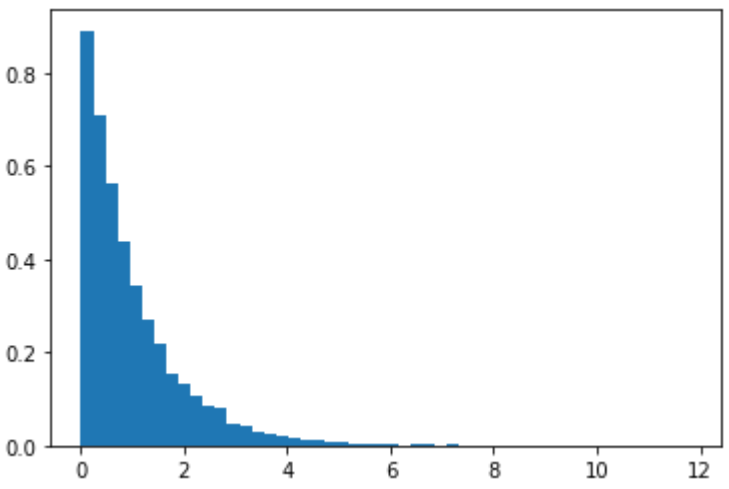
In [8]:
```
# Verify that this really is the log of the probability
np.log(normal.prob(0))
```

Out[8]:
```
-0.9189385
```

In [9]:
```
# Plot a histogram, approximating the density
plt.hist(x = normal.sample(10000).numpy(), bins = 50, density = True)
plt.show()
```



In [10]:
```
# Do the same for the exponential distribution
exponential = tfd.Exponential(rate = 1)
plt.hist(x = exponential.sample(10000).numpy(), bins = 50, density = True)
plt.show()
```



In [11]:
```
# Sample as before
exponential.sample(10)
```

Out[11]:
```
<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([0.3179616 , 0.9595855 , 0.6190708 , 1.3738598 , 1.6796894 ,
       0.40142855, 1.6830153 , 1.78942   , 0.38126466, 0.5528394 ],
      dtype=float32)>
```

In [12]:
```
# Create a Bernoulli distribution (discrete)
bernoulli = tfd.Bernoulli(probs = 0.8)
bernoulli.sample(20)
```

Out[12]:
```
<tf.Tensor: shape=(20,), dtype=int32, numpy=
array([0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1],
      dtype=int32)>
```

A word of caution on discrete distributions

In [13]:
```
# Calculate Bernoulli prob and see that 0.5 and -1 do not give the correct probability!
for k in [0, 0.5, 1, -1]:
    print('prob result {} for k = {} '.format(bernoulli.prob(k), k))
```

```
prob result 0.20000000298023224 for k = 0
prob result 0.4000000059604645 for k = 0.5
```

```
prob result 0.800000011920929 for k = 1
prob result 0.04999999701976776 for k = -1
```

In [14]:
```
# Replicate the scores to see what is occurring under the hood
def my_bernoulli(p_success, k):
    return np.power(p_success, k) * np.power((1 - p_success), (1 - k))
```

In [15]:
```
# Evaluate it as before
for k in [0, 0.5, 1, -1]:
    print('prob result {} for k = {} '.format(my_bernoulli(p_success = 0.8, k = k), k))
```

```
prob result 0.19999999999999996 for k = 0
prob result 0.3999999999999999 for k = 0.5
prob result 0.8 for k = 1
prob result 0.049999999999999975 for k = -1
```

Work with batch distributions

In [16]:
```
# Create a batched Bernoulli distribution
bernoulli_batch = tfd.Bernoulli(probs = [0.1, 0.25, 0.5, 0.75, 0.9])
bernoulli_batch
```

Out[16]:
```
<tfp.distributions.Bernoulli 'Bernoulli' batch_shape=[5] event_shape=[] dtype=int32>
```

In [17]:
```
# Sample from it, noting the shape
bernoulli_batch.sample(5)
```

Out[17]:
```
<tf.Tensor: shape=(5, 5), dtype=int32, numpy=
array([[0, 0, 0, 1, 1],
       [0, 1, 0, 1, 1],
       [0, 0, 1, 1, 1],
       [0, 0, 0, 1, 1],
       [0, 0, 0, 1, 1]], dtype=int32)>
```

In [18]:
```
# Use a batch shape with higher rank
probs = [[[0.5, 0.5],  [0.8, 0.3],  [0.25, 0.75]]]
bernoulli_batch_2D = tfd.Bernoulli(probs = probs)
bernoulli_batch_2D
```

Out[18]:
```
<tfp.distributions.Bernoulli 'Bernoulli' batch_shape=[1, 3, 2] event_shape=[] dtype=int32>
```

In [19]:
```
# Sample from this batch of distributions
bernoulli_batch_2D.sample(5)
```

Out[19]:
```
<tf.Tensor: shape=(5, 1, 3, 2), dtype=int32, numpy=
array([[[[0, 0],
         [1, 1],
         [0, 1]]],


       [[[0, 0],
         [1, 0],
         [0, 1]]],


       [[[1, 0],
         [1, 1],
         [0, 1]]],


       [[[1, 0],
         [0, 0],
         [0, 1]]],


       [[[0, 1],
         [1, 1],
         [1, 0]]]], dtype=int32)>
```

In [20]:
```
# Determine probabilities from this batch distribution
bernoulli_batch_2D.prob([[[1, 0], [0, 0], [1, 1]]])
```

Out[20]:
```
<tf.Tensor: shape=(1, 3, 2), dtype=float32, numpy=
array([[[0.5      , 0.5      ],
        [0.2      , 0.6999999],
        [0.25     , 0.75     ]]], dtype=float32)>
```

Multivariate Distributions

Basic multivariate distributions

In [21]:
```
# Define 2D multivariate Gaussian with diagonal covariance matrix
normal_diag = tfd.MultivariateNormalDiag(loc = [0, 1], scale_diag = [1, 2])
normal_diag
```

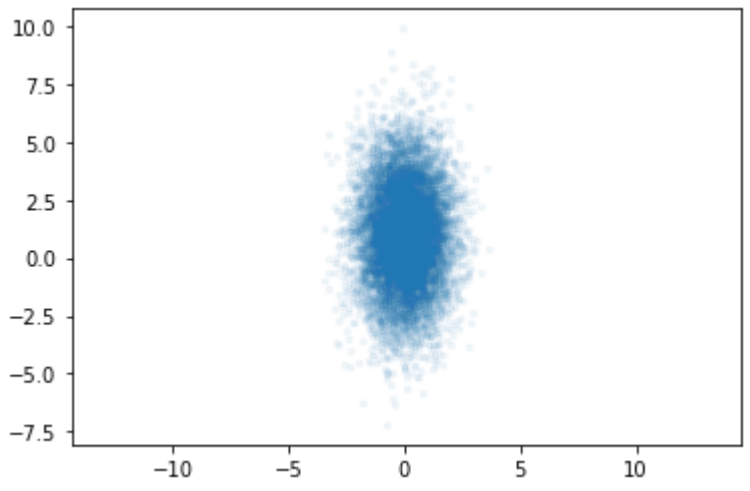
Out[21]:
```
<tfp.distributions.MultivariateNormalDiag 'MultivariateNormalDiag' batch_shape=[] event_shape=[2] dtype=float32>
```

In [22]:
```
# Sample from it
normal_diag.sample(10)
```

Out[22]:
```
<tf.Tensor: shape=(10, 2), dtype=float32, numpy=
array([[-0.37992278,  2.3674164 ],
```

```
[-2.224005  , -0.28514457],
[ 0.923083  , -1.4528892 ],
[-0.62774605, -0.33852375],
[-0.6252951 , -1.3324146 ],
[-0.42454168,  1.3192185 ],
[-1.702882  ,  1.8533869 ],
[-0.4608376 , -0.7023523 ],
[-1.1919353 , -0.12865639],
[ 0.48053816, -0.2693485 ]], dtype=float32)>
```

In [23]:
```
# Make a plot
plt_sample = normal_diag.sample(10000)
plt.scatter(plt_sample[:, 0], plt_sample[:, 1], marker = '.', alpha = 0.05)
plt.axis('equal')
plt.show()
```



Batches of multivariate distributions

In [24]:
```
# Create three "batches" of multivariate normals
normal_diag_batch = tfd.MultivariateNormalDiag \
    (loc = [[0, 0], [0, 0], [0, 0]], scale_diag = [[1, 2], [2, 1], [2, 2]])
normal_diag_batch
```

Out[24]:
```
<tfp.distributions.MultivariateNormalDiag 'MultivariateNormalDiag' batch_shape=[3] event_shape=[2] dtype=float32>
```

In [25]:
```
# Sample from it
samples = normal_diag_batch.sample(5)
samples
```

Out[25]:
```
<tf.Tensor: shape=(5, 3, 2), dtype=float32, numpy=
array([[[-0.8012545 , -2.128108  ],
        [ 2.0774972 , -2.7921855 ],
        [ 0.52665955,  0.60957587]],

       [[ 0.9923561 , -0.9778331 ],
        [-0.8376892 ,  0.70630246],
        [-1.0894657 , -0.65969497]],

       [[-1.6264789 ,  2.2429497 ],
        [-4.301875  , -0.7626804 ],
        [-0.4345196 , -0.57022095]],

       [[ 0.73075646,  2.835662  ],
        [ 1.8173586 ,  1.2079152 ],
        [-3.2939956 ,  2.33647   ]],

       [[-0.24759005,  0.56306183],
        [-0.6053428 ,  0.06578209],
        [ 1.4922864 , -0.55439734]]], dtype=float32)>
```

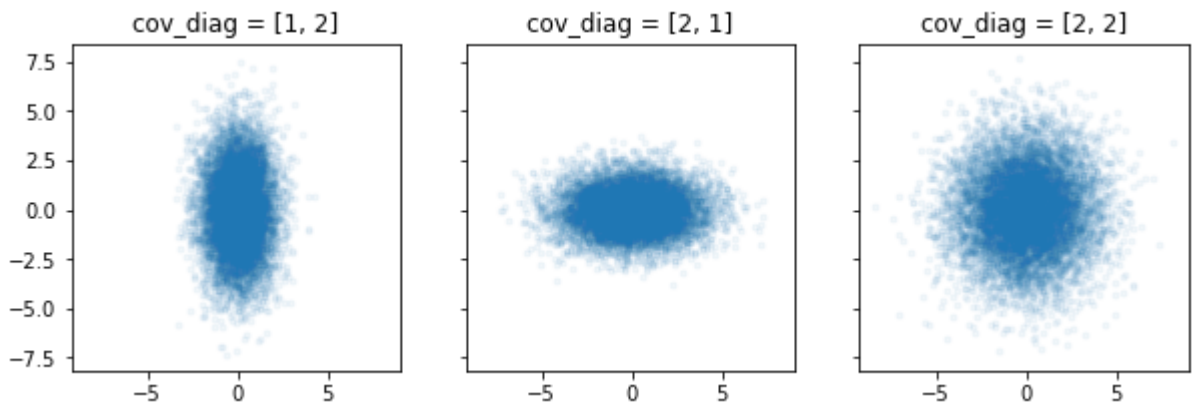
In [26]:
```
# Compute log probs
normal_diag_batch.log_prob(samples)
```

Out[26]:
```
<tf.Tensor: shape=(5, 3), dtype=float32, numpy=
array([[-3.4181342, -6.9686737, -3.3052907],
       [-3.142929 , -2.8681712, -3.426938 ],
       [-4.482594 , -5.1351314, -3.2884164],
       [-3.803149 , -3.6734028, -5.262859 ],
       [-2.6013045, -2.5789928, -3.5409558]], dtype=float32)>
```

In [27]:
```
# Create a sample for a plot -- notice the shape
plt_sample_batch = normal_diag_batch.sample(10000)
plt_sample_batch.shape
```

Out[27]:
```
TensorShape([10000, 3, 2])
```

In [28]:
```
# Plot samples from the batched multivariate Gaussian
fig, axs = (plt.subplots(1, 3, sharex = True, sharey = True, figsize = (10, 3)))
titles = ['cov_diag = [1, 2]','cov_diag = [2, 1]', 'cov_diag = [2, 2]']

for i, (ax, title) in enumerate(zip(axs, titles)):
    samples = plt_sample_batch[:, i, :] #take the ith batch [samples x event_shape]
    ax.scatter(samples[:, 0], samples[:, 1], marker = '.', alpha = 0.05)
    ax.set_title(title)
plt.show()
```

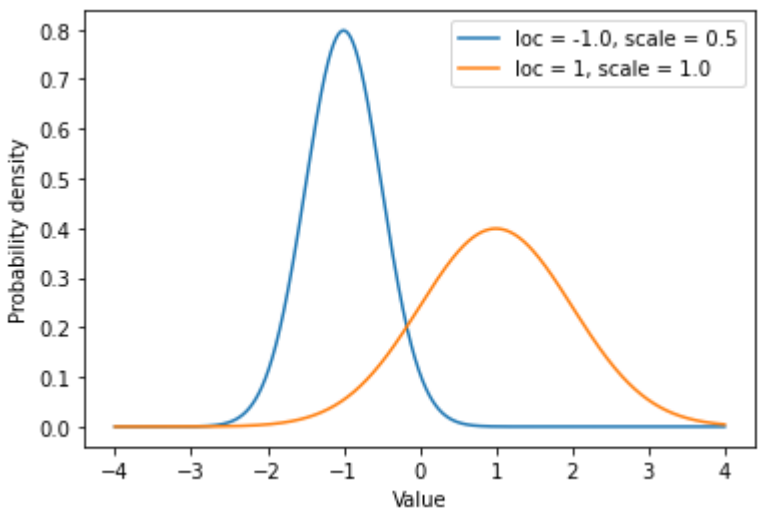


The Independent Distribution

In [29]:
```
# Start by defining a batch of two univariate Gaussians, then
# combine them into a bivariate Gaussian with independent components
locs = [-1., 1]
scales = [0.5, 1.]
batch_of_normals = tfd.Normal(loc = locs, scale = scales)
```

In [30]:
```
# Univariate density functions
import seaborn as sns

t = np.linspace(-4, 4, 10000)
densities = batch_of_normals.prob(np.repeat(t[:, np.newaxis], 2, axis = 1)) # each column is a vector of densities fo
sns.lineplot(t, densities[:, 0], label = 'loc = {}, scale = {}'.format(locs[0], scales[0]))
sns.lineplot(t, densities[:, 1], label = 'loc = {}, scale = {}'.format(locs[1], scales[1]))
plt.ylabel('Probability density')
plt.xlabel('Value')
plt.legend()
plt.show()
```

```
/home/bacti/anaconda3/envs/tensor/lib/python3.7/site-packages/seaborn/_decorators.py:43: FutureWarning: Pass the follo
wing variables as keyword args: x, y. From version 0.12, the only valid positional argument will be `data`, and passin
g other arguments without an explicit keyword will result in an error or misinterpretation.
  FutureWarning
/home/bacti/anaconda3/envs/tensor/lib/python3.7/site-packages/seaborn/_decorators.py:43: FutureWarning: Pass the follo
wing variables as keyword args: x, y. From version 0.12, the only valid positional argument will be `data`, and passin
g other arguments without an explicit keyword will result in an error or misinterpretation.
  FutureWarning
```



In [31]:
```
# Check their batch_shape and event_shape
batch_of_normals
```

Out[31]:
```
<tfp.distributions.Normal 'Normal' batch_shape=[2] event_shape=[] dtype=float32>
```

In [32]:
```
# Use Independent to convert the batch shape to the event shape
bivariate_normal_from_Independent = tfd.Independent(batch_of_normals, reinterpreted_batch_ndims = 1)
```

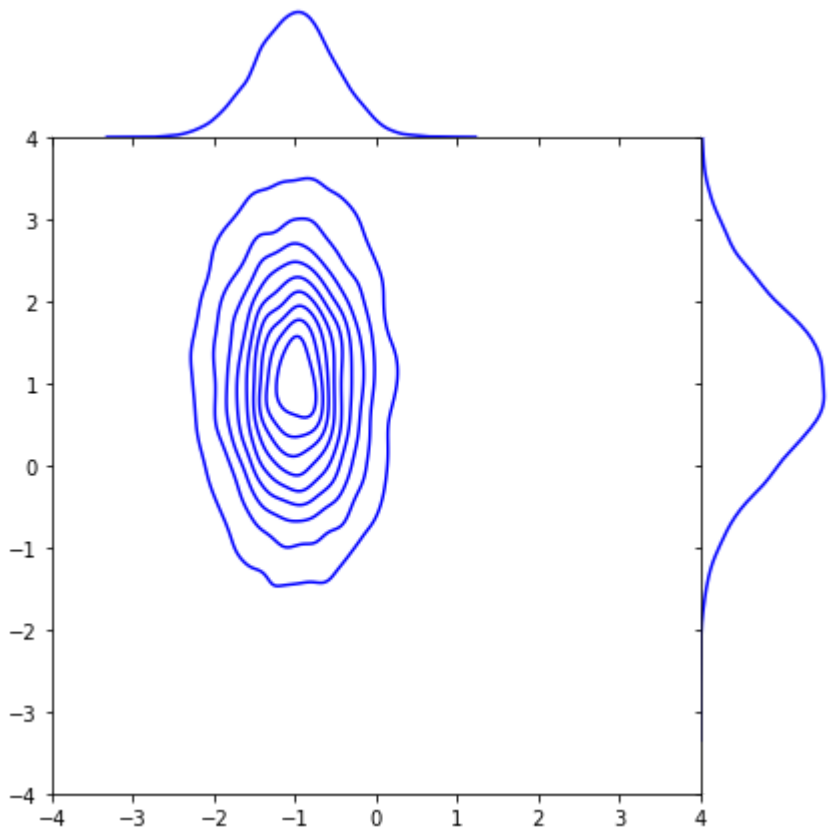
In [33]:
```
# Note that dimension from batch_shape has shifted to event_shape
bivariate_normal_from_Independent
```

Out[33]:
```
<tfp.distributions.Independent 'IndependentNormal' batch_shape=[] event_shape=[2] dtype=float32>
```

In [34]:
```
# Create a plot showing joint density contours and marginal density functions
samples = bivariate_normal_from_Independent.sample(10000)
x1 = samples[:, 0]
x2 = samples[:, 1]
sns.jointplot(x1, x2, kind = 'kde', space = 0, color = 'b', xlim = [-4, 4], ylim = [-4, 4])
```

```
/home/bacti/anaconda3/envs/tensor/lib/python3.7/site-packages/seaborn/_decorators.py:43: FutureWarning: Pass the follo
wing variables as keyword args: x, y. From version 0.12, the only valid positional argument will be `data`, and passin
g other arguments without an explicit keyword will result in an error or misinterpretation.
  FutureWarning
```

Out[34]:
```
<seaborn.axisgrid.JointGrid at 0x7f2dbec1d190>
```



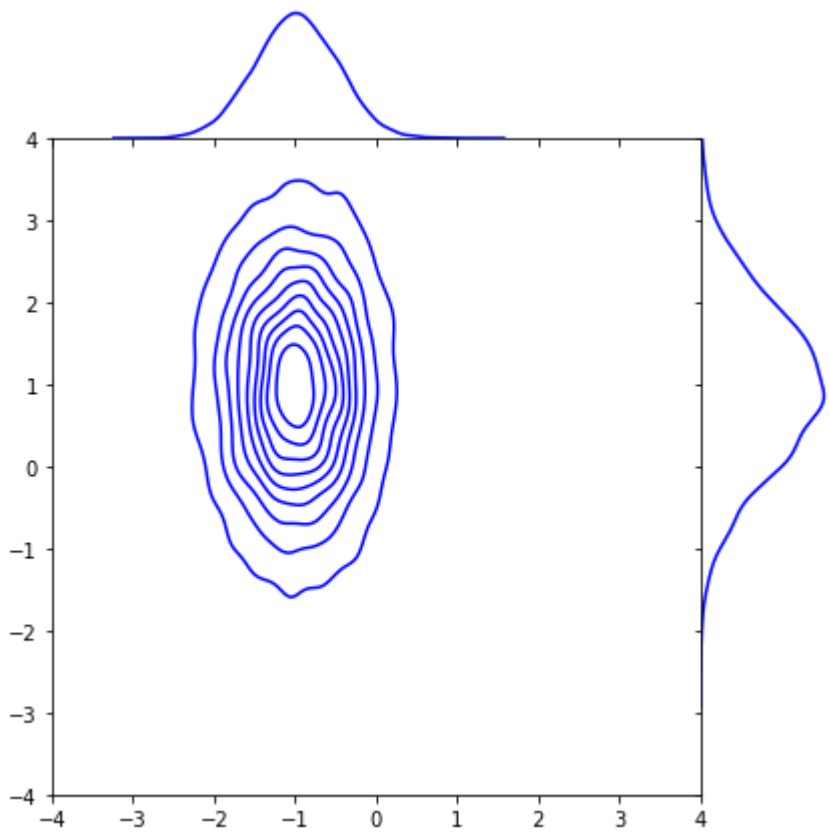
In [35]:
```
# Use MultivariateNormalDiag to create the equivalent distribution
# Note that diagonal covariance mat rix => no correlation => independence
# (for the multivariate normal distribution)
bivariate_normal_from_Multivariate = tfd.MultivariateNormalDiag(loc = locs, scale_diag = scales)
bivariate_normal_from_Multivariate
```

Out[35]:
```
<tfp.distributions.MultivariateNormalDiag 'MultivariateNormalDiag' batch_shape=[] event_shape=[2] dtype=float32>
```

In [36]:
```
# Plot the joint density function of bivariate_normal_from_Independent
# Refer back to bivariate_normal_from_Independent to show that the plot is the same
# Summarise how Independent has been used
samples = bivariate_normal_from_Multivariate.sample(10000)
x1 = samples[:, 0]
x2 = samples[:, 1]
sns.jointplot(x1, x2, kind = 'kde', space = 0, color = 'b', xlim = [-4, 4], ylim = [-4, 4])
```

```
/home/bacti/anaconda3/envs/tensor/lib/python3.7/site-packages/seaborn/_decorators.py:43: FutureWarning: Pass the follo
wing variables as keyword args: x, y. From version 0.12, the only valid positional argument will be `data`, and passin
g other arguments without an explicit keyword will result in an error or misinterpretation.
  FutureWarning
```

Out[36]:
```
<seaborn.axisgrid.JointGrid at 0x7f2dbec1d9d0>
```



Shifting batch dimensions to event dimensions using

`reinterpreted_batch_ndims`

In [37]:
```
# Demonstrate use of reinterpreted_batch_ndims
# By default all batch dims except the first are transferred to event dims
loc_grid = [[-100., -100.], [100., 100.], [0., 0.]]
scale_grid = [[1., 10.], [1., 10.], [1., 1.]]
normals_batch_3by2_event_1 = tfd.Normal(loc = loc_grid, scale = scale_grid)
```

In [38]:
```
# Highlight batch_shape
normals_batch_3by2_event_1
```

Out[38]:
```
<tfp.distributions.Normal 'Normal' batch_shape=[3, 2] event_shape=[] dtype=float32>
```

In [39]:
```
# We now have a batch of 3 bivariate normal distributions,
# each parametrised by a column of our original parameter grid
```

```
normals_batch_3_event_2 = tfd.Independent(normals_batch_3by2_event_1)
normals_batch_3_event_2
```

Out[39]:
```
<tfp.distributions.Independent 'IndependentNormal' batch_shape=[3] event_shape=[2] dtype=float32>
```

In [40]:
```
# Evaluate log_prob
normals_batch_3_event_2.log_prob(value = [[-10., 10.], [100., 100.], [1., 1.]])
```

Out[40]:
```
<tf.Tensor: shape=(3,), dtype=float32, numpy=array([-4.1146406e+03, -4.1404624e+00, -2.8378770e+00], dtype=float32)>
```

In [41]:
```
# Can reinterpret _all_ batch dimensions as event dimensions
normals_batch_1_event_3by2 = tfd.Independent \
    (normals_batch_3by2_event_1, reinterpreted_batch_ndims = 2)
normals_batch_1_event_3by2
```

Out[41]:
```
<tfp.distributions.Independent 'IndependentNormal' batch_shape=[] event_shape=[3, 2] dtype=float32>
```

In [42]:
```
# Take log_probs
normals_batch_1_event_3by2.log_prob(value = [[-10., 10.], [100., 100.], [1., 1.]])
```

Out[42]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=-4121.619>
```

Using `Independent` to build a Naive Bayes classifier

Introduction to `newsgroups` data set

In this tutorial, just load the dataset, fetch train/test splits, probably choose a subset of the data.

Construct the class conditional feature distribution (with Independent, using the Naive Bayes assumption) and sample from it.

We can just use the ML estimates for parameters, in later tutorials we will learn them.

In [43]:
```
# Convenience function for retrieving the 20 newsgroups data set

# Usenet was a forerunner to modern internet forums
# Users could post and read articles
# Newsgroup corresponded to a topic
# Example topics in this data set: IBM computer hardware, baseball
# Our objective is to use an article's contents to predict its newsgroup,
# a 20-class classification problem.

# 18000 newsgroups, posts on 20 topics
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import CountVectorizer
```

In [44]:
```
# Get the train data
newsgroups_data = fetch_20newsgroups(data_home = '20_Newsgroup_Data/', subset = 'train')
```

In [45]:
```
# More information about the data set
print(newsgroups_data['DESCR'][:1000])
```

```
.. _20newsgroups_dataset:

The 20 newsgroups text dataset
------------------------------

The 20 newsgroups dataset comprises around 18000 newsgroups posts on
20 topics split in two subsets: one for training (or development)
and the other one for testing (or for performance evaluation). The split
between the train and test set is based upon a messages posted before
and after a specific date.

This module contains two loaders. The first one,
:func:`sklearn.datasets.fetch_20newsgroups`,
returns a list of the raw texts that can be fed to text feature
extractors such as :class:`sklearn.feature_extraction.text.CountVectorizer`
with custom parameters so as to extract feature vectors.
The second one, :func:`sklearn.datasets.fetch_20newsgroups_vectorized`,
returns ready-to-use features, i.e., it is not necessary to use a feature
extractor.

**Data Set Characteristics:**

    =================   ==========
    Classes                     20
    Samples total            18846
    Dimensionality
```

In [46]:
```
# Example article
print(newsgroups_data['data'][0])
```

```
From: lerxst@wam.umd.edu (where's my thing)
Subject: WHAT car is this!?
Nntp-Posting-Host: rac3.wam.umd.edu
Organization: University of Maryland, College Park
Lines: 15

 I was wondering if anyone out there could enlighten me on this car I saw
the other day. It was a 2-door sports car, looked to be from the late 60s/
early 70s. It was called a Bricklin. The doors were really small. In addition,
the front bumper was separate from the rest of the body. This is
all I know. If anyone can tellme a model name, engine specs, years
of production, where this car is made, history, or whatever info you
```

```
have on this funky looking car, please e-mail.

Thanks,
- IL
   ---- brought to you by your neighborhood Lerxst ----
```

In [47]:
```
# Associated label
newsgroups_data['target'][0]
```

Out[47]: 7

In [48]:
```
# Name of label
newsgroups_data['target_names'][7]
```

Out[48]: 'rec.autos'

In [49]:
```
# Preprocessing boilerplate
n_documents = len(newsgroups_data['data'])
count_vectorizer = CountVectorizer \
    (input = 'content', binary = True, max_df = 0.25, min_df = 1.01 / n_documents) # ignore common words, words that

# input is a list of strings
binary_bag_of_words = count_vectorizer.fit_transform(newsgroups_data['data'])
```

In [50]:
```
# Check shape
binary_bag_of_words.shape
```

Out[50]: (11314, 56365)

In [51]:
```
# Check that the fit has been successful
count_vectorizer.inverse_transform(binary_bag_of_words[0, :])
```

Out[51]:
```
[array(['lerxst', 'wam', 'umd', 'where', 'thing', 'car', 'rac3',
        'maryland', 'college', 'park', '15', 'wondering', 'anyone',
        'could', 'enlighten', 'saw', 'day', 'door', 'sports', 'looked',
        'late', '60s', 'early', '70s', 'called', 'bricklin', 'doors',
        'were', 'really', 'small', 'addition', 'front', 'bumper',
        'separate', 'rest', 'body', 'tellme', 'model', 'name', 'engine',
        'specs', 'years', 'production', 'made', 'history', 'whatever',
        'info', 'funky', 'looking', 'please', 'mail', 'thanks', 'il',
        'brought', 'neighborhood'], dtype='<U80')]
```

In [52]:
```
# Dict that will be useful later
inv_vocabulary = {value:key for key, value in count_vectorizer.vocabulary_.items()}
```

A Naive Bayes classifier for `newsgroup`

Each feature vector x is a list of indicators for whether a word appears in the article. x_i is 1 if the ith word appears, and 0 otherwise.

`inv_vocabulary` matches word indices i to words.

Each label y is a value in $0, 1, \ldots, 19$.

The parts of a naive Bayes classifier for this problem can be summarised as:

- A probability distribution for the feature vector by class, $p(x \mid y = j)$ for each $j = 0, 1, \ldots, 19$. These probability distributions are assumed to have independent components: we can factorize the joint probability as a product of marginal probabilities

$$p(x \mid y = j) = \prod_{i=1}^{d} p(x_i \mid y = j)$$

  These marginal probability distributions are Bernoulli distributions, each of which has a single parameter $\theta_{ji} := p(x_i = 1 \mid y = j)$. This parameter is the probability of observing word i in an article of class j.

- We will use the Laplace smoothed maximum likelihood estimate to compute these parameters. Laplace smoothing involves adding small counts to every feature for each class. Else, if a feature did not appear in the training set of a class, but then we observed it in our test data the log probability would be undefined.

- A collection of class prior probabilities $p(y = j)$. These will be set by computing the class base rates in the training set.

- A function for computing the probability of class membership via Bayes' theorem:

$$p(y = j \mid x) = \frac{p(x \mid y = j) p(y = j)}{p(x)}$$

In [53]:
```
# Compute the parameter estimates (adjusted fraction of documents in class that contain word)
n_classes = newsgroups_data['target'].max() + 1
y = newsgroups_data['target']
n_words = binary_bag_of_words.shape[1]

alpha = 1e-6 # parameters for Laplace smoothing

theta = np.zeros([n_classes, n_words]) # stores parameter values - prob. word given class
```

```
for c_k in range(n_classes): # 0, 1, ..., 19
    class_mask = (y == c_k)
    N = class_mask.sum() # number of articles in class
    theta[c_k, :] = (binary_bag_of_words[class_mask, :].sum(axis = 0) + alpha) / (N + alpha * 2)
```

In [54]:
```
# Check whether the most probable word in each class is reasonable
most_probable_word_ix = theta.argmax(axis = 1) # most probable word for each class

for j, ix in enumerate(most_probable_word_ix):
    print (
        'Most probable word in class {} is "{}".'.format \
        (newsgroups_data['target_names'][j], inv_vocabulary[ix])
    )
```

```
Most probable word in class alt.atheism is "people".
Most probable word in class comp.graphics is "graphics".
Most probable word in class comp.os.ms-windows.misc is "windows".
Most probable word in class comp.sys.ibm.pc.hardware is "thanks".
Most probable word in class comp.sys.mac.hardware is "mac".
Most probable word in class comp.windows.x is "window".
Most probable word in class misc.forsale is "sale".
Most probable word in class rec.autos is "car".
Most probable word in class rec.motorcycles is "dod".
Most probable word in class rec.sport.baseball is "he".
Most probable word in class rec.sport.hockey is "ca".
Most probable word in class sci.crypt is "clipper".
Most probable word in class sci.electronics is "use".
Most probable word in class sci.med is "reply".
Most probable word in class sci.space is "space".
Most probable word in class soc.religion.christian is "god".
Most probable word in class talk.politics.guns is "people".
Most probable word in class talk.politics.mideast is "people".
Most probable word in class talk.politics.misc is "people".
Most probable word in class talk.religion.misc is "he".
```

In [55]:
```
# Define a distribution for each class
batch_of_bernoullis = tfd.Bernoulli(probs = theta)
p_x_given_y = tfd.Independent(batch_of_bernoullis, reinterpreted_batch_ndims = 1)
p_x_given_y
```

Out[55]:
```
<tfp.distributions.Independent 'IndependentBernoulli' batch_shape=[20] event_shape=[56365] dtype=int32>
```

In [56]:
```
# Take a sample of words from each class
n_samples = 10
sample = p_x_given_y.sample(n_samples)
sample.shape
```

Out[56]:
```
TensorShape([10, 20, 56365])
```

In [57]:
```
# Choose a class
chosen_class = 15
newsgroups_data['target_names'][chosen_class]
```

Out[57]:
```
'soc.religion.christian'
```

In [58]:
```
# Indicators for words that appear in the sample
class_sample = sample[:, chosen_class, :]
class_sample
```

Out[58]:
```
<tf.Tensor: shape=(10, 56365), dtype=int32, numpy=
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]], dtype=int32)>
```

In [59]:
```
# Perform inverse transform to test quality of fit
count_vectorizer.inverse_transform(class_sample)[0]
```

Out[59]:
```
array(['09', '19', '23', '28', '31', '34', '35', '41', '78228', 'acs',
       'actually', 'advise', 'ages', 'also', 'andrew', 'anyone',
       'anything', 'appease', 'apr', 'bathroom', 'bc', 'because',
       'beggar', 'believe', 'better', 'bigelow', 'brains', 'chest',
       'child', 'claim', 'cleaned', 'commented', 'concepts', 'country',
       'day', 'daycare', 'death', 'decided', 'definition', 'detailed',
       'difference', 'disciple', 'discuss', 'does', 'emotional',
       'encourage', 'even', 'excellent', 'explain', 'faith', 'familiar',
       'fellowship', 'forgiven', 'free', 'garnet', 'gentiles', 'go',
       'god', 'good', 'growing', 'guess', 'hear', 'heard', 'here', 'him',
       'hold', 'honoring', 'however', 'ignore', 'kind', 'knowing', 'last',
       'lawrence', 'least', 'legal', 'less', 'life', 'minds', 'moment',
       'monotheism', 'must', 'never', 'now', 'offers', 'original', 'our',
       'over', 'pa', 'path', 'perception', 'person', 'personally', 'phil',
       'political', 'predestination', 'previously', 'problem', 'problems',
       'prove', 'punish', 'questionnaire', 'qui', 'quite', 'quote',
       'quotes', 'reading', 'real', 'reasonable', 'requirement', 'result',
       'reversed', 'say', 'second', 'sense', 'shaken', 'small', 'sons',
       'souls', 'span', 'speaking', 'spiritually', 'ssd', 'statements',
       'stem', 'still', 'study', 'take', 'talk', 'testament', 'than',
       'their', 'them', 'then', 'theories', 'these', 'things', 'think',
       'through', 'trying', 'two', 'uk', 'understand', 'us', 'uu4',
       'verse', 'very', 'views', 'vituperousness', 'wasn', 'watt', 'way',
       'well', 'whose', 'why', 'willing', 'wisdom', 'words', 'world'],
      dtype='<U80')
```

Sampling and log probs

In [60]:
```
# Make Multivariate Distribution
normal_distributions = tfd.MultivariateNormalDiag (
    loc = [[0.5, 1], [0.1, 0], [0, 0.2]],
    scale_diag = [[2, 3], [1, 3], [4, 4]]
)
normal_distributions
```

Out[60]:
```
<tfp.distributions.MultivariateNormalDiag 'MultivariateNormalDiag' batch_shape=[3] event_shape=[2] dtype=float32>
```

In [61]:
```
# Sample
normal_distributions.sample(5)
```

Out[61]:
```
<tf.Tensor: shape=(5, 3, 2), dtype=float32, numpy=
array([[[-0.61642456, -2.6508875 ],
        [-0.02484179, -0.58626443],
        [-2.384066  , -2.2478259 ]],

       [[-1.3702699 ,  9.627965  ],
        [ 0.34466884, -2.335009  ],
        [-0.6538167 ,  5.7416673 ]],

       [[-2.086178  ,  4.7825446 ],
        [ 1.154147  , -4.501431  ],
        [-4.3120627 ,  0.32013714]],

       [[-2.5057547 ,  2.4638143 ],
        [-0.11568717, -5.0724864 ],
        [ 0.63433516, -0.52497053]],

       [[ 1.380878  ,  1.7464799 ],
        [ 0.11383934,  1.1534938 ],
        [-0.91430813,  2.0012252 ]]], dtype=float32)>
```

In [62]:
```
# Multivariate Normal batched Distribution
# We are broadcasting batch shapes of `loc` and `scal_diag`
# against each other
loc = [[[0.3, 1.5, 1.], [0.2, 0.4, 2.8]], [[2., 2.3, 8], [1.4, 1, 1.3]]]
scale_diag = [0.4, 1., 0.7]
normal_distributions = tfd.MultivariateNormalDiag(loc = loc, scale_diag = scale_diag)
normal_distributions
```

Out[62]:
```
<tfp.distributions.MultivariateNormalDiag 'MultivariateNormalDiag' batch_shape=[2, 2] event_shape=[3] dtype=float32>
```

In [63]:
```
# Use independent to move part of the batch shape
ind_normal_distributions = tfd.Independent(normal_distributions, reinterpreted_batch_ndims = 1)
ind_normal_distributions
```

Out[63]:
```
<tfp.distributions.Independent 'IndependentMultivariateNormalDiag' batch_shape=[2] event_shape=[2, 3] dtype=float32>
```

In [64]:
```
# Draw some samples
samples = ind_normal_distributions.sample(5)
samples.shape
```

Out[64]:
```
TensorShape([5, 2, 2, 3])
```

In [65]:
```
# `[B, E]` shaped input
inp = tf.random.uniform((2, 2, 3))
ind_normal_distributions.log_prob(inp)
```

Out[65]:
```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([-11.756409, -66.098785], dtype=float32)>
```

In [66]:
```
# `[E]` shaped input (broadcasting over batch size)
inp = tf.random.uniform((2, 3))
ind_normal_distributions.log_prob(inp)
```

Out[66]:
```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([-11.850761, -76.12896 ], dtype=float32)>
```

In [67]:
```
#`[S, B, E]` shaped input (broadcasting over samples)
inp = tf.random.uniform((9, 2, 2, 3))
ind_normal_distributions.log_prob(inp)
```

Out[67]:
```
<tf.Tensor: shape=(9, 2), dtype=float32, numpy=
array([[ -8.763502 , -74.403946 ],
       [ -8.963752 , -78.71576  ],
       [-10.053402 , -67.10482  ],
       [-10.113637 , -70.84384  ],
       [ -8.458472 , -77.99654  ],
       [-13.988488 , -74.964966 ],
       [ -7.66702  , -65.602325 ],
       [-10.51793  , -65.675446 ],
       [-11.9138565, -72.61048  ]], dtype=float32)>
```

In [68]:
```
# `[S, b, e]` shaped input, where [b, e] is broadcastable over [B, E]
inp = tf.random.uniform((5, 1, 2, 1))
ind_normal_distributions.log_prob(inp)
```

Out[68]:
```
<tf.Tensor: shape=(5, 2), dtype=float32, numpy=
array([[ -9.843089 , -61.68399  ],
```

```
       [-10.387953 , -61.109745 ],
       [ -9.2092285, -71.7383   ],
       [-12.003781 , -83.812454 ],
       [ -9.123208 , -68.52439  ]], dtype=float32)>
```

Naive Bayes example

Lets now use what we have learned and continue the Naive Bayes classifier we were building last tutorial.

In [69]:
```
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import BernoulliNB
from sklearn.metrics import f1_score
```

In [70]:
```
# Making a function get_data which:
#   1) Fetches the 20 newsgroup dataset
#   2) Performs a word count on the articles and binarizes the result
#   3) Returns the data as a numpy matrix with the labels
def get_data(categories):

    newsgroups_train_data = fetch_20newsgroups \
        (data_home = '20_Newsgroup_Data/', subset = 'train', categories = categories)
    newsgroups_test_data = fetch_20newsgroups \
        (data_home = '20_Newsgroup_Data/', subset = 'test', categories = categories)

    n_documents = len(newsgroups_train_data['data'])
    count_vectorizer = CountVectorizer \
        (input = 'content', binary = True, max_df = 0.25, min_df = 1.01 / n_documents)

    train_binary_bag_of_words = count_vectorizer.fit_transform(newsgroups_train_data['data'])
    test_binary_bag_of_words = count_vectorizer.transform(newsgroups_test_data['data'])

    return (train_binary_bag_of_words.todense(), newsgroups_train_data['target']), \
           (test_binary_bag_of_words.todense(), newsgroups_test_data['target'])
```

In [71]:
```
# Defining a function to conduct Laplace smoothing.
# This adds a base level of probability for a given feature to occur in every class.
def laplace_smoothing(labels, binary_data, n_classes):
    # Compute the parameter estimates
    # (adjusted fraction of documents in class that contain word)
    n_words = binary_data.shape[1]
    alpha = 1 # parameters for Laplace smoothing
    theta = np.zeros([n_classes, n_words]) # stores parameter values - prob. word given class
    for c_k in range(n_classes): # 0, 1, ..., 19
        class_mask = (labels == c_k)
        N = class_mask.sum() # number of articles in class
        theta[c_k, :] = (binary_data[class_mask, :].sum(axis = 0) + alpha) / (N + alpha * 2)

    return theta
```

In [72]:
```
# Getting a subset of the 20 newsgroup dataset
categories = ['alt.atheism', 'talk.religion.misc', 'comp.graphics', 'sci.space']

(train_data, train_labels), (test_data, test_labels) = get_data(categories = categories)
smoothed_counts = laplace_smoothing \
    (labels = train_labels, binary_data = train_data, n_classes = len(categories))
```

To now make our NB classifier we need to build three functions:

- Compute the class priors
- Build our class conditional distributions
- Put it all together and classify our data

In [73]:
```
# Function which computes the prior probability of every class
# based on frequency of occurence in the dataset
def class_priors(n_classes, labels):
    counts = np.zeros(n_classes)
    for c_k in range(n_classes):
        counts[c_k] = np.sum(np.where(labels == c_k, 1, 0))
    priors = counts / np.sum(counts)
    print('The class priors are {}'.format(priors))
    return priors
```

In [74]:
```
# Run the function
priors = class_priors(n_classes = len(categories), labels = train_labels)
```

```
The class priors are [0.2359882  0.28711898 0.29154376 0.18534907]
```

In [75]:
```
# Now we will do a function that given the feature occurence counts returns a Bernoulli distribution of
# batch_shape=number of classes and event_shape=number of features.
def make_distribution(probs):
    batch_of_bernoullis = tfd.Bernoulli(probs = probs)
    dist = tfd.Independent(batch_of_bernoullis, reinterpreted_batch_ndims = 1)
    return dist

tf_dist = make_distribution(smoothed_counts)
tf_dist
```

Out[75]:
```
<tfp.distributions.Independent 'IndependentBernoulli' batch_shape=[4] event_shape=[17495] dtype=int32>
```

In [76]:
```
# The final function predict_sample which given the distribution, a test sample, and the class priors:
#   1) Computes the class conditional probabilities given the sample
```

```
#   2) Forms the joint likelihood
#   3) Normalises the joint likelihood and returns the log prob

def predict_sample(dist, sample, priors):
    cond_probs = dist.log_prob(sample)
    joint_likelihood = tf.add(np.log(priors), cond_probs)
    norm_factor = tf.math.reduce_logsumexp(joint_likelihood, axis = -1, keepdims = True)
    log_prob = joint_likelihood - norm_factor

    return log_prob
```

Computing log_probs

In [77]:
```
# Predicting one example from our test data
log_probs = predict_sample(tf_dist, test_data[0], priors)
log_probs
```

Out[77]:
```
<tf.Tensor: shape=(4,), dtype=float32, numpy=
array([-6.1736343e+01, -1.5258789e-05, -1.1620026e+01, -6.3327866e+01],
      dtype=float32)>
```

In [78]:
```
# Loop over our test data and classify.
probabilities = []
for sample, label in zip(test_data, test_labels):
    probabilities.append(tf.exp(predict_sample(tf_dist, sample, priors)))

probabilities = np.asarray(probabilities)
predicted_classes = np.argmax(probabilities, axis = -1)
print('f1 ', f1_score(test_labels, predicted_classes, average = 'macro'))
```

```
f1  0.7848499112849504
```

In [79]:
```
# Make a Bernoulli Naive Bayes classifier using sklearn with the same level of alpha smoothing.
clf = BernoulliNB(alpha = 1)
clf.fit(train_data, train_labels)
pred = clf.predict(test_data)
print('f1 from sklean ', f1_score(test_labels, pred, average = 'macro'))
```

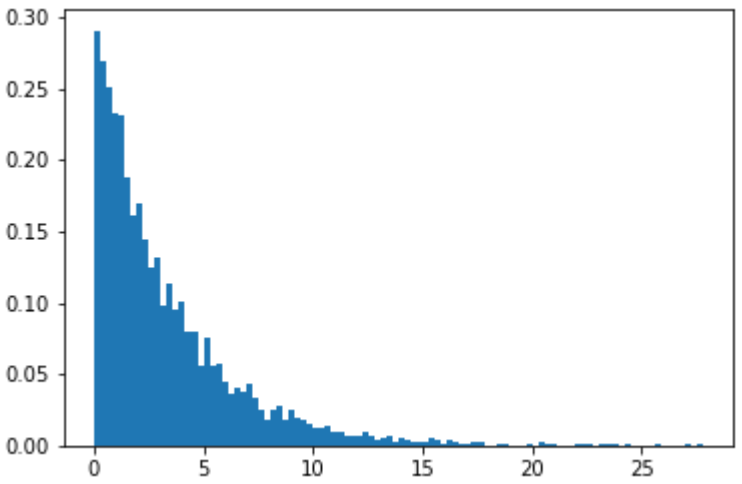
```
f1 from sklean  0.7848499112849504
```

Trainable Distributions

In [80]:
```
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import BernoulliNB
from sklearn.metrics import f1_score
```

In [81]:
```
# Define an exponential distribution
exponential = tfd.Exponential(rate = 0.3, name = 'exp')
```

In [82]:
```
# Plot
plt.hist(exponential.sample(5000).numpy(), bins = 100, density = True)
plt.show()
```



In [83]:
```
# Define an exponential distribution with a trainable rate parameter
exp_train = tfd.Exponential(rate = tf.Variable(1., name = 'rate'), name = 'exp_train')
exp_train.trainable_variables
```

Out[83]:
```
(<tf.Variable 'rate:0' shape=() dtype=float32, numpy=1.0>,)
```

In [84]:
```
# Define the negative log likelihood
def nll(x_train, distribution):
    return -tf.reduce_mean(distribution.log_prob(x_train))
```

In [85]:
```
# Define a function to compute the loss and gradients
@tf.function
def get_loss_and_grads(x_train, distribution):
    with tf.GradientTape() as tape:
        tape.watch(distribution.trainable_variables)
        loss = nll(x_train, distribution)
        grads = tape.gradient(loss, distribution.trainable_variables)
    return loss, grads
```

In [86]:
```
# Optimize
def exponential_dist_optimisation(data, distribution):

    # Keep results for plotting
    train_loss_results = []
    train_rate_results = []
    optimizer = tf.keras.optimizers.SGD(learning_rate = 0.05)
    num_steps = 10

    for i in range(num_steps):
        loss, grads = get_loss_and_grads(data, distribution)
        optimizer.apply_gradients(zip(grads, distribution.trainable_variables))

        rate_value = distribution.rate.value()
        train_loss_results.append(loss)
        train_rate_results.append(rate_value)
        print("Step {:03d}: Loss: {:.3f}: Rate: {:.3f}".format(i, loss, rate_value))

    return train_loss_results, train_rate_results
```

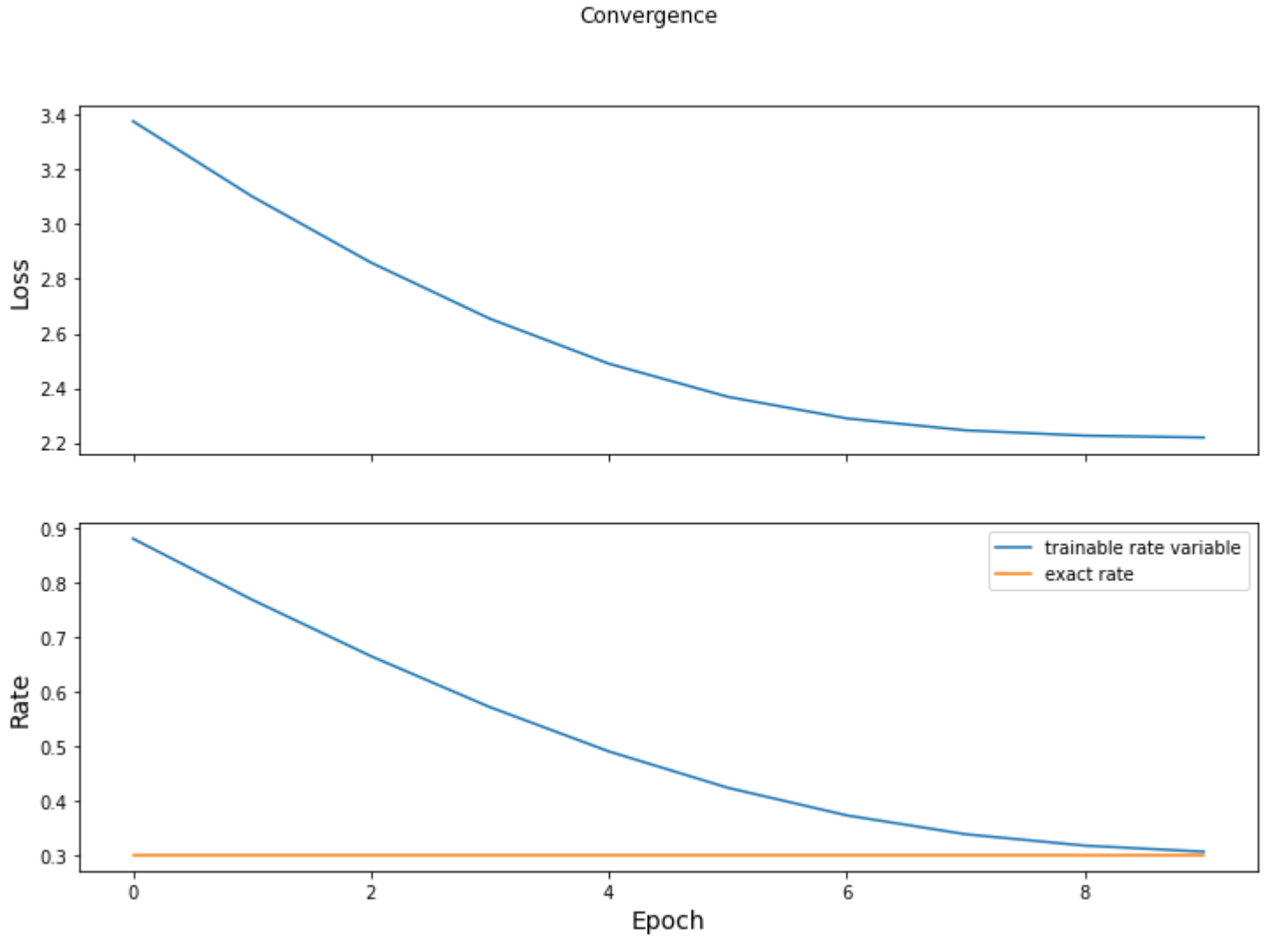
In [87]:
```
# Get some data and train
sampled_data = exponential.sample(5000)
train_loss_results, train_rate_results = exponential_dist_optimisation \
    (data = sampled_data, distribution = exp_train)
```

```
Step 000: Loss: 3.377: Rate: 0.881
Step 001: Loss: 3.102: Rate: 0.769
Step 002: Loss: 2.860: Rate: 0.665
Step 003: Loss: 2.654: Rate: 0.572
Step 004: Loss: 2.490: Rate: 0.490
Step 005: Loss: 2.368: Rate: 0.423
Step 006: Loss: 2.289: Rate: 0.373
Step 007: Loss: 2.246: Rate: 0.338
Step 008: Loss: 2.226: Rate: 0.317
Step 009: Loss: 2.219: Rate: 0.306
```

In [88]:
```
# Predicted value for the rate parameter
pred_value = exp_train.rate.numpy()
exact_value = exponential.rate.numpy()

print('Exact rate: ', exact_value)
print('Pred rate:  ', pred_value)
```

```
Exact rate:  0.3
Pred rate:   0.3058872
```

In [89]:
```
# Plot to see the convergence of the estimated and true parameters
tensor_exact_value = tf.constant(exact_value, shape = [len(train_rate_results)])

fig, axes = plt.subplots(2, sharex = True, figsize = (12, 8))
fig.suptitle('Convergence')

axes[0].set_ylabel("Loss", fontsize = 14)
axes[0].plot(train_loss_results)

axes[1].set_ylabel("Rate", fontsize = 14)
axes[1].set_xlabel("Epoch", fontsize = 14)
axes[1].plot(train_rate_results, label = 'trainable rate variable')
axes[1].plot(tensor_exact_value, label = 'exact rate')
axes[1].legend()
plt.show()
```



In [90]:
```
# Making a function get_data which:
#   1) Fetches the 20 newsgroup dataset
#   2) Performs a word count on the articles and binarizes the result
#   3) Returns the data as a numpy matrix with the labels
def get_data(categories):

    newsgroups_train_data = fetch_20newsgroups \
        (data_home = '20_Newsgroup_Data/', subset = 'train', categories = categories)
    newsgroups_test_data = fetch_20newsgroups \
        (data_home = '20_Newsgroup_Data/', subset = 'test', categories = categories)

    n_documents = len(newsgroups_train_data['data'])
    count_vectorizer = CountVectorizer \
        (input = 'content', binary = True, max_df = 0.25, min_df = 1.01 / n_documents)
    train_binary_bag_of_words = count_vectorizer.fit_transform(newsgroups_train_data['data'])
    test_binary_bag_of_words = count_vectorizer.transform(newsgroups_test_data['data'])

    return (train_binary_bag_of_words.todense(), newsgroups_train_data['target']), \
           (test_binary_bag_of_words.todense(), newsgroups_test_data['target'])
```

In [91]:
```
# Defining a function to conduct laplace smoothing. This adds a base level of probability for a given feature
# to occur in every class.
def laplace_smoothing(labels, binary_data, n_classes):
    # Compute the parameter estimates (adjusted fraction of documents in class that contain word)
    n_words = binary_data.shape[1]
    alpha = 1 # parameters for Laplace smoothing
    theta = np.zeros([n_classes, n_words]) # stores parameter values - prob. word given class
    for c_k in range(n_classes): # 0, 1, ..., 19
        class_mask = (labels == c_k)
        N = class_mask.sum() # number of articles in class
        theta[c_k, :] = (binary_data[class_mask, :].sum(axis = 0) + alpha) / (N + alpha * 2)

    return theta
```

In [92]:
```
# Now we will do a function that given the feature occurence counts returns a Bernoulli distribution of
# batch_shape=number of classes and event_shape=number of features.
def make_distributions(probs):
    batch_of_bernoullis = tfd.Bernoulli(probs = probs) # shape (n_classes, n_words)
    dist = tfd.Independent(batch_of_bernoullis, reinterpreted_batch_ndims = 1)
    return dist
```

In [93]:
```
# Function which computes the prior probability of every class based on frequency of occurence in
# the dataset
def class_priors(n_classes, labels):
    counts = np.zeros(n_classes)
    for c_k in range(n_classes):
        counts[c_k] = np.sum(np.where(labels == c_k, 1, 0))
    priors = counts / np.sum(counts)
    print('The class priors are {}'.format(priors))
    return priors
```

In [94]:
```
# The final function predict_sample which given the distribution, a test sample, and the class priors:
#   1) Computes the class conditional probabilities given the sample
#   2) Forms the joint likelihood
#   3) Normalises the joint likelihood and returns the log prob
def predict_sample(dist, sample, priors):
    cond_probs = dist.log_prob(sample)
    joint_likelihood = tf.add(np.log(priors), cond_probs)
    norm_factor = tf.math.reduce_logsumexp(joint_likelihood, axis = -1, keepdims = True)
    log_prob = joint_likelihood - norm_factor
    return log_prob
```

In [95]:
```
# Now we learn the distribution using gradient tape
def make_distribution_withGT(data, labels, nb_classes):

    class_data = []
    train_vars = []
    distributions = []
    for c in range(nb_classes):
        train_vars.append(tf.Variable \
            (initial_value=np.random.uniform(low = 0.01, high = 0.1, size = data.shape[-1])))
        distributions.append(tfd.Bernoulli(probs = train_vars[c]))
        class_mask = (labels == c)
        class_data.append(data[class_mask, :])

    for c_num in range(0,nb_classes):
        optimizer = tf.keras.optimizers.Adam()
        print('\n%------------------%')
        print('Class ', c_num)
        print('%------------------%')

        for i in range(0, 100):
            loss, grads = get_loss_and_grads(class_data[c_num], distributions[c_num])
            if i % 10 == 0:
                print('iter: {} loss: {}'.format(i, loss))
            optimizer.apply_gradients(zip(grads, distributions[c_num].trainable_variables))
            eta = 1e-3
            clipped_probs = tf.clip_by_value \
                (distributions[c_num].trainable_variables, clip_value_min = eta, clip_value_max = 1)
            train_vars[c_num] = tf.squeeze(clipped_probs)

    dist = tfd.Bernoulli(probs = train_vars)
    dist = tfd.Independent(dist, reinterpreted_batch_ndims = 1)
```

```
    print(dist)
    return dist
```

In [96]:
```
# Make the same Naive Bayes classifier we did last tutorial
categories = ['alt.atheism', 'talk.religion.misc', 'comp.graphics', 'sci.space']
(train_data, train_labels), (test_data, test_labels) = get_data(categories)

smoothed_counts = laplace_smoothing \
    (labels = train_labels, binary_data = train_data, n_classes = len(categories))
priors = class_priors(n_classes = len(categories), labels = train_labels)
tf_dist = make_distributions(smoothed_counts)
```

```
The class priors are [0.2359882  0.28711898 0.29154376 0.18534907]
```

In [97]:
```
# Now train the distributions with gradient tape
GT_dist = make_distribution_withGT(data = train_data, labels = train_labels, nb_classes = 4)
```

```
%------------------%
Class  0
%------------------%
iter: 0 loss: 0.07829254456005681
iter: 10 loss: 0.06899232657607354
iter: 20 loss: 0.06031953688434695
iter: 30 loss: 0.05228091542158744
iter: 40 loss: 0.044827120954877314
iter: 50 loss: 0.037923124902828766
iter: 60 loss: 0.03153174284920682
iter: 70 loss: 0.02562061911389209
iter: 80 loss: 0.020148650299908685
iter: 90 loss: 0.015078609758130055

%------------------%
Class  1
%------------------%
iter: 0 loss: 0.07146276037880028
iter: 10 loss: 0.06212357453199265
iter: 20 loss: 0.05334094127326725
iter: 30 loss: 0.0451854947433399
iter: 40 loss: 0.03763059468595021
iter: 50 loss: 0.030644072872020898
iter: 60 loss: 0.024187183809748204
iter: 70 loss: 0.018231306796357812
iter: 80 loss: 0.01273630114869445
iter: 90 loss: 0.00764733326735143

%------------------%
Class  2
%------------------%
iter: 0 loss: 0.07831977980297099
iter: 10 loss: 0.06920893765933767
iter: 20 loss: 0.060811155571121364
iter: 30 loss: 0.05315205171401028
iter: 40 loss: 0.04618726901559432
iter: 50 loss: 0.03988330075349113
iter: 60 loss: 0.034195962461422966
iter: 70 loss: 0.029088565905427646
iter: 80 loss: 0.02450239545572051
iter: 90 loss: 0.020364445161358293

%------------------%
Class  3
%------------------%
iter: 0 loss: 0.07956402386025159
iter: 10 loss: 0.0703239091751621
iter: 20 loss: 0.06169454189197293
iter: 30 loss: 0.05368818463089667
iter: 40 loss: 0.04627427900950802
iter: 50 loss: 0.039410224060753805
iter: 60 loss: 0.03304793772691462
iter: 70 loss: 0.027140914093159326
iter: 80 loss: 0.021659204942776244
iter: 90 loss: 0.016548925857701124
tfp.distributions.Independent("IndependentBernoulli", batch_shape=[4], event_shape=[17495], dtype=int32)
```

In [98]:
```
# Compare the two results
for dist in [GT_dist,tf_dist]:
    probabilities = []
    for sample, label in zip(test_data, test_labels):
        probabilities.append(predict_sample(dist, sample, priors))

    probabilities = np.asarray(probabilities)
    predicted_classes = np.argmax(probabilities, axis = -1)
    print('f1 ', f1_score(test_labels, predicted_classes, average = 'macro'))
```

```
f1  0.8331666344676952
f1  0.7848499112849504
```

Maximum likelihood estimation

This reading is a review of maximum likelihood estimation (MLE), an important learning principle used in neural network training.

Introduction

Why are neural networks trained the way they are? For example, why do you use a mean squared error loss function for a regression task, but a sparse categorical crossentropy loss for classification? The answer lies in the *likelihood* function, with a long history in statistics. In this reading, we'll look at what this function is and how it leads to the loss functions used to train deep learning models.

Since you're taking a course in Tensorflow Probability, I'll assume you already have some understanding of probability distributions, both discrete and continous. If you don't, there are countless resources to help you understand them. I find the Wikipedia page (https://en.wikipedia.org/wiki/Probability_distribution) works well for an intuitive introduction. For a more solid mathematical description, see an introductory statistics course.

Probability mass and probability density functions

Every probability distribution has either a probability mass function (if the distribution is discrete) or a probability density function (if the distribution is continuous). This function roughly indicates the probability of a sample taking a particular value. We will denote this function $P(y \mid \theta)$ where y is the value of the sample and θ is the parameter describing the probability distribution. Written out mathematically, we have:

$$P(y \mid \theta) = \text{Prob(sampling value } y \text{ from a distribution with parameter } \theta).$$

When more than one sample is drawn *independently* from the same distribution (which we usually assume), the probability mass/density function of the sample values $y_1, \ldots, y_n$ is the product of the probability mass/density functions for each individual y_i. Written formally:

$$P(y_1, \ldots, y_n \mid \theta) = \prod_{i=1}^{n} P(y_i \mid \theta).$$

This all sounds more complicated than it is: see the examples below for a more concrete illustration.

The likelihood function

Probability mass/density functions are usually considered functions of $y_1, \ldots, y_n$, with the parameter θ considered fixed. They are used when you know the parameter θ and want to know the probability of a sample taking some values $y_1, \ldots, y_n$. You use this function in *probability*, where you know the distribution and want to make deductions about possible values sampled from it.

The *likelihood* function is the same, but with the $y_1, \ldots, y_n$ considered fixed and with θ considered the independent variable. You usually use this function when you know the sample values $y_1, \ldots, y_n$ (because you've observed them by collecting data), but don't know the parameter θ. You use this function in *statistics*, where you know the data and want to make inferences about the distribution they came from.

This is an important point, so I'll repeat it: $P(y_1, \ldots, y_n \mid \theta)$ is called the *probability mass/density function* when considered as a function of $y_1, \ldots, y_n$ with θ fixed. It's called the *likelihood* when considered as a function of θ with $y_1, \ldots, y_n$ fixed. For the likelihood, the convention is using the letter L, so that

$$\underbrace{L(y_1, \ldots, y_n \mid \theta)}_{\text{likelihood, function of } \theta} = \underbrace{P(y_1, \ldots, y_n \mid \theta)}_{\text{probabiliy mass/density, function of } y_1, \ldots, y_n}$$

Let's see some examples of this below.

Bernoulli distribution

We'll start by looking at the Bernoulli distribution (https://en.wikipedia.org/wiki/Bernoulli_distribution) with parameter θ. It's the distribution of a random variable that takes value 1 with probability θ and 0 with probability $1 - \theta$. Let $P(y \mid \theta)$ be the probability that the event returns value y given parameter θ. Then

$$\begin{aligned} L(y \mid \theta) = P(y \mid \theta) &= \begin{cases} 1 - \theta & \text{if } y = 0 \\ \theta & \text{if } y = 1 \end{cases} \\ &= (1 - \theta)^{1-y}\theta^{y} \quad y \in \{0, 1\} \end{aligned}$$

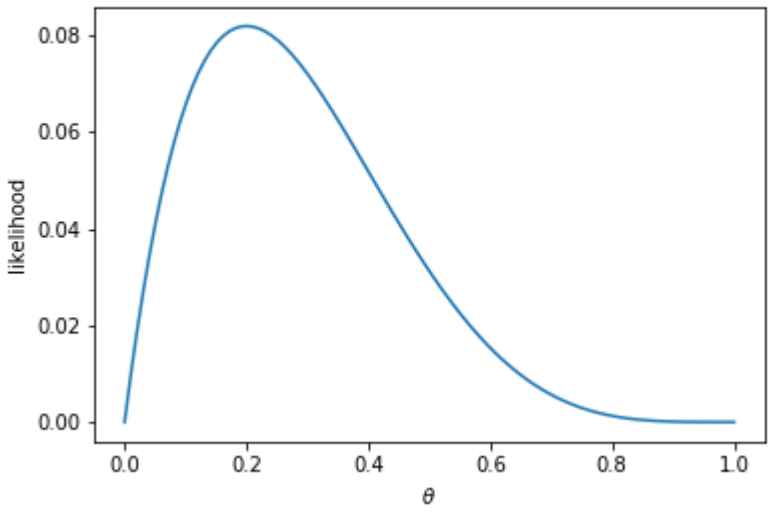
If we assume samples are independent, we also have

$$L(y_1, \ldots, y_n \mid \theta) = \prod_{i=1}^{n} (1 - \theta)^{1-y_i}\theta^{y_i}.$$

For example, the probability of observing $0, 0, 0, 1, 0$ is

$$L(0, 0, 0, 1, 0 \mid \theta) = (1 - \theta)(1 - \theta)(1 - \theta)\theta(1 - \theta) = \theta(1 - \theta)^4.$$

Note that, in this case, we have fixed the data, and are left with a function just of θ. This is called the *likelihood* function. Let's plot the likelihood as a function of θ below.



Normal (Gaussian) distribution

This idea also generalises naturally to the Normal distribution (https://en.wikipedia.org/wiki/Normal_distribution) (also called the *Gaussian* distribution). This distribution has two parameters: a mean μ and a standard deviation σ. We hence let $\theta = (\mu, \sigma)$. The probability density function (the analogue of the probability mass function for continuous distributions) is:

$$L(y \mid \theta) = P(y \mid \theta) = P(y \mid \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\Big(-\frac{1}{2\sigma^2}(y-\mu)^2 \Big).$$

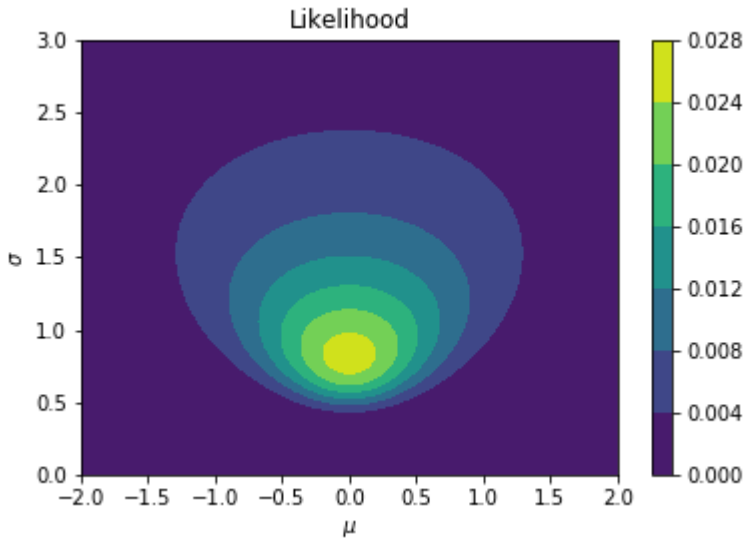
For a sequence of independent observations $y_1, \ldots, y_n$, the likelihood is

$$L(y_1, \ldots, y_n \mid \mu, \sigma) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\Big(-\frac{1}{2\sigma^2}(y_i-\mu)^2 \Big).$$

The *likelihood* is hence the same, but viewed as a function of μ and σ, with $y_1, \ldots, y_n$ viewed as constants. For example, if the observed data is -1, 0, 1, the likelihood becomes

$$L(-1, 0, 1 \mid \mu, \sigma) = (2\pi\sigma^2)^{-3/2} \exp\Big(-\frac{1}{2\sigma^2}\big((\mu-1)^2 + (\mu)^2 + (\mu+1)^2\big) \Big).$$

which we can plot as a function of μ an σ below.



Maximum likelihood estimation

The likelihood function is commonly used in statistical inference when we are trying to fit a distribution to some data. This is usually done as follows. Suppose we have observed data $y_1, \ldots, y_n$, assumed to be from some distribution with unknown parameter θ, which we want to estimate. The likelihood is

$$L(y_1, \ldots, y_n \mid \theta).$$

The *maximum likelihood estimate* θ_{MLE} of the parameter θ is then the value that maximises the likelihood $L(y_1, \ldots, y_n \mid \theta)$. For the example of the Bernoulli distribution with observed data 0, 0, 0, 1, 0 (as in the plot above), this gives us $p = \frac{1}{5}$, which is where the plot takes its maximum. For the normal distribution with data -1, 0, 1, this is the region where the plot is brightest (indicating the highest value), and this occurs at $\mu = 0, \sigma = \sqrt{\frac{2}{3}}$. In this way, we *pick the values of the parameter that make the data we have observed the most likely*. Written in mathematical notation, this is

$$\theta_{\text{MLE}} = \arg\max_{\theta} L(y_1, \ldots, y_n \mid \theta).$$

The negative log-likelihood

Recall that, for independent observations, the likelihood becomes a product:

$$L(y_1, \ldots, y_n \,|\, \theta) = \prod_{i=1}^{n} P(y_i \,|\, \theta).$$

Furthermore, since the $\log$ function increases with its argument, maximising the likelihood is equivalent to maximising the log-likelihood $\log L(y_1, \ldots, y_n \,|\, \theta)$. This changes the product into a sum:

$$\begin{aligned}
\theta_{\text{MLE}} &= \arg\max_{\theta} \; L(y_1, \ldots, y_n \,|\, \theta) \\
&= \arg\max_{\theta} \; \log L(y_1, \ldots, y_n \,|\, \theta) \\
&= \arg\max_{\theta} \; \log \prod_{i=1}^{n} L(y_i \,|\, \theta) \\
&= \arg\max_{\theta} \; \sum_{i=1}^{n} \log L(y_i \,|\, \theta).
\end{aligned}$$

Furthermore, convention in optimisation is that we always *minimise* a function instead of maximising it. Hence, maximising the likelihood is equivalent to *minimising* the *negative log-likelihood*:

$$\theta_{\text{MLE}} = \arg\min_{\theta} \; \text{NLL}(y_1, \ldots, y_n \,|\, \theta)$$

where the *negative log-likelihood* NLL is defined as

$$\text{NLL}(y_1, \ldots, y_n \,|\, \theta) = \; - \sum_{i=1}^{n} \log L(y_i \,|\, \theta).$$

Training neural networks

How is all this used to train neural networks? We do this, given some training data, by picking the weights of the neural network that maximise the likelihood (or, equivalently, minimise the negative loglikelihood) of having observed that data. More specifically, the neural network is a function that maps a data point x_i to the parameter θ of some distribution. This parameter indicates the probability of seeing each possible label. We then use our true labels and the likelihood to find the best weights of the neural network.

Let's be a bit more precise about this. Suppose we have a neural network NN with weights $\mathbf{w}$. Furthemore, suppose x_i is some data point, e.g. an image to be classified, or an x value for which we want to predict the y value. The neural network prediction (the feedforward value) $\hat{y}_i$ is

$$\hat{y}_i = \text{NN}(x_i \,|\, \mathbf{w}).$$

We can use this to train the neural network (determine its weights $\mathbf{w}$) as follows. We assume that the neural network prediction $\hat{y}_i$ forms part of a distribution that the true label is drawn from. Suppose we have some training data consisting of inputs and the associated labels. Let the data be x_i and the labels y_i for $i = 1, \ldots, n$, where n is the number of training samples. The training data is hence

$$\text{training data} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$$

For each point x_i, we have the neural network prediction $\hat{y}_i = \text{NN}(x_i \,|\, \mathbf{w})$, which we assume specifies a distribution. We also have the true label y_i. The weights of the trained neural network are then those that minimise the negative log-likelihood:

$$\begin{aligned}
\mathbf{w}^* &= \arg\min_{\mathbf{w}} \; \Big(- \sum_{i=1}^{n} \log L(y_i \,|\, \hat{y}_i) \Big) \\
&= \arg\min_{\mathbf{w}} \; \Big(- \sum_{i=1}^{n} \log L(y_i \,|\, \text{NN}(x_i \,|\, \mathbf{w})) \Big)
\end{aligned}$$

In practice, determining the true optimum $\mathbf{w}^*$ is not always possible. Instead, an approximate value is sought using stochastic gradient descent, usually via a *backpropagation* of derivatives and some optimization algorithm such as `RMSprop` or `Adam`.

Let's see some examples to make this idea more concrete.

Bernoulli distribution: binary classifiers

Suppose we want a neural network NN that classifies images into either cats or dogs. Here, x_i is an image of either a cat or a dog, and $\hat{y}_i$ is the probability that this image is either a cat (value 0) or a dog (value 1):

$$\hat{y}_i = \text{NN}(x_i \,|\, \mathbf{w}) = \text{Prob}(\text{image is dog}).$$

Note that this is just a Bernoulli distribution with values 0 and 1 corresponding to cat and dog respectively, of which we discussed the likelihood function above. Given training data $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, with $y_i \in \{0, 1\}$, we have the negative log-likelihood

$$\begin{aligned}
\text{NLL}((x_1, y_1), \ldots, (x_n, y_n) \mid \mathbf{w}) &= -\sum_{i=1}^{n} \log L(y_i \mid \hat{y}_i) \\
&= -\sum_{i=1}^{n} \log\left((1-\hat{y}_i)^{1-y_i} \hat{y}_i^{y_i}\right) \\
&= -\sum_{i=1}^{n} \left((1-y_i)\log(1-\hat{y}_i) + y_i \log \hat{y}_i\right) \\
&= -\sum_{i=1}^{n} \left((1-y_i)\log(1-\text{NN}(x_i \mid \mathbf{w})) + y_i \log \text{NN}(x_i \mid \mathbf{w})\right).
\end{aligned}$$

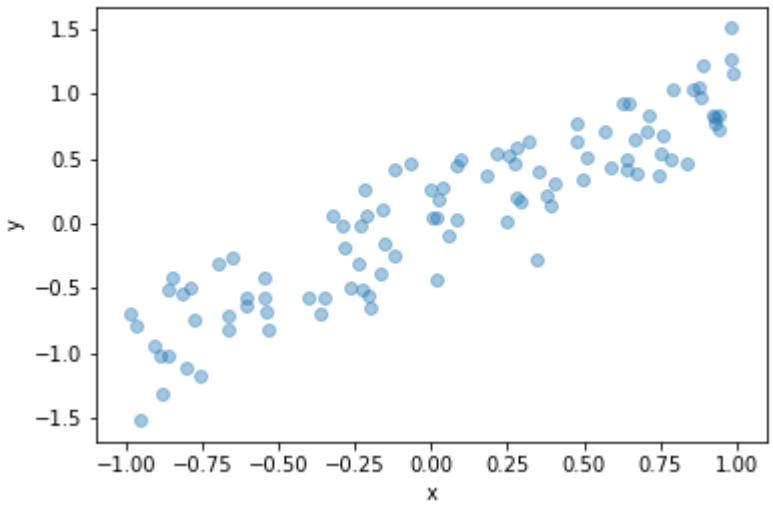
This is exactly the sparse categorical cross-entropy loss function used when training a classification neural network. Hence, the reason why we typically use categorical cross-entropy loss functions when training classification data is exactly because this is the negative log-likelihood under a Bernoulli (or, when there are more than 2 classes, a categorical) distribution.

Normal distribution: least squares regression

The idea works the same way in a regression task. Here, we have an x-value x_i and want to predict the associated y-value y_i. We can use a neural network to do this, giving a prediction $\hat{y}_i$:

$$\hat{y}_i = \text{NN}(x_i \mid \mathbf{w}).$$

For example, suppose we were doing linear regression with the following data.



It's not possible to put a straight line through every data point. Furthermore, even points with the same x value might not have the same y value. We can interpret this as y being linearly related to x with some noise. More precisely, we may assume that

$$y_i = f(x_i) + \epsilon_i \qquad \epsilon_i \sim N(0, \sigma^2)$$

where f is some function we want to determine (the regression) and ϵ_i is some Gaussian noise with mean 0 and constant variance σ^2. In deep learning, we might approximate $f(x_i)$ by a neural network $\text{NN}(x_i \mid \mathbf{w})$ with weights $\mathbf{w}$ and output $\hat{y}_i$.

$$\hat{y}_i = \text{NN}(x_i \mid \mathbf{w}) = f(x_i)$$

Under this assumption, we have

$$\epsilon_i = y_i - \hat{y}_i \sim N(0, \sigma^2)$$

and hence, given training data $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, we have the negative log-likelihood (assuming the noise terms are independent):

$$\begin{aligned}
\text{NLL}((x_1, y_1), \ldots, (x_n, y_n) \mid \mathbf{w}) &= -\sum_{i=1}^{n} \log L(y_i \mid \hat{y}_i) \\
&= -\sum_{i=1}^{n} \log\left(\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(\hat{y}_i - y_i)^2\right)\right) \\
&= \frac{n}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \\
&= \frac{n}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}\sum_{i=1}^{n}(\text{NN}(x_i \mid \mathbf{w}) - y_i)^2.
\end{aligned}$$

Note that only the last term includes the weights. Hence, minimising the negative log-likelihood is equivalent to minimising

$$\sum_{i=1}^{n}(\text{NN}(x_i \mid \mathbf{w}) - y_i)^2$$

which is exactly the sum of squared errors. Hence, least squares regression (or training a neural network using the mean squared error) is equivalent to training a neural network to match the expected value of an output by minimising the negative log-likelihood assuming a Gaussian error term with constant variance.

Conclusion

This was a very short introduction to maximum likelihood estimation, which is essential for deep learning, especially of the probabilistic variety that we'll be doing in this course. The method of maximum likelihood estimation is key to training neural networks, and typically informs the choice of loss function. In fact, you have probably trained neural networks using maximum likelihood estimation without even knowing it!

Further reading and resources

I find that the Wikipedia pages for many statistical concepts offer excellent intuition. If you'd like to read up on these ideas in more detail, I'd recommend these:

- The Wikipedia page for Probability Distribution: https://en.wikipedia.org/wiki/Probability_distribution (https://en.wikipedia.org/wiki/Probability_distribution)
- The Wikipedia page for Maximum Likelihood Estimation: https://en.wikipedia.org/wiki/Maximum_likelihood_estimation (https://en.wikipedia.org/wiki/Maximum_likelihood_estimation)

Bayes by backprop

This reading is a review of the *Bayes by backprop* method for quantifying epistemic uncertainty in a neural network model.

Introduction

This reading discusses one way to introduce weight uncertainty into neural networks. It will describe the *Bayes by Backprop* method introduced in this paper:

- Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, Daan Wierstra. Weight uncertainty in neural networks (https://arxiv.org/pdf/1505.05424.pdf). In *Proceedings of the 32nd International Conference on Machine Learning*.

The main idea is as follows. In a traditional neural network, as shown in the left figure below, each weight has a single value. The true value of this weight is not certain. A lot of this uncertainty comes from imperfect training data, which does not exactly describe the distribution the data were drawn from.

As an analogy, consider the problem of determining the population average height by measuring 100 randomly selected people. An estimate of the population mean is then the mean calculated across these 100. However, this is just an estimate; we may, by chance, have selected 100 people that are slightly taller than expected, or shorter. Recall that this is called *epistemic* uncertainty, and we expect it to decrease as the amount of training data increases. For example, the uncertainty on an estimate obtained using 100 people is larger than one using 10,000 people.

In this reading, we want to include such uncertainty in deep learning models. This is done by changing each weight from a single deterministic value to a *probability distribution*. We then learn the parameters of this distribution. Consider a neural network weight w_i. In a standard (deterministic) neural network, this has a single value $\hat{w}_i$, learnt via backpropagation. In a neural network with weight uncertainty, each weight is represented by a probability distribution, and the *parameters* of this distribution are learned via backpropagation. Suppose, for example, that each weight has a normal distribution. This has two parameters: a mean μ_i and a standard deviation σ_i.

- Classic deterministic NN: $w_i = \hat{w}_i$
- NN with weight uncertainty represented by normal distribution: $w_i \sim N(\hat{\mu}_i, \hat{\sigma}_i)$.

An image of the situation, provided in the paper, is as follows.

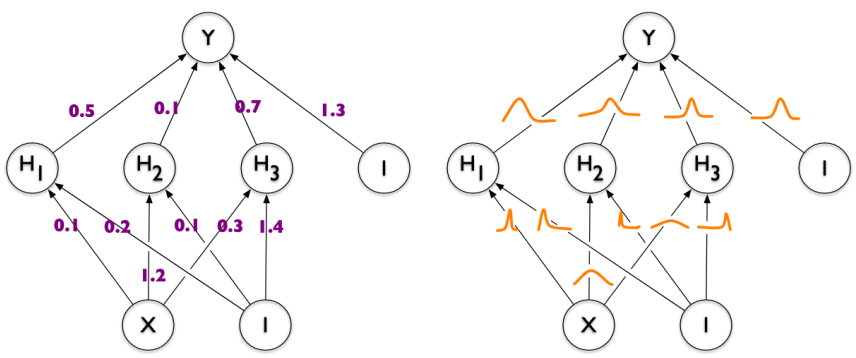


Figure 1. Left: each weight has a fixed value, as provided by classical backpropagation. Right: each weight is assigned a distribution, as provided by Bayes by Backprop.

Since the weights are uncertain, the feedforward value of some input x_i is not constant. A single feedforward value is determined in two steps:

1. Sample each network weight from their respective distributions -- this gives a single set of network weights.
2. Use these weights to determine a feedforward value $\hat{y}_i$.

Hence, the key question is how to determine the parameters of the distribution for each network weight. The paper introduces exactly such a scheme, called *Bayes by Backprop*. The details are discussed in the remainder of this reading.

Bayesian learning

Note: In this reading, we use the notation P to refer to a probability density. For simplicity, we'll only consider continuous distributions (which have a density). In the case of discrete distributions, P would represent a probability mass and integrals should be changed to sums. However, the formulae are the same.

Bayesian methods represent one common framework in which to conduct statistical inference. We only provide a very short introduction here, but for a more detailed account of Bayesian inference you could read the Wikipedia article (https://en.wikipedia.org/wiki/Bayesian_inference) and references therein.

What you need to know now is that Bayesian methods can be used to calculate the distribution of a model parameter given some data. In the context of weight uncertainty in neural networks, this is convenient, since we are looking for the distribution of weights (model parameters) given some (training) data. The key step relies on Bayes' theorem. This theorem states, in mathematical notation, that

$$P(w \mid D) = \frac{P(D \mid w)P(w)}{\int P(D \mid w')P(w')\mathrm{d}w'}$$

where the terms mean the following:

- D is some data, e.g. x and y value pairs: $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$. This is sometimes called the *evidence*.
- w is the value of a model weight.
- $P(w)$ is called the *prior*. This is our "prior" belief on the probability density of a model weight, i.e. the distribution that we postulate before seeing any data.
- $P(D \mid w)$ is the *likelihood* of having observed data D given weight w. It is precisely the same likelihood we discussed in the previous reading and is used to calculate the negative log-likelihood.
- $P(w \mid D)$ is the *posterior* density of the distribution of the model weight at value w, given our training data. It is called *posterior* since it represents the distribution of our model weight *after* taking the training data into account.

Note that the term $\int P(D \mid w')P(w')\mathrm{d}w' = P(D)$ does not depend on w (as the w' is an integration variable). It is only a normalisation term. For this reason, we will from this point on write Bayes' theorem as

$$P(w \mid D) = \frac{P(D \mid w)P(w)}{P(D)}.$$

Bayes' theorem gives us a way of combining data with some "prior belief" on model parameters to obtain a distribution for these model parameters that considers the data, called the *posterior distribution*.

Bayesian neural network with weight uncertainty -- in principle

The above formula gives a way to determine the distribution of each weight in the neural network:

1. Pick a prior density $P(w)$.
2. Using training data D, determine the likelihood $P(D \mid w)$.
3. Determine the posterior density $P(w \mid D)$ using Bayes' theorem. This is the distribution of the NN weight.

While this works in principle, in many practical settings it is difficult to implement. The main reason is that the normalisation constant $\int P(D \mid w')P(w')\mathrm{d}w' = P(D)$ may be very difficult to calculate, as it involves solving or approximating a complicated integral. For this reason, approximate methods, such as *Variational Bayes* described below, are often employed.

Variational Bayes

Variational Bayes methods approximate the posterior distribution with a second function, called a *variational posterior*. This function has a known functional form, and hence avoids the need to determine the posterior $P(w \mid D)$ exactly. Of course, approximating a function with another one has some risks, since the approximation may be very bad, leading to a posterior that is highly inaccurate. In order to mediate this, the variational posterior usually has a number of parameters, denoted by θ, that are tuned so that the function approximates the posterior as well as possible. Let's see how this works below.

Instead of $P(w \mid D)$, we assume the network weight has density $q(w \mid \theta)$, parameterized by θ. $q(w \mid \theta)$ is known as the *variational posterior*. We want $q(w \mid \theta)$ to approximate $P(w \mid D)$, so we want the "difference" between $q(w \mid \theta)$ and $P(w \mid D)$ to be as small as possible. This "difference" between the two distributions is usually measured by the Kullback-Leibler divergence (https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence) D_{KL} (note that this is **unrelated** to the D we use to denote the data). The Kullback-Leibler divergence between two distributions with densities $f(x)$ and $g(x)$ respectively is defined as

$$D_{KL}(f(x) \mid\mid g(x)) = \int f(x)\log\left(\frac{f(x)}{g(x)}\right)\mathrm{d}x.$$

Note that this function has value 0 (indicating no difference) when $f(x) \equiv g(x)$, which is the result we expect. We use the convention that $\frac{0}{0} = 1$ here.

Viewing the data D as a constant, the Kullback-Leibler divergence between $q(w \mid \theta)$ and $P(w \mid D)$ is hence

$$
\begin{aligned}
D_{KL}(q(w\,|\,\theta)\,||\,P(w\,|\,D)) &= \int q(w\,|\,\theta)\log\left(\frac{q(w\,|\,\theta)}{P(w\,|\,D)}\right)\mathrm{d}w \\
&= \int q(w\,|\,\theta)\log\left(\frac{q(w\,|\,\theta)P(D)}{P(D\,|\,w)P(w)}\right)\mathrm{d}w \\
&= \int q(w\,|\,\theta)\log P(D)\mathrm{d}w + \int q(w\,|\,\theta)\log\left(\frac{q(w\,|\,\theta)}{P(w)}\right)\mathrm{d}w - \int q(w\,|\,\theta)\log P(D\,|\,w)\mathrm{d}w \\
&= \log P(D) + D_{KL}(q(w\,|\,\theta)\,||\,P(w)) - \mathrm{E}_{q(w\,|\,\theta)}(\log P(D\,|\,w))
\end{aligned}
$$

where, in the last line, we have used $\int q(w\,|\,\theta)\log P(D)\mathrm{d}w = \log P(D)\int q(w\,|\,\theta)\mathrm{d}w = \log P(D)$ since $q(w\,|\,\theta)$ is a probability distribution and hence integrates to 1. If we consider the data D to be constant, the first term is a constant also, and we may ignore it when minimising the above. Hence, we are left with the function

$$
L(\theta\,|\,D) = D_{KL}(q(w\,|\,\theta)\,||\,P(w)) - \mathrm{E}_{q(w\,|\,\theta)}(\log P(D\,|\,w))
$$

Note that this function depends only on θ and D, since w is an integration variable. This function has a nice interpretation as the sum of:

- The Kullback-Leibler divergence between the variational posterior $q(w\,|\,\theta)$ and the prior $P(w)$. This is called the *complexity cost*, and it depends on θ and the prior but not the data D.
- The expectation of the negative loglikelihood $\log P(D\,|\,w)$ under the variational posterior $q(w\,|\,\theta)$. This is called the *likelihood cost* and it depends on θ and the data but not the prior.

$L(\theta\,|\,D)$ is the loss function that we minimise to determine the parameter θ. Note also from the above derivation, that we have

$$
\begin{aligned}
\log P(D) &= \mathrm{E}_{q(w\,|\,\theta)}(\log P(D\,|\,w)) - D_{KL}(q(w\,|\,\theta)\,||\,P(w)) + D_{KL}(q(w\,|\,\theta)\,||\,P(w\,|\,D)) \\
&\geq \mathrm{E}_{q(w\,|\,\theta)}(\log P(D\,|\,w)) - D_{KL}(q(w\,|\,\theta)\,||\,P(w)) =: ELBO
\end{aligned}
$$

which follows because $D_{KL}(q(w\,|\,\theta)\,||\,P(w\,|\,D))$ is nonnegative. The final expression on the right hand side is therefore a lower bound on the log-evidence, and is called the *evidence lower bound*, often shortened to *ELBO*. The ELBO is the negative of our loss function, so minimising the loss function is equivalent to maximising the ELBO.

Maximising the ELBO requires a tradeoff between the KL term and expected log-likelihood term. On the one hand, the divergence between $q(w\,|\,\theta)$ and $P(w)$ should be kept small, meaning the variational posterior shouldn't be too different to the prior. On the other, the variational posterior parameters should maximise the expectation of the log-likelihood $\log P(D\,|\,w)$, meaning the model assigns a high likelihood to the data.

A backpropagation scheme

The idea

We can use the above ideas to create a neural network with weight uncertainty, which we will call a *Bayesian neural network*. From a high level, this works as follows. Suppose we want to determine the distribution of a particular neural network weight w.

1. Assign the weight a prior distribution with density $P(w)$, which represents our beliefs on the possible values of this network before any training data. This may be something simple, like a unit Gaussian. Furthermore, this prior distribution will usually not have any trainable parameters.
2. Assign the weight a variational posterior with density $q(w\,|\,\theta)$ with some trainable parameter θ.
3. $q(w\,|\,\theta)$ is the approximation for the weight's posterior distribution. Tune θ to make this approximation as accurate as possible as measured by the ELBO.

The remaining question is then how to determine θ. Recall that neural networks are typically trained via a backpropagation algorithm, in which the weights are updated by perturbing them in a direction that reduces the loss function. We aim to do the same here, by updating θ in a direction that reduces $L(\theta\,|\,D)$.

Hence, the function we want to minimise is

$$
\begin{aligned}
L(\theta\,|\,D) &= D_{KL}(q(w\,|\,\theta)\,||\,P(w)) - \mathrm{E}_{q(w\,|\,\theta)}(\log P(D\,|\,w)) \\
&= \int q(w\,|\,\theta)(\log q(w\,|\,\theta) - \log P(D\,|\,w) - \log P(w))\mathrm{d}w.
\end{aligned}
$$

In principle, we could take derivatives of $L(\theta\,|\,D)$ with respect to θ and use this to update its value. However, this involves doing an integral over w, and this is a calculation that may be impossible or very computationally expensive. Instead, we want to write this function as an expectation and use a Monte Carlo approximation to calculate derivatives. At present, we can write this function as

$$
L(\theta\,|\,D) = \mathrm{E}_{q(w\,|\,\theta)}(\log q(w\,|\,\theta) - \log P(D\,|\,w) - \log P(w)).
$$

However, taking derivatives with respect to θ is difficult because the underlying distribution the expectation is taken with respect to depends on θ. One way we can handle this is with the *reparameterization trick*.

The reparameterization trick

The reparameterization trick is a way to move the dependence on θ around so that an expectation may be taken independently of it. It's easiest to see how this works with an example. Suppose $q(w\,|\,\theta)$ is a Gaussian, so that $\theta = (\mu, \sigma)$. Then, for some arbitrary $f(w; \mu, \sigma)$, we have

$$
\begin{aligned}
\mathrm{E}_{q(w|\mu,\sigma)}(f(w;\mu,\sigma)) &= \int q(w|\mu,\sigma) f(w;\mu,\sigma)\mathrm{d}w \\
&= \int \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(w-\mu)^2\right) f(w;\mu,\sigma)\mathrm{d}w \\
&= \int \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}\epsilon^2\right) f(\mu+\sigma\epsilon;\mu,\sigma)\mathrm{d}\epsilon \\
&= \mathrm{E}_{\epsilon\sim N(0,1)}(f(\mu+\sigma\epsilon;\mu,\sigma))
\end{aligned}
$$

where we used the change of variable $w = \mu + \sigma\epsilon$. Note that the dependence on $\theta = (\mu, \sigma)$ is now only in the integrand and we can take derivatives with respect to μ and σ:

$$
\frac{\partial}{\partial\mu}\mathrm{E}_{q(w|\mu,\sigma)}(f(w;\mu,\sigma)) = \frac{\partial}{\partial\mu}\mathrm{E}_{\epsilon\sim N(0,1)}(f(w;\mu,\sigma)) = \mathrm{E}_{\epsilon\sim N(0,1)}\frac{\partial}{\partial\mu}f(\mu+\sigma\epsilon;\mu,\sigma)
$$

$$
\frac{\partial}{\partial\sigma}\mathrm{E}_{q(w|\mu,\sigma)}(f(w;\mu,\sigma)) = \frac{\partial}{\partial\sigma}\mathrm{E}_{\epsilon\sim N(0,1)}(f(w;\mu,\sigma)) = \mathrm{E}_{\epsilon\sim N(0,1)}\frac{\partial}{\partial\sigma}f(\mu+\sigma\epsilon;\mu,\sigma)
$$

Finally, note that we can approximate the expectation by its Monte Carlo estimate:

$$
\mathrm{E}_{\epsilon\sim N(0,1)}\frac{\partial}{\partial\theta}f(\mu+\sigma\epsilon;\mu,\sigma) \approx \sum_i \frac{\partial}{\partial\theta}f\left(\mu+\sigma\epsilon_i;\mu,\sigma\right), \qquad \epsilon_i \sim N(0,1).
$$

The above reparameterization trick works in cases where we can write the $w = g(\epsilon, \theta)$, where the distribution of the random variable ϵ is independent of θ.

Implementation

Putting this all together, for our loss function $L(\theta|D) \equiv L(\mu, \sigma|D)$, we have

$$
f(w;\mu,\sigma) = \log q(w|\mu,\sigma) - \log P(D|w) - \log P(w)
$$

$$
\frac{\partial}{\partial\mu}L(\mu,\sigma|D) \approx \sum_i \left(\frac{\partial f(w_i;\mu,\sigma)}{\partial w_i} + \frac{\partial f(w_i;\mu,\sigma)}{\partial\mu}\right)
$$

$$
\frac{\partial}{\partial\sigma}L(\mu,\sigma|D) \approx \sum_i \left(\frac{\partial f(w_i;\mu,\sigma)}{\partial w_i}\sigma + \frac{\partial f(w_i;\mu,\sigma)}{\partial\sigma}\right)
$$

$$
f(w;\mu,\sigma) = \log q(w|\mu,\sigma) - \log P(D|w) - \log P(w)
$$

where $w_i = \mu + \sigma\epsilon_i$, $\epsilon_i \sim N(0, 1)$. In practice, we often only take a single sample ϵ_1 for each training point. This leads to the following backpropagation scheme:

1. Sample $\epsilon_i \sim N(0, 1)$.
2. Let $w_i = \mu + \sigma\epsilon_i$
3. Calculate

$$
\nabla_\mu f = \frac{\partial f(w_i;\mu,\sigma)}{\partial w_i} + \frac{\partial f(w_i;\mu,\sigma)}{\partial\mu} \qquad \nabla_\sigma f = \frac{\partial f(w_i;\mu,\sigma)}{\partial w_i}\sigma + \frac{\partial f(w_i;\mu,\sigma)}{\partial\sigma}
$$

4. Update the parameters with some gradient-based optimiser using the above gradients.

This is how we learn the parameters of the distribution for each neural network weight.

Minibatches

Note that the loss function (or negative of the ELBO) is

$$
\begin{aligned}
L(\theta|D) &= D_{KL}(q(w|\theta)||P(w)) - \mathrm{E}_{q(w|\theta)}(\log P(D|w)) \\
&= D_{KL}(q(w|\theta)||P(w)) - \sum_{j=1}^{N} \log P(y_j, x_j|w_j)
\end{aligned}
$$

where j runs over all the data points in the training data (N in total) and $w_j = \mu + \sigma\epsilon_j$ is sampled using $\epsilon_j \sim N(0, 1)$ (we assume a single sample from the approximate posterior per data point for simplicity).

If training occurs in minibatches of size B, typically much smaller than N, we instead have a loss function

$$
D_{KL}(q(w|\theta)||P(w)) - \sum_{j=1}^{B} \log P(y_j, x_j|w_j).
$$

Note that the scaling factors between the first and second terms have changed, since before the sum ran from 1 to N, but it now runs from 1 to B. To correct for this, we should add a correction factor $\frac{N}{B}$ to the second term to ensure that its expectation is the same as before. This leads to the loss function, after dividing by N to take the average per training value, of

$$
\frac{1}{N}D_{KL}(q(w|\theta)||P(w)) - \frac{1}{B}\sum_{j=1}^{B} \log P(y_j, x_j|w_j).
$$

By default, when Tensorflow calculates the loss function, it calculates the average across the minibatch. Hence, it already uses the factor $\frac{1}{B}$ present on the second term. However, it does not, by default, divide the first term by N. In an implementation, we will have to specify this. You'll see in the next lectures and coding tutorials how to do this.

Conclusion

This reading introduces the *Bayes by Backpropagation* method, which can be used to embed weight uncertainty into neural networks. Good job getting through it, as the topic is rather advanced. This approach allows the modelling of *epistemic* uncertainty on the model weights. We expect that, as the number of training points increases, the uncertainty on the model weights decreases. This can be shown to be the case in many settings. In the next few lectures and coding tutorials, you'll learn how to apply these methods to your own models, which will make the idea much clearer.

Further reading and resources

- Bayes by backprop paper: https://arxiv.org/pdf/1505.05424.pdf (https://arxiv.org/pdf/1505.05424.pdf)
- Wikipedia article on Bayesian inference: https://en.wikipedia.org/wiki/Bayesian_inference (https://en.wikipedia.org/wiki/Bayesian_inference)

Probabilistic Layers And Bayesian Neural Networks

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

print('TF version:', tf.__version__)
print('TFP version:', tfp.__version__)
```

```
TF version: 2.3.0
TFP version: 0.11.0
```

1. The DistributionLambda layer

2. Probabilistic layers

3. The DenseVariational layer

4. Reparameterization layers

The `DistributionLambda` layer

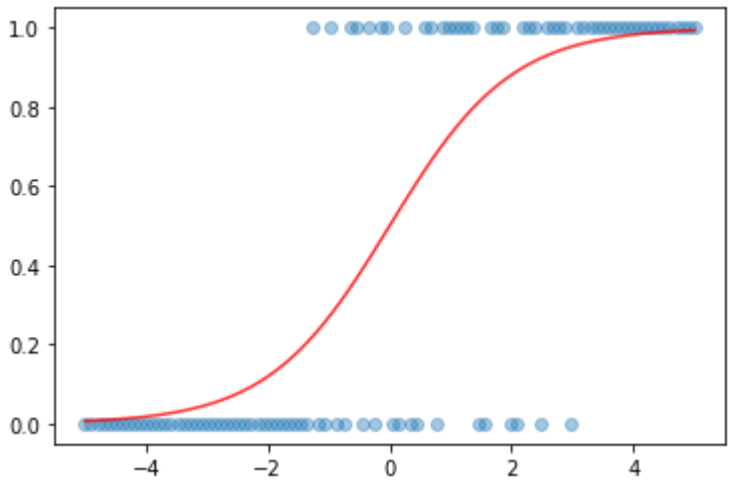
In [2]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import RMSprop
import numpy as np
import matplotlib.pyplot as plt
```

Create a probabilistic model using the `DistributionLambda` layer

Create a model whose first layer represents:

$$y = \text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}.$$

In [3]:
```
# Create a sigmoid model, first deterministic, then probabilistic
model = Sequential ([
    Dense (
        input_shape = (1,), units = 1, activation = 'sigmoid',
        kernel_initializer = tf.constant_initializer(1),
        bias_initializer = tf.constant_initializer(0)
    ),
    tfpl.DistributionLambda (
        lambda t: tfd.Bernoulli(probs = t),
        convert_to_tensor_fn = tfd.Distribution.sample
    )
])

# Plot the function
x_plot = np.linspace(-5, 5, 100)
plt.scatter(x_plot, model.predict(x_plot), alpha = 0.4)
plt.plot(x_plot, 1 / (1 + np.exp(-x_plot)), color = 'r', alpha = 0.8)
plt.show()
```



In [4]:
```
# Create a constant input for this model
x = np.array([[0]])
x
```

Out[4]:
```
array([[0]])
```

In [5]:
```
# Explore the feedforward object...
y_model = model(x)
y_model
```

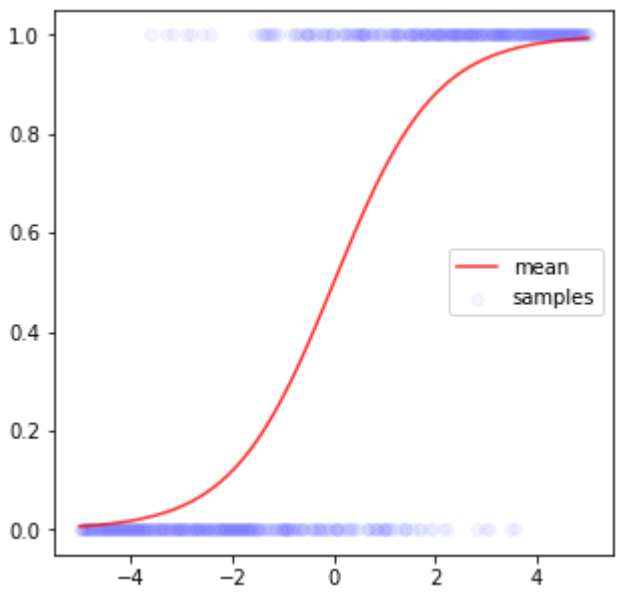
Out[5]:
```
<tfp.distributions.Bernoulli 'sequential_distribution_lambda_Bernoulli' batch_shape=[1, 1] event_shape=[] dtype=int32>
```

In [6]:
```
# ... and its behaviour under repeated calls
for _ in range(5):
    print(model.predict(x))
```

```
[[0]]
[[0]]
[[0]]
[[1]]
[[0]]
```

Use the forward model to create probabilistic training data

In [7]:
```
# Use the model to create 500 training points
x_train = np.linspace(-5, 5, 500)[:, np.newaxis]
y_train = model.predict(x_train)

# Plot the data and the mean of the distribution
fig, ax = plt.subplots(figsize = (5, 5))
ax.scatter(x_train, y_train, alpha = 0.04, color = 'blue', label = 'samples')
ax.plot(x_train, model(x_train).mean().numpy().flatten(),
        color = 'red', alpha = 0.8, label = 'mean')
ax.legend()
plt.show()
```



Create a new probabilistic model with the wrong weights

In [8]:
```
# Create a new version of the model, with the wrong weights
model_untrained = Sequential ([
    Dense (
        input_shape = (1,), units = 1, activation = 'sigmoid',
        kernel_initializer = tf.constant_initializer(2),
        bias_initializer = tf.constant_initializer(2)
    ),
    tfpl.DistributionLambda (
        lambda t: tfd.Bernoulli(probs = t),
        convert_to_tensor_fn = tfd.Distribution.sample
    )
])
```

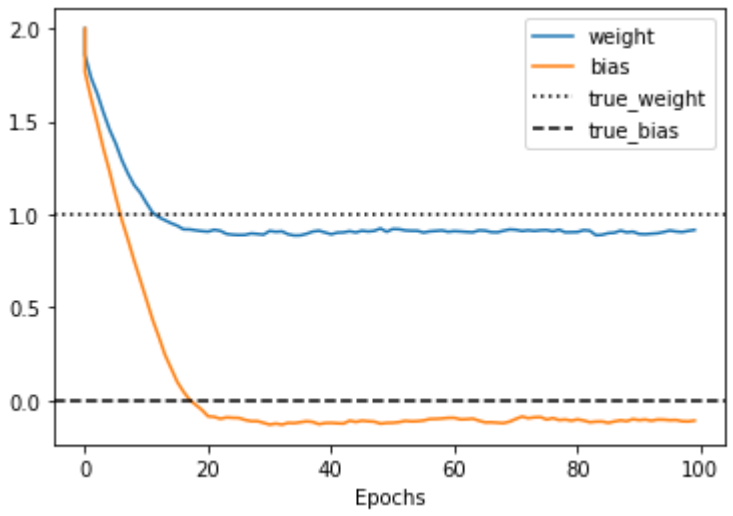
Train the new model with the negative loglikelihood

In [9]:
```
# Define negative loglikelihood, which we will use for training
def nll(y_true, y_pred):
    return -y_pred.log_prob(y_true)
```

In [10]:
```
# Compile untrained model
model_untrained.compile(loss = nll, optimizer = RMSprop(learning_rate = 0.01))
```

In [11]:
```
# Train model, record weights after each epoch
epochs = [0]
training_weights = [model_untrained.weights[0].numpy()[0, 0]]
training_bias = [model_untrained.weights[1].numpy()[0]]
for epoch in range(100):
    model_untrained.fit(x = x_train, y = y_train, epochs = 1, verbose = False)
    epochs.append(epoch)
    training_weights.append(model_untrained.weights[0].numpy()[0, 0])
    training_bias.append(model_untrained.weights[1].numpy()[0])
```

In [12]:
```
# Plot the model weights as they train, converging to the correct values
plt.plot(epochs, training_weights, label = 'weight')
plt.plot(epochs, training_bias, label = 'bias')
plt.axhline(y = 1, label = 'true_weight', color = 'k', linestyle = ':')
plt.axhline(y = 0, label = 'true_bias', color = 'k', linestyle = '--')
plt.xlabel('Epochs')
plt.legend()
plt.show()
```



Probabilistic layers

In [13]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.losses import MeanSquaredError
from tensorflow.keras.optimizers import RMSprop
import numpy as np
import matplotlib.pyplot as plt
```

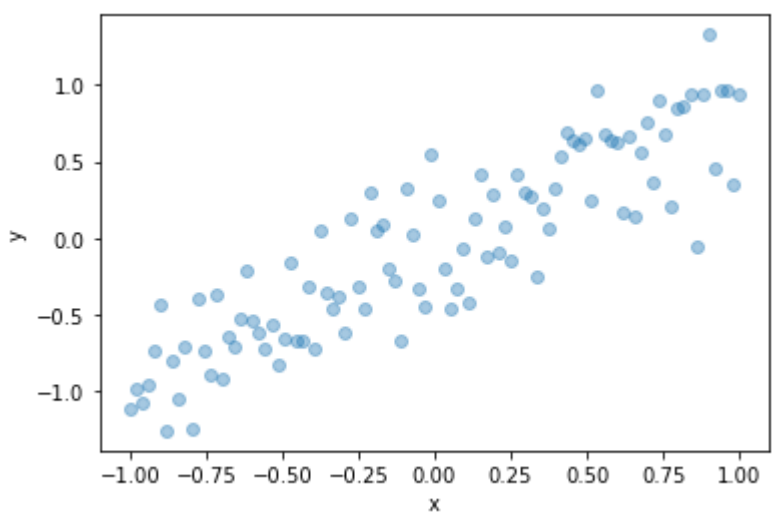
Create data

The data you'll be working with is artifically created from the following equation:

$$y_i = x_i + \frac{3}{10}\epsilon_i$$

where $\epsilon_i \sim N(0, 1)$ are independent and identically distributed.

In [14]:
```
# Create and plot 100 points of training data
x_train = np.linspace(-1, 1, 100)[:, np.newaxis]
y_train = x_train + 0.3 * np.random.randn(100)[:, np.newaxis]

plt.scatter(x_train, y_train, alpha = 0.4)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

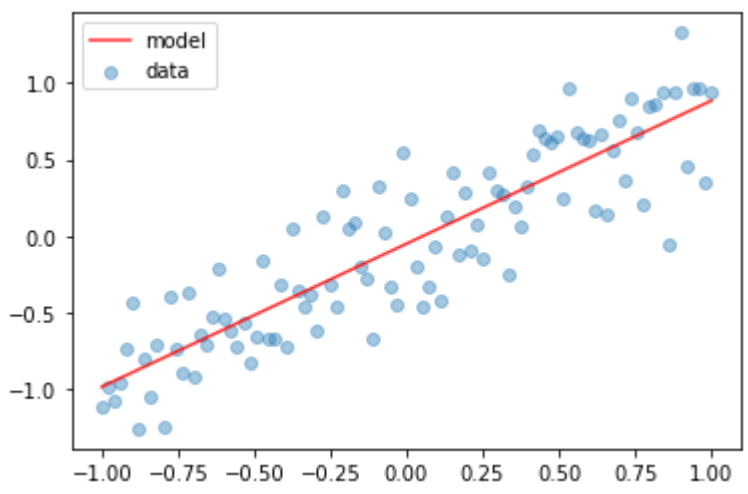


Deterministic linear regression with MSE loss

In [15]:
```
# Create and train deterministic linear model using mean squared error loss

# Create linear regression via Sequential model
model = Sequential([Dense(units = 1, input_shape = (1,))])
model.compile(loss = MeanSquaredError(), optimizer = RMSprop(learning_rate = 0.005))
```

```
model.summary()
model.fit(x_train, y_train, epochs = 200, verbose = False)

# Plot the data and model
plt.scatter(x_train, y_train, alpha = 0.4, label = 'data')
plt.plot(x_train, model.predict(x_train), color = 'red', alpha = 0.8, label = 'model')
plt.legend()
plt.show()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 1)                 2
=================================================================
Total params: 2
Trainable params: 2
Non-trainable params: 0
_________________________________________________________________
```



In [16]:
```
# Examine the model predictions
x = np.array([[0]])
y_model = model(x)
y_model
```

Out[16]:
```
<tf.Tensor: shape=(1, 1), dtype=float32, numpy=array([[-0.05189721]], dtype=float32)>
```

Probabilistic linear regression with both user-defined and learned variance

In [17]:
```
# Create probabilistic regression with normal distribution as final layer
event_shape = 1
model = Sequential ([
    Dense(units = tfpl.IndependentNormal.params_size(event_shape), input_shape = (1,)),
    tfpl.IndependentNormal(event_shape)
])
model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_3 (Dense)              (None, 2)                 4
_________________________________________________________________
independent_normal (Independ ((None, 1), (None, 1))    0
=================================================================
Total params: 4
Trainable params: 4
Non-trainable params: 0
_________________________________________________________________
```

In [18]:
```
# Train model using the negative loglikelihood
def nll(y_true, y_pred):
    return -y_pred.log_prob(y_true)

model.compile(loss = nll, optimizer = RMSprop(learning_rate = 0.005))
model.fit(x_train, y_train, epochs = 200, verbose = False)
```

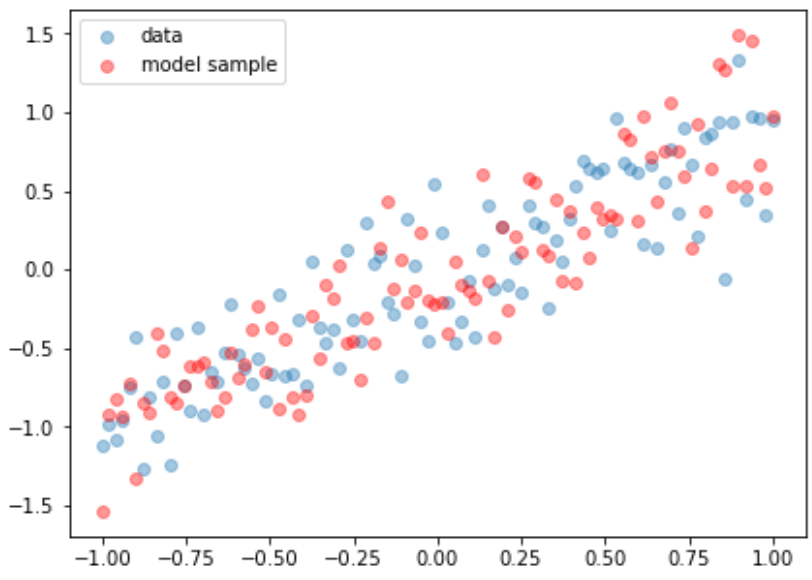
Out[18]:
```
<tensorflow.python.keras.callbacks.History at 0x7fb3f45d8e50>
```

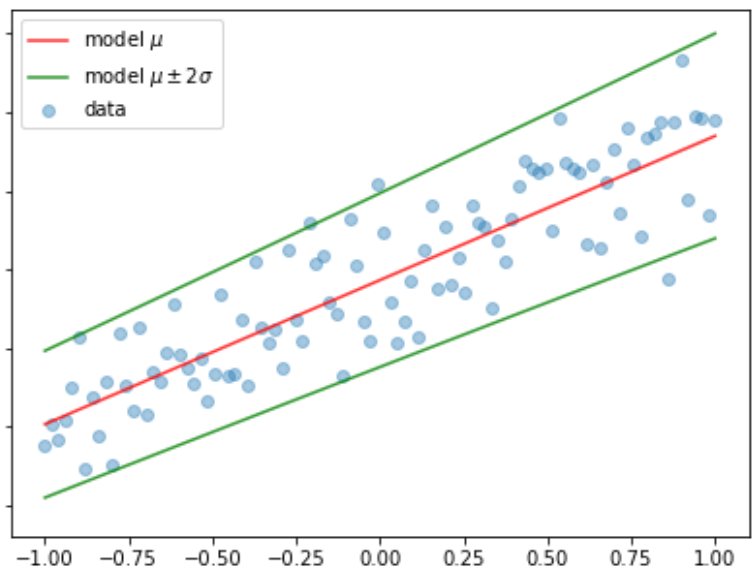
In [19]:
```
# Examine the distribution created as a feedforward value
y_model = model(x)
y_model
```

Out[19]:
```
<tfp.distributions.Independent 'sequential_3_independent_normal_IndependentNormal_Independentsequential_3_independent_
normal_IndependentNormal_Normal' batch_shape=[1] event_shape=[1] dtype=float32>
```

In [20]:
```
# Plot the data and a sample from the model
y_model = model(x_train)
y_sample = y_model.sample()
y_hat = y_model.mean()
y_sd = y_model.stddev()
y_hat_m2sd = y_hat - 2 * y_sd
y_hat_p2sd = y_hat + 2 * y_sd

fig, (ax1, ax2) = plt.subplots(1, 2, figsize = (15, 5), sharey = True)
ax1.scatter(x_train, y_train, alpha = 0.4, label = 'data')
ax1.scatter(x_train, y_sample, alpha = 0.4, color = 'red', label = 'model sample')
ax1.legend()
ax2.scatter(x_train, y_train, alpha = 0.4, label = 'data')
ax2.plot(x_train, y_hat, color = 'red', alpha = 0.8, label = 'model $\mu$')
ax2.plot(x_train, y_hat_m2sd, color = 'green', alpha = 0.8, label = 'model $\mu \pm 2 \sigma$')
```

```
ax2.plot(x_train, y_hat_p2sd, color = 'green', alpha = 0.8)
ax2.legend()
plt.show()
```





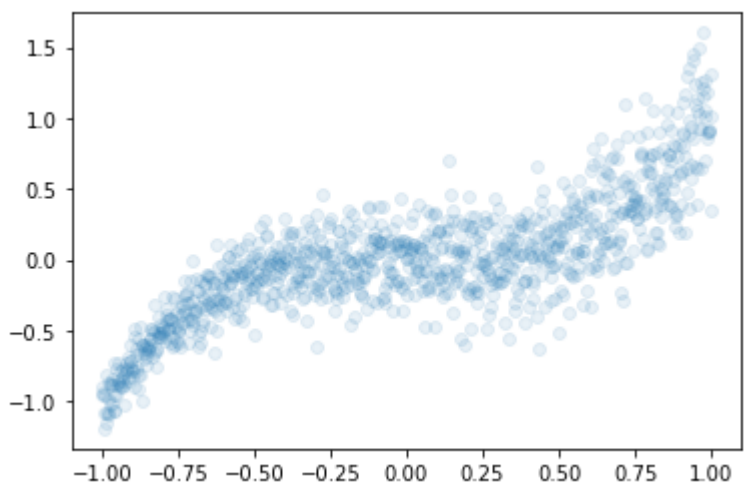
Probabilistic linear regression with nonlinear learned mean & variance

Let's change the data to being nonlinear:

$$y_i = x_i^3 + \frac{1}{10}(2 + x_i)\epsilon_i$$

where $\epsilon_i \sim N(0, 1)$ are independent and identically distributed.

In [21]:
```
# Create and plot 10000 data points
x_train = np.linspace(-1, 1, 1000)[:, np.newaxis]
y_train = np.power(x_train, 3) + 0.1 * (2 + x_train) * np.random.randn(1000)[:, np.newaxis]

plt.scatter(x_train, y_train, alpha = 0.1)
plt.show()
```



In [22]:
```
# Create probabilistic regression: normal distribution with fixed variance
model = Sequential ([
    Dense(input_shape = (1,), units = 8, activation = 'sigmoid'),
    Dense(tfpl.IndependentNormal.params_size(event_shape = 1)),
    tfpl.IndependentNormal(event_shape = 1)
])
model.compile(loss = nll, optimizer = RMSprop(learning_rate = 0.01))
model.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 8)                 16
_________________________________________________________________
dense_5 (Dense)              (None, 2)                 18
_________________________________________________________________
independent_normal_1 (Indepe ((None, 1), (None, 1))    0
=================================================================
Total params: 34
Trainable params: 34
Non-trainable params: 0
_________________________________________________________________
```

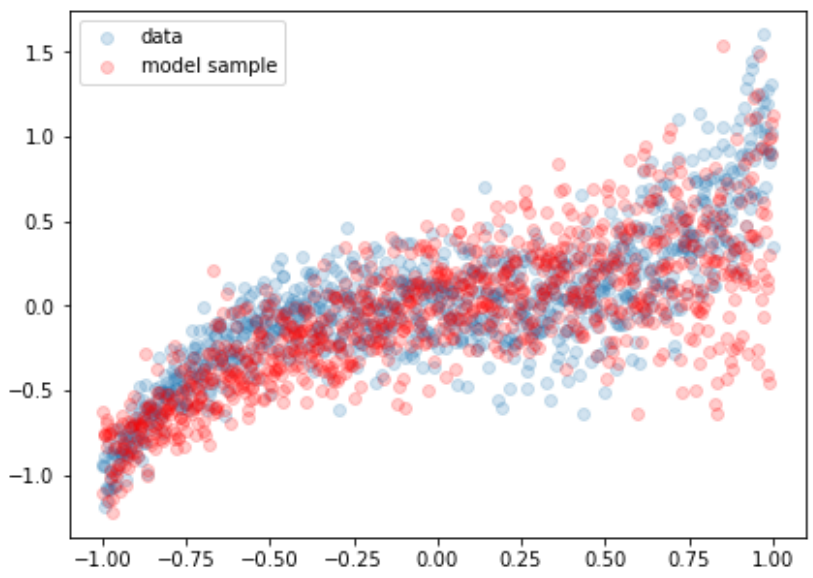
In [23]:
```
# Train model
model.fit(x_train, y_train, epochs = 200, verbose = False)
model.evaluate(x_train, y_train)
```

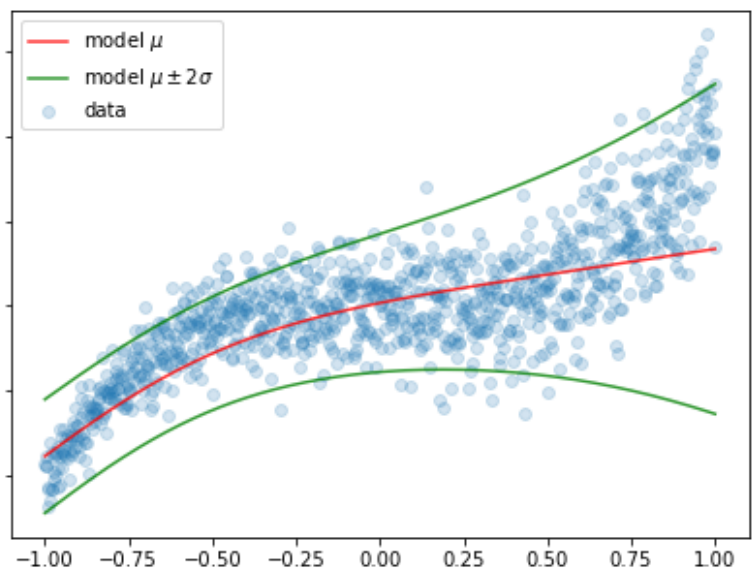
```
32/32 [==============================] - 0s 3ms/step - loss: 0.0217
```

Out[23]:
```
0.021747954189777374
```

In [24]:
```
# Plot the data and a sample from the model
y_model = model(x_train)
y_sample = y_model.sample()
y_hat = y_model.mean()
y_sd = y_model.stddev()
y_hat_m2sd = y_hat - 2 * y_sd
y_hat_p2sd = y_hat + 2 * y_sd

fig, (ax1, ax2) = plt.subplots(1, 2, figsize = (15, 5), sharey=True)
ax1.scatter(x_train, y_train, alpha = 0.2, label = 'data')
```

```
ax1.scatter(x_train, y_sample, alpha = 0.2, color = 'red', label = 'model sample')
ax1.legend()
ax2.scatter(x_train, y_train, alpha = 0.2, label = 'data')
ax2.plot(x_train, y_hat, color = 'red', alpha = 0.8, label = 'model $\mu$')
ax2.plot(x_train, y_hat_m2sd, color = 'green', alpha = 0.8, label = 'model $\mu \pm 2 \sigma$')
ax2.plot(x_train, y_hat_p2sd, color = 'green', alpha = 0.8)
ax2.legend()
plt.show()
```





The `DenseVariational` layer

In [25]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.losses import MeanSquaredError
from tensorflow.keras.optimizers import RMSprop
import numpy as np
import matplotlib.pyplot as plt
```

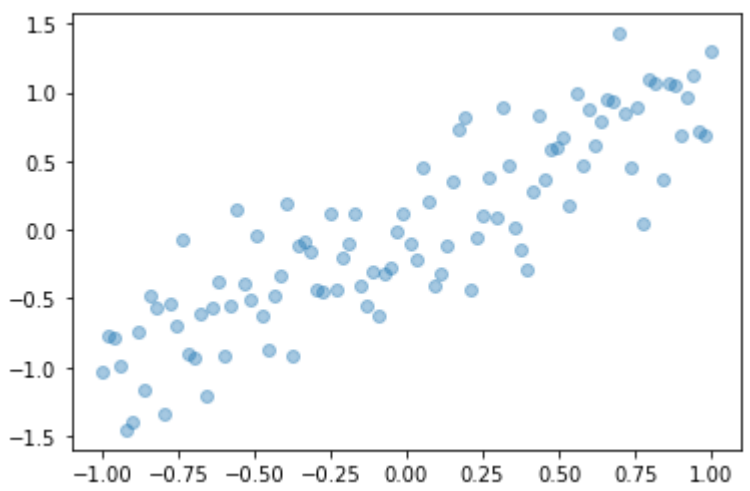
Create linear data with Gaussian noise

The data you'll be working with is the same as you used before:

$$y_i = x_i + \frac{3}{10}\epsilon_i$$

where $\epsilon_i \sim N(0, 1)$ are independent and identically distributed. We'll be running a Bayesian linear regression on this data.

In [26]:
```
# Use the same data as before -- create and plot 100 data points
x_train = np.linspace(-1, 1, 100)[:, np.newaxis]
y_train = x_train + 0.3 * np.random.randn(100)[:, np.newaxis]

plt.scatter(x_train, y_train, alpha = 0.4)
plt.show()
```



Create the prior and posterior distribution for model weights

In [27]:
```
# Define the prior weight distribution -- all N(0, 1) -- and not trainable
def prior(kernel_size, bias_size, dtype = None):
    n = kernel_size + bias_size
    prior_model = Sequential ([
        tfpl.DistributionLambda \
            (lambda t: tfd.MultivariateNormalDiag(loc = tf.zeros(n), scale_diag = tf.ones(n)))
    ])
    return prior_model
```

In [28]:
```
# Define variational posterior weight distribution -- multivariate Gaussian
def posterior(kernel_size, bias_size, dtype = None):
    n = kernel_size + bias_size
    posterior_model = Sequential ([
        tfpl.VariableLayer(tfpl.MultivariateNormalTriL.params_size(n), dtype = dtype),
        tfpl.MultivariateNormalTriL(n)
    ])
    return posterior_model
```

Aside: analytical posterior

In this tutorial, we're using a variational posterior because, in most settings, it's not possible to derive an analytical one. However, in this simple setting, it is possible. Specifically, running a Bayesian linear regression on x_i and y_i with $i = 1, \ldots, n$ and a unit Gaussian prior on both α and β:

$$y_i = \alpha + \beta x_i + \epsilon_i, \quad \epsilon_i \sim N(0, \sigma^2), \quad \alpha \sim N(0, 1), \quad \beta \sim N(0, 1)$$

gives a multivariate Gaussian posterior on α and β:

$$\begin{pmatrix} \alpha \\ \beta \end{pmatrix} \sim N(\mu, \mathbf{\Sigma})$$

where $$

\mathbf{\mu}

\mathbf{\Sigma}

$$\begin{pmatrix} \hat{n}\bar{y} \\ \hat{n}xy \end{pmatrix}$$

, \quad \mathbf{\Sigma} = \frac{1}{(\hat{n} + 1)(\hat{n} \overline{x^2} + 1) - \hat{n}^2 \bar{x}^2}

$$\begin{pmatrix} \hat{n}x^2 + 1 & -\hat{n}\bar{x} \\ -\hat{n}\bar{x} & \hat{n} + 1 \end{pmatrix}$$

. $$

In the above, $\hat{n} = \frac{n}{\sigma^2}$ and $\bar{t} = \frac{1}{n}\sum_{i=1}^{n} t_i$ for any t. In general, however, it's not possible to determine the analytical form for the posterior. For example, in models with a hidden layer with nonlinear activation function, the analytical posterior cannot be determined in general, and variational methods as below are useful.

Create the model with `DenseVariational` layers

In [29]:
```
# Create linear regression model with weight uncertainty: weights are
# distributed according to posterior (and, indirectly, prior) distribution
model = Sequential ([
    tfpl.DenseVariational (
        input_shape = (1,), units = 1,
        make_prior_fn = prior, make_posterior_fn = posterior,
        kl_weight = 1 / x_train.shape[0], kl_use_exact = True
    )
])
model.compile(loss = MeanSquaredError(), optimizer = RMSprop(learning_rate = 0.005))
model.summary()
```

```
WARNING:tensorflow:From /home/bacti/anaconda3/envs/tensor/lib/python3.7/site-packages/tensorflow/python/ops/linalg/lin
ear_operator_lower_triangular.py:158: calling LinearOperator.__init__ (from tensorflow.python.ops.linalg.linear_operat
or) with graph_parents is deprecated and will be removed in a future version.
Instructions for updating:
Do not pass `graph_parents`.  They will  no longer be used.
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_variational (DenseVari (None, 1)                 5
=================================================================
Total params: 5
Trainable params: 5
Non-trainable params: 0
_________________________________________________________________
```

Train model and inspect

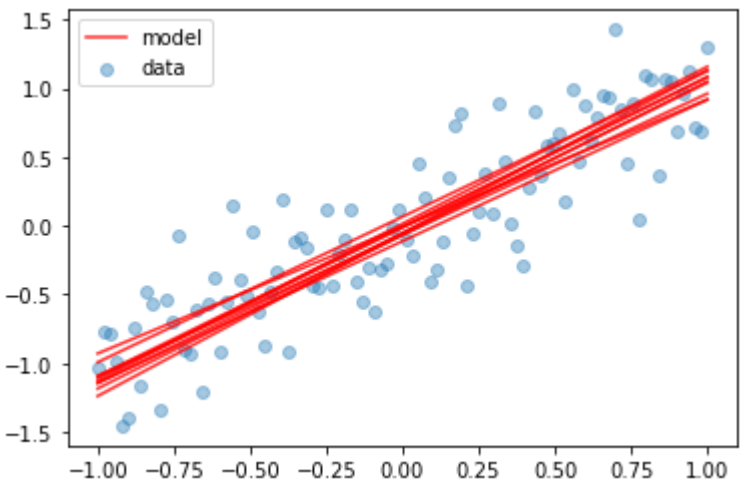
In [30]:
```
# Fit the model, just like a deterministic linear regression
model.fit(x_train, y_train, epochs = 500, verbose = False)
```

Out[30]:
```
<tensorflow.python.keras.callbacks.History at 0x7fb3f42e2550>
```

In [31]:
```
# Check out the parameters of the prior and posterior distribution
dummy_input = np.array([[0]])
model_prior = model.layers[0]._prior(dummy_input)
model_posterior = model.layers[0]._posterior(dummy_input)
print('prior mean:           ', model_prior.mean().numpy())
print('prior variance:       ', model_prior.variance().numpy())
print('posterior mean:       ', model_posterior.mean().numpy())
print('posterior covariance: ', model_posterior.covariance().numpy()[0])
print('                      ', model_posterior.covariance().numpy()[1])
```

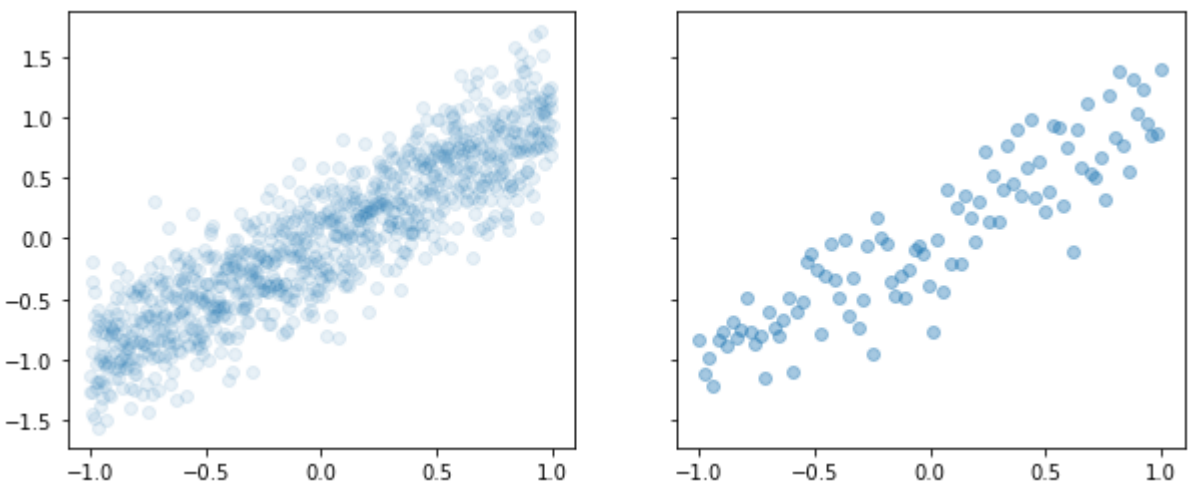
```
prior mean:            [0. 0.]
prior variance:        [1. 1.]
posterior mean:        [ 1.0078331 -0.0018866]
posterior covariance:  [0.01788209 0.00079696]
                       [0.00079696 0.00576819]
```

In [32]:
```
# Plot an ensemble of linear regressions, with weights sampled from
# the posterior distribution
plt.scatter(x_train, y_train, alpha = 0.4, label = 'data')
for _ in range(10):
    y_model = model(x_train)
    if _ == 0:
        plt.plot(x_train, y_model, color = 'red', alpha = 0.8, label = 'model')
    else:
        plt.plot(x_train, y_model, color = 'red', alpha = 0.8)
plt.legend()
plt.show()
```



Explore the effect of sample size

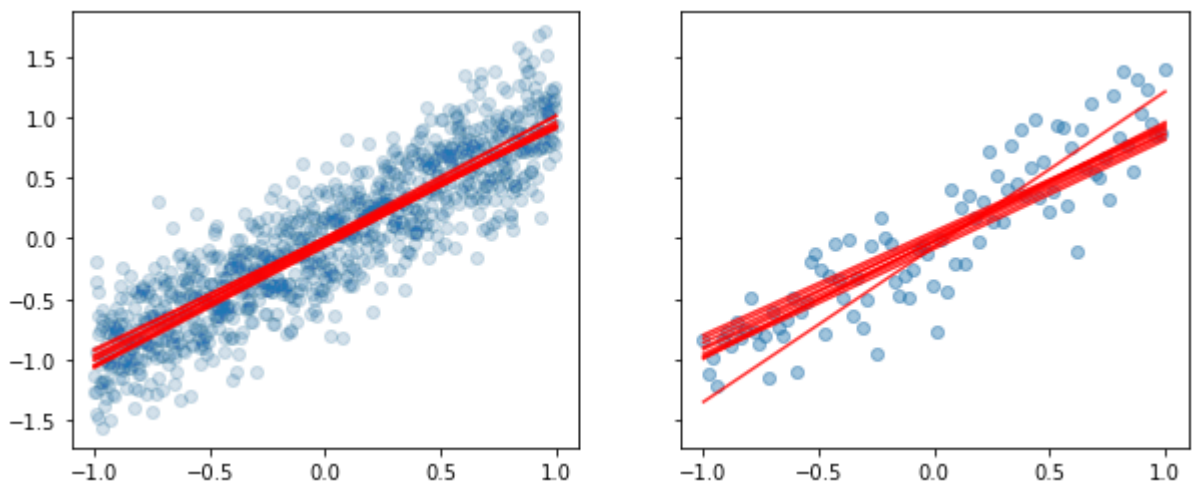
In [33]:
```
# Create two datasets, one with 1000 points, another with 100
x_train_1000 = np.linspace(-1, 1, 1000)[:, np.newaxis]
y_train_1000 = x_train_1000 + 0.3 * np.random.randn(1000)[:, np.newaxis]

x_train_100 = np.linspace(-1, 1, 100)[:, np.newaxis]
y_train_100 = x_train_100 + 0.3 * np.random.randn(100)[:, np.newaxis]

fig, (ax1, ax2) = plt.subplots(1, 2, figsize = (10, 4), sharex = True, sharey = True)
ax1.scatter(x_train_1000, y_train_1000, alpha = 0.1)
ax2.scatter(x_train_100, y_train_100, alpha = 0.4)
plt.show()
```



In [34]:
```
# Train a model on each dataset

model_1000 = Sequential ([
    tfpl.DenseVariational (
        input_shape = (1,), units = 1,
        make_prior_fn = prior, make_posterior_fn = posterior, kl_weight = 1 / 1000
    )
])

model_100 = Sequential ([
    tfpl.DenseVariational (
        input_shape = (1,), units = 1,
        make_prior_fn = prior, make_posterior_fn = posterior, kl_weight = 1 / 100
    )
])

model_1000.compile(loss = MeanSquaredError(), optimizer = RMSprop(learning_rate = 0.005))
model_100.compile(loss = MeanSquaredError(), optimizer = RMSprop(learning_rate = 0.005))

model_1000.fit(x_train_1000, y_train_1000, epochs = 50, verbose = False)
model_100.fit(x_train_100, y_train_100, epochs = 500, verbose = False)
```

Out[34]:
```
<tensorflow.python.keras.callbacks.History at 0x7fb3d85f2b10>
```

In [35]:
```
# Plot an ensemble of linear regressions from each model
fig, (ax1, ax2) = plt.subplots(1, 2, figsize = (10, 4), sharex = True, sharey = True)
for _ in range(10):
    y_model_1000 = model_1000(x_train_1000)
    ax1.scatter(x_train_1000, y_train_1000, color = 'C0', alpha = 0.02)
    ax1.plot(x_train_1000, y_model_1000, color = 'red', alpha = 0.8)
    y_model_100 = model_100(x_train_100)
    ax2.scatter(x_train_100, y_train_100, color = 'C0', alpha = 0.05)
    ax2.plot(x_train_100, y_model_100, color = 'red', alpha = 0.8)
plt.show()
```



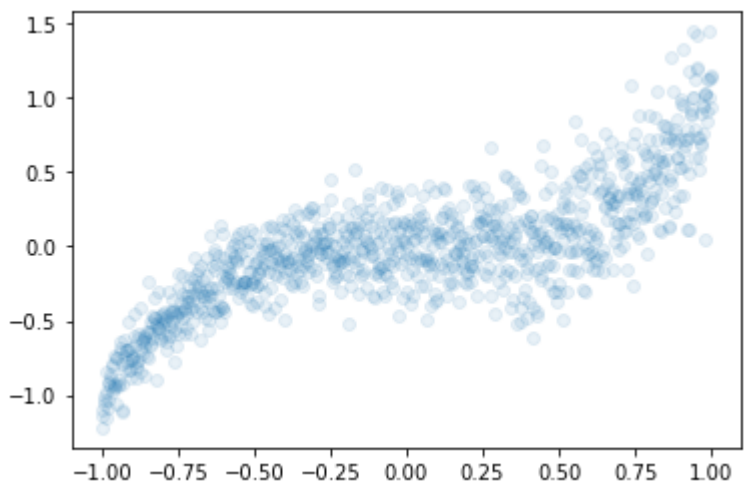
Put it all together: nonlinear probabilistic regression with weight uncertainty

Let's change the data to being nonlinear:

$$y_i = x_i^3 + \frac{1}{10}(2 + x_i)\epsilon_i$$

where $\epsilon_i \sim N(0, 1)$ are independent and identically distributed.

In [36]:
```
# Create and plot 1000 data points
x_train = np.linspace(-1, 1, 1000)[:, np.newaxis]
y_train = np.power(x_train, 3) + 0.1 * (2 + x_train) * np.random.randn(1000)[:, np.newaxis]

plt.scatter(x_train, y_train, alpha = 0.1)
plt.show()
```



In [37]:
```
# Create probabilistic regression with one hidden layer, weight uncertainty
model = Sequential ([
    tfpl.DenseVariational (
        units = 8, input_shape = (1,),
        make_prior_fn = prior, make_posterior_fn = posterior,
        kl_weight = 1 / x_train.shape[0], activation = 'sigmoid'
    ),
    tfpl.DenseVariational (
        units = tfpl.IndependentNormal.params_size(1),
        make_prior_fn = prior, make_posterior_fn = posterior,
        kl_weight = 1 / x_train.shape[0]
    ),
    tfpl.IndependentNormal(1)
])

def nll(y_true, y_pred):
    return -y_pred.log_prob(y_true)

model.compile(loss = nll, optimizer = RMSprop(learning_rate = 0.005))
model.summary()
```

```
Model: "sequential_8"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_variational_3 (DenseVa (None, 8)                 152
_________________________________________________________________
dense_variational_4 (DenseVa (None, 2)                 189
_________________________________________________________________
independent_normal_2 (Indepe ((None, 1), (None, 1))    0
=================================================================
Total params: 341
Trainable params: 341
Non-trainable params: 0
_________________________________________________________________
```

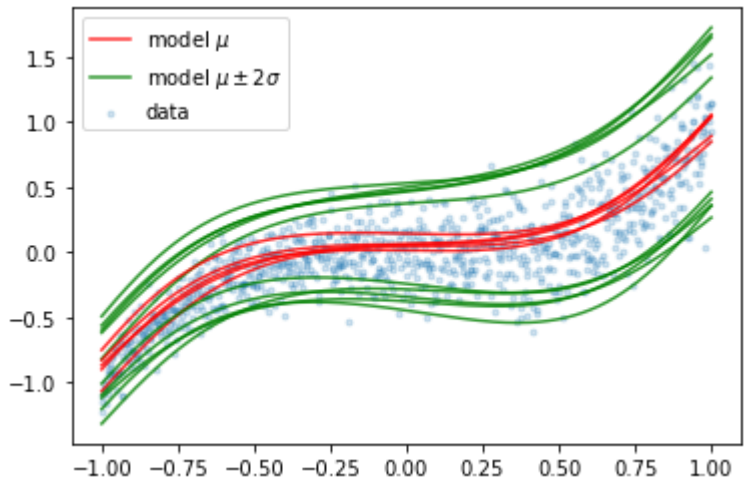
In [38]:
```
# Train the model
model.fit(x_train, y_train, epochs = 1000, verbose = False)
model.evaluate(x_train, y_train)
```

```
32/32 [==============================] - 0s 4ms/step - loss: -0.0065
```

Out[38]:
```
-0.00650815200060606
```

In [39]:
```
# Plot an ensemble of trained probabilistic regressions
plt.scatter(x_train, y_train, marker = '.', alpha = 0.2, label = 'data')
for _ in range(5):
    y_model = model(x_train)
```

```
    y_hat = y_model.mean()
    y_hat_m2sd = y_hat - 2 * y_model.stddev()
    y_hat_p2sd = y_hat + 2 * y_model.stddev()
    if _ == 0:
        plt.plot(x_train, y_hat, color = 'red', alpha = 0.8, label = 'model $\mu$')
        plt.plot(x_train, y_hat_m2sd, color = 'green', alpha = 0.8, label = 'model $\mu \pm 2 \sigma$')
        plt.plot(x_train, y_hat_p2sd, color = 'green', alpha = 0.8)
    else:
        plt.plot(x_train, y_hat, color = 'red', alpha = 0.8)
        plt.plot(x_train, y_hat_m2sd, color = 'green', alpha = 0.8)
        plt.plot(x_train, y_hat_p2sd, color = 'green', alpha = 0.8)
plt.legend()
plt.show()
```



Reparameterization layers

In [40]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv1D, MaxPooling1D, Flatten
from tensorflow.keras.losses import SparseCategoricalCrossentropy
from tensorflow.keras.optimizers import RMSprop
import os
import numpy as np
import matplotlib.pyplot as plt
```

Load in the HAR dataset

You'll be working with the Human Activity Recognition (HAR) Using Smartphones (https://archive.ics.uci.edu/ml/datasets/Human+Activity+Recognition+Using+Smartphones) dataset. It consists of the readings from an accelerometer (which measures acceleration) carried by a human doing different activities. The six activities are walking horizontally, walking upstairs, walking downstairs, sitting, standing and laying down. The accelerometer is inside a smartphone, and, every 0.02 seconds (50 times per second), it takes six readings: linear and gyroscopic acceleration in the x, y and z directions. See this link (https://archive.ics.uci.edu/ml/datasets/Human+Activity+Recognition+Using+Smartphones) for details and download. If you use it in your own research, please cite the following paper:

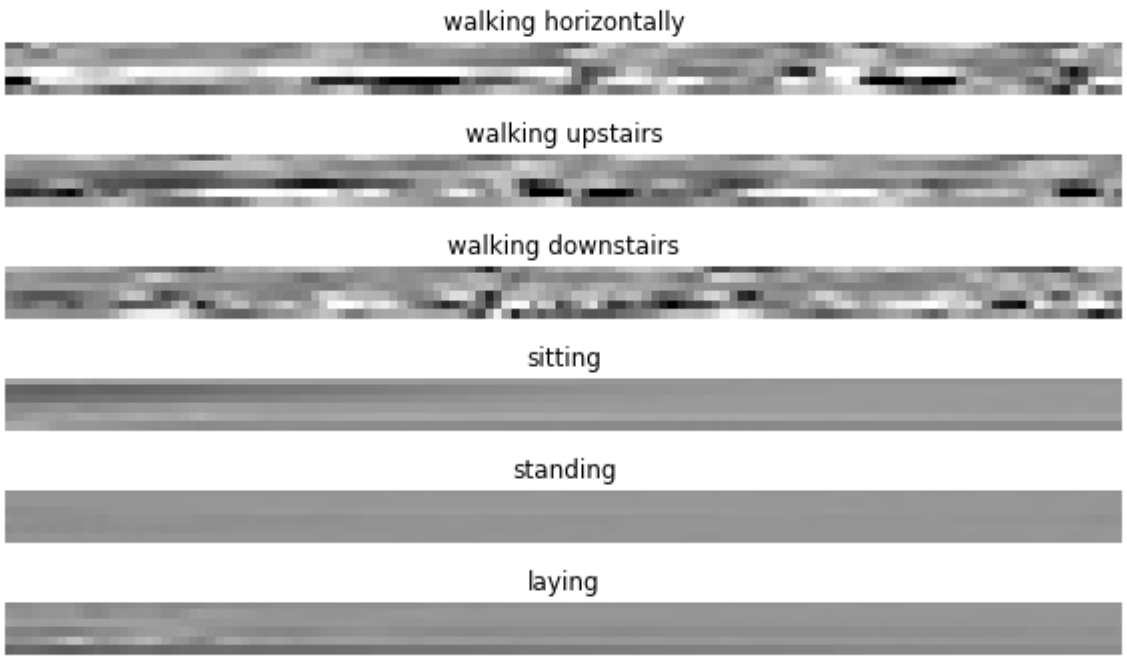
- Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra and Jorge L. Reyes-Ortiz. A Public Domain Dataset for Human Activity Recognition Using Smartphones. 21th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, ESANN 2013. Bruges, Belgium 24-26 April 2013.

The goal is to use the accelerometer data to predict the activity.

In [41]:
```
# Load the HAR dataset and create some data processing functions

# Function to load the data from file
def load_HAR_data():
    data_dir = 'data/HAR/'
    x_train = np.load(os.path.join(data_dir, 'x_train.npy'))[..., :6]
    y_train = np.load(os.path.join(data_dir, 'y_train.npy')) - 1
    x_test  = np.load(os.path.join(data_dir, 'x_test.npy'))[..., :6]
    y_test  = np.load(os.path.join(data_dir, 'y_test.npy')) - 1
    return (x_train, y_train), (x_test, y_test)

# Dictionary containing the labels and the associated activities
label_to_activity = {
    0: 'walking horizontally', 1: 'walking upstairs', 2: 'walking downstairs',
    3: 'sitting', 4: 'standing', 5: 'laying'
}

# Function to change integer labels to one-hot labels
def integer_to_onehot(data_integer):
    data_onehot = np.zeros(shape=(data_integer.shape[0], data_integer.max() + 1))
    for row in range(data_integer.shape[0]):
        integer = int(data_integer[row])
        data_onehot[row, integer] = 1
    return data_onehot

# Load the data
(x_train, y_train), (x_test, y_test) = load_HAR_data()
y_train_oh = integer_to_onehot(y_train)
y_test_oh = integer_to_onehot(y_test)
```

In [42]:
```
# Inspect some of the data by making plots
def make_plots(num_examples_per_category):
    for label in range(6):
```

```
        x_label = x_train[y_train[:, 0] == label]
        for i in range(num_examples_per_category):
            fig, ax = plt.subplots(figsize = (10, 1))
            ax.imshow(x_label[100 * i].T, cmap = 'Greys', vmin = -1, vmax = 1)
            ax.axis('off')
            if i == 0:
                ax.set_title(label_to_activity[label])
            plt.show()

make_plots(1)
```



1D deterministic convolutional neural network

In [43]:
```
# Create standard deterministic model with:
# - Conv1D
# - MaxPooling
# - Flatten
# - Dense with Softmax
model = Sequential ([
    Conv1D(input_shape = (128, 6), filters = 8, kernel_size = 16, activation = 'relu'),
    MaxPooling1D(pool_size = 16),
    Flatten(),
    Dense(units = 6, activation = 'softmax')
])
model.summary()
```

```
Model: "sequential_9"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d (Conv1D)              (None, 113, 8)            776
_________________________________________________________________
max_pooling1d (MaxPooling1D) (None, 7, 8)              0
_________________________________________________________________
flatten (Flatten)            (None, 56)                0
_________________________________________________________________
dense_6 (Dense)              (None, 6)                 342
=================================================================
Total params: 1,118
Trainable params: 1,118
Non-trainable params: 0
_________________________________________________________________
```

Probabilistic 1D convolutional neural network, with both weight and output uncertainty

In [44]:
```
# Create probablistic model with the following layers:
#  - Conv1D
#  - MaxPooling
#  - Flatten
#  - Dense
#  - OneHotCategorical
divergence_fn = lambda q, p, _: tfd.kl_divergence(q, p) / x_train.shape[0]
model = Sequential ([
    tfpl.Convolution1DReparameterization (
        input_shape = (128, 6), filters = 8, kernel_size = 16, activation = 'relu',
        kernel_prior_fn = tfpl.default_multivariate_normal_fn,
        kernel_posterior_fn = tfpl.default_mean_field_normal_fn(is_singular = False),
        kernel_divergence_fn = divergence_fn,
        bias_prior_fn = tfpl.default_multivariate_normal_fn,
        bias_posterior_fn = tfpl.default_mean_field_normal_fn(is_singular = False),
        bias_divergence_fn = divergence_fn,
    ),
    MaxPooling1D(pool_size = 16),
    Flatten(),
    tfpl.DenseReparameterization (
        units = tfpl.OneHotCategorical.params_size(6), activation = None,
        kernel_prior_fn = tfpl.default_multivariate_normal_fn,
        kernel_posterior_fn = tfpl.default_mean_field_normal_fn(is_singular = False),
        kernel_divergence_fn = divergence_fn,
        bias_prior_fn = tfpl.default_multivariate_normal_fn,
        bias_posterior_fn = tfpl.default_mean_field_normal_fn(is_singular = False),
        bias_divergence_fn = divergence_fn,
    ),
    tfpl.OneHotCategorical(6)
])
model.summary()
```

```
WARNING:tensorflow:From /home/bacti/anaconda3/envs/tensor/lib/python3.7/site-packages/tensorflow_probability/python/layers/util.py:106: Layer.add_variable (from tensorflow.python.keras.engine.base_layer) is deprecated and will be removed in a future version.
Instructions for updating:
Please use `layer.add_weight` method instead.
Model: "sequential_10"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d_reparameterization (C (None, 113, 8)            1552
_________________________________________________________________
max_pooling1d_1 (MaxPooling1 (None, 7, 8)              0
_________________________________________________________________
flatten_1 (Flatten)          (None, 56)                0
_________________________________________________________________
dense_reparameterization (De (None, 6)                 684
_________________________________________________________________
one_hot_categorical (OneHotC ((None, 6), (None, 6))    0
=================================================================
Total params: 2,236
Trainable params: 2,236
Non-trainable params: 0
_________________________________________________________________
```

In [45]:
```
# Replace analytical Kullback-Leibler divergence with approximated one
def kl_approx(q, p, q_tensor):
    return tf.reduce_mean(q.log_prob(q_tensor) - p.log_prob(q_tensor))

divergence_fn = lambda q, p, q_tensor : kl_approx(q, p, q_tensor) / x_train.shape[0]
```

In [46]:
```
# Compile the model using the negative loglikelihood
def nll(y_true, y_pred):
    return -y_pred.log_prob(y_true)

model.compile (
    loss = nll, optimizer = RMSprop(learning_rate = 0.005),
    metrics = ['accuracy'], experimental_run_tf_function = False
)
```

In [47]:
```
# Train the model
model.fit(x_train, y_train_oh, epochs = 20, verbose = False)
model.evaluate(x_train, y_train_oh)
model.evaluate(x_test, y_test_oh)
```

```
230/230 [==============================] - 2s 9ms/step - loss: 0.6452 - accuracy: 0.7514
93/93 [==============================] - 1s 9ms/step - loss: 1.0390 - accuracy: 0.7241
```

Out[47]:
```
[1.0390419960021973, 0.7241262197494507]
```

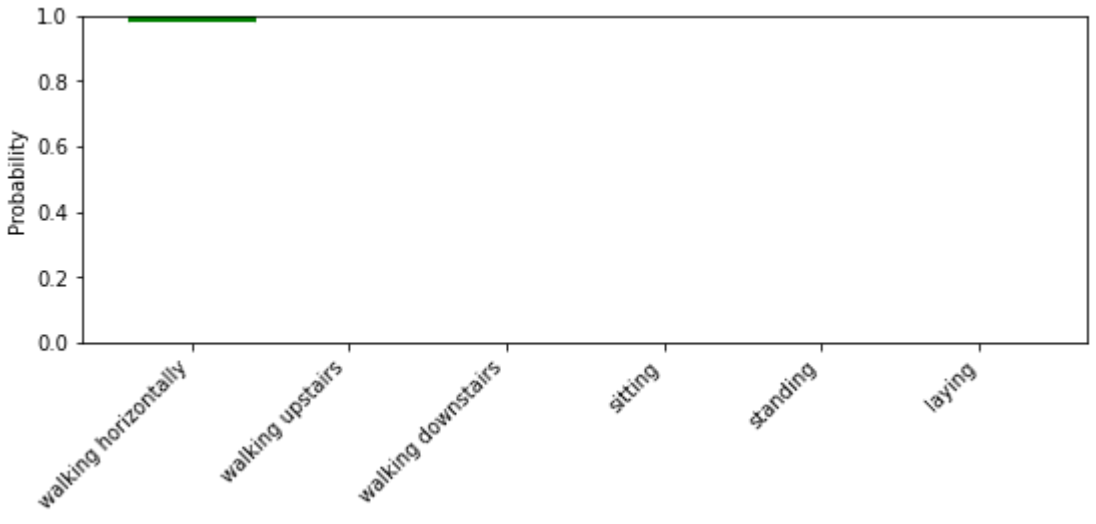
Inspect model performance

In [48]:
```
# Define function to analyse model predictions versus true labels
def analyse_model_predictions(image_num):

    # Show the accelerometer data
    print('------------------------------')
    print('Accelerometer data:')
    fig, ax = plt.subplots(figsize = (10, 1))
    ax.imshow(x_test[image_num].T, cmap = 'Greys', vmin = -1, vmax = 1)
    ax.axis('off')
    plt.show()

    # Print the true activity
    print('------------------------------')
    print('True activity:', label_to_activity[y_test[image_num, 0]])
    print('')

    # Print the probabilities the model assigns
    print('------------------------------')
    print('Model estimated probabilities:')
    # Create ensemble of predicted probabilities
    predicted_probabilities = np.empty(shape = (200, 6))
    for i in range(200):
        predicted_probabilities[i] = model(x_test[image_num][np.newaxis, ...]).mean().numpy()[0]
    pct_2p5 = np.array([np.percentile(predicted_probabilities[:, i], 2.5) for i in range(6)])
    pct_97p5 = np.array([np.percentile(predicted_probabilities[:, i], 97.5) for i in range(6)])
    # Make the plots
    fig, ax = plt.subplots(figsize = (9, 3))
    bar = ax.bar(np.arange(6), pct_97p5, color = 'red')
    bar[y_test[image_num, 0]].set_color('green')
    bar = ax.bar(np.arange(6), pct_2p5 - 0.02, color = 'white', linewidth = 1, edgecolor = 'white')
    ax.set_xticklabels (
        [''] + [activity for activity in label_to_activity.values()],
        rotation = 45,
        horizontalalignment = 'right'
    )
    ax.set_ylim([0, 1])
    ax.set_ylabel('Probability')
    plt.show()
```

In [49]:
```
analyse_model_predictions(image_num = 79)
```

```
------------------------------
Accelerometer data:
```

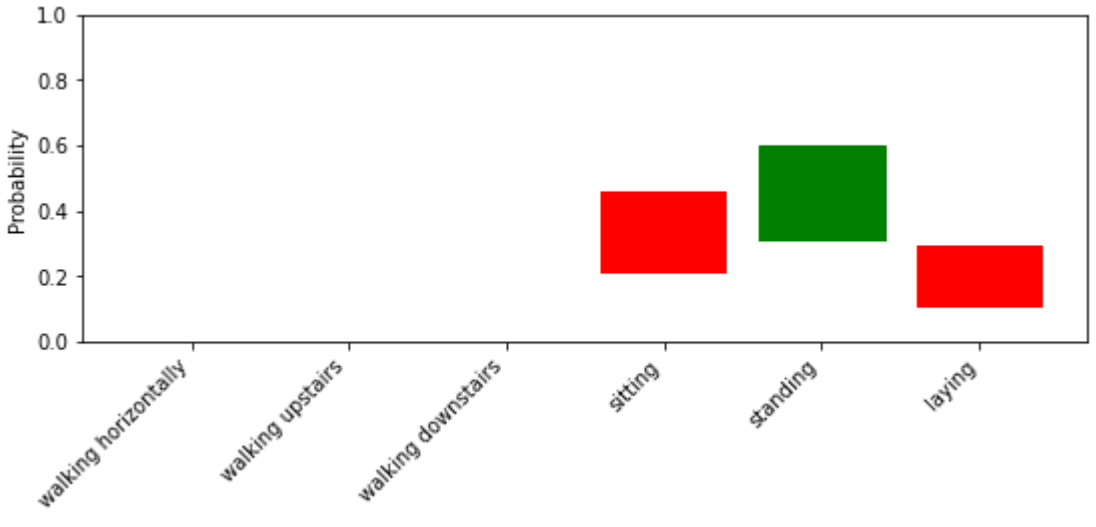
```
------------------------------
True activity: walking horizontally

------------------------------
Model estimated probabilities:
/home/bacti/.local/lib/python3.7/site-packages/ipykernel_launcher.py:34: UserWarning: FixedFormatter should only be us
ed together with FixedLocator
```



In [50]:
```
analyse_model_predictions(image_num = 633)
```

```
------------------------------
Accelerometer data:
```

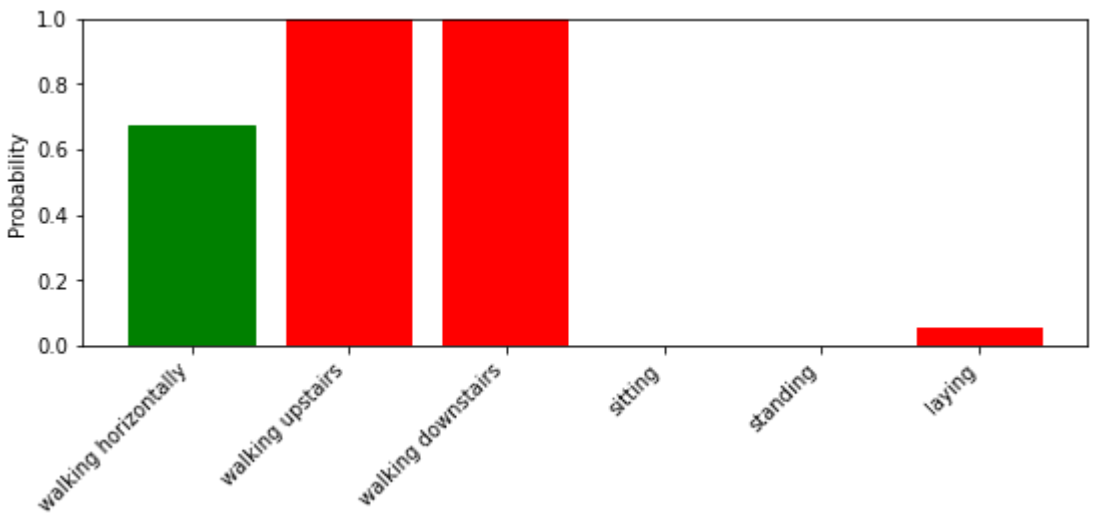
```
------------------------------
True activity: standing

------------------------------
Model estimated probabilities:
/home/bacti/.local/lib/python3.7/site-packages/ipykernel_launcher.py:34: UserWarning: FixedFormatter should only be us
ed together with FixedLocator
```



In [51]:
```
analyse_model_predictions(image_num = 1137)
```

```
------------------------------
Accelerometer data:
```

```
------------------------------
True activity: walking horizontally

------------------------------
Model estimated probabilities:
/home/bacti/.local/lib/python3.7/site-packages/ipykernel_launcher.py:34: UserWarning: FixedFormatter should only be us
ed together with FixedLocator
```



Scale bijectors and LinearOperator

This reading is an introduction to scale bijectors, as well as the `LinearOperator` class, which can be used with them.

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

```
tfb = tfp.bijectors
print("TF version:", tf.__version__)
print("TFP version:", tfp.__version__)
```

```
TF version: 2.3.0
TFP version: 0.11.0
```

Introduction

You have now seen how bijectors can be used to transform tensors and tensor spaces. Until now, you've only seen this in the scalar case, where the bijector acts on a single value. When the tensors you fed into the bijectors had multiple components, the bijector acted on each component individually by applying batch operations to scalar values. For probability distributions, this corresponds to a scalar event space.

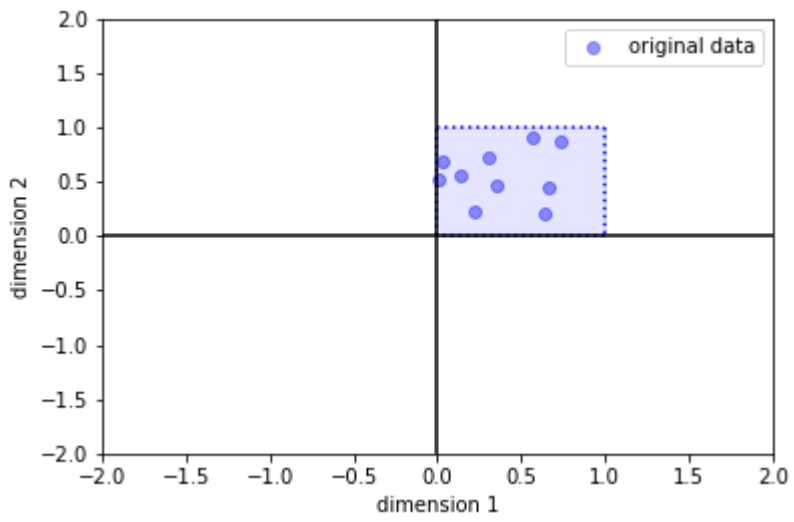
However, bijectors can also act on higher-dimensional space. You've seen, for example, the multivariate normal distribution, for which samples are tensors with more than one component. You'll need higher-dimensional bijectors to work with such distributions. In this reading, you'll see how bijectors can be used to generalise scale transformations to higher dimensions. You'll also see the `LinearOperator` class, which you can use to construct highly general scale bijectors. In this reading, you'll walk through the code, and we'll use figure examples to demonstrate these transformations.

This reading contains many images, as this allows you to visualise how a space is transformed. For this reason, the examples are limited to two dimensions, since these allow easy plots. However, these ideas generalise naturally to higher dimensions. Let's start by creating a point that is randomly distributed across the unit square $[0, 1] \times [0, 1]$:

In [2]:
```
# Create the base distribution and a single sample
uniform = tfd.Uniform \
    (low = [0.0, 0.0], high = [1.0, 1.0], name = 'uniform2d')
x = uniform.sample()
x
```

Out[2]:
```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([0.79194343, 0.41313922], dtype=float32)>
```

We will be applying linear transformations to this data. To get a feel for how these transformations work, we show ten example sample points, and plot them, as well as the domain of the underlying distribution:



Each of the ten points is hence represented by a two-dimensional vector. Let $\mathbf{x} = [x_1, x_2]^T$ be one of these points. Then scale bijectors are linear transformations of $\mathbf{x}$, which can be represented by a 2×2 matrix B. The forward bijection to $\mathbf{y} = [y_1, y_2]^T$ is

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = B\mathbf{x} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

This is important to remember: any two-dimensional scale bijector can be represented by a 2×2 matrix. For this reason, we'll sometimes use the term "matrix" to refer to the bijector itself. You'll be seeing how these points and domain are transformed under different bijectors in two dimensions.

The `ScaleMatvec` bijectors

The `ScaleMatvecDiag` bijector

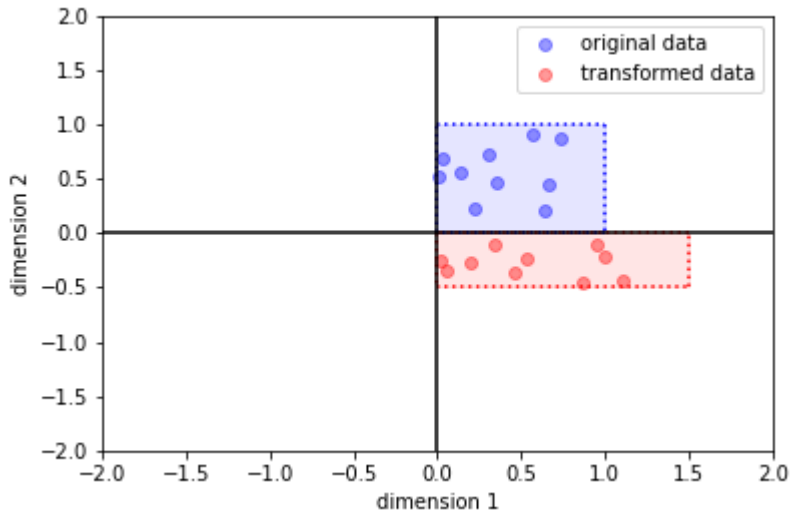
We'll start with a simple scale bijector created using the `ScaleMatvecDiag` class:

In [3]:
```
# Create the ScaleMatvecDiag bijector
bijector = tfb.ScaleMatvecDiag(scale_diag = [1.5, -0.5])
```

which creates a bijector represented by the diagonal matrix

$$B = \begin{bmatrix} 1.5 & 0 \\ 0 & -0.5 \end{bmatrix}.$$

We can apply this to the data using `y = bijector(x)` for each of the ten points. This transforms the data as follows:



You can see what happened here: the first coordinate is multiplied by 1.5 while the second is multipled by -0.5, flipping it through the horizontal axis.

In [4]:
```
# Apply the bijector to the sample point
y = bijector(x)
y
```

Out[4]:
```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([ 1.1879151 , -0.20656961], dtype=float32)>
```

The `ScaleMatvecTriL` bijector

In the previous example, the bijector matrix was diagonal, which essentially performs an independent scale operation on each of the two dimensions. The domain under the bijection remains rectangular. However, not all scale tarnsformations have to be like this. With a non-diagonal matrix, the domain will transform to a quadrilateral. One way to do this is by using the `tfb.ScaleMatvecTriL` class, which implements a bijection based on a lower-triangular matrix. For example, to implement the lower-triangular matrix

$$B = \begin{bmatrix} -1 & 0 \\ -1 & -1 \end{bmatrix}$$

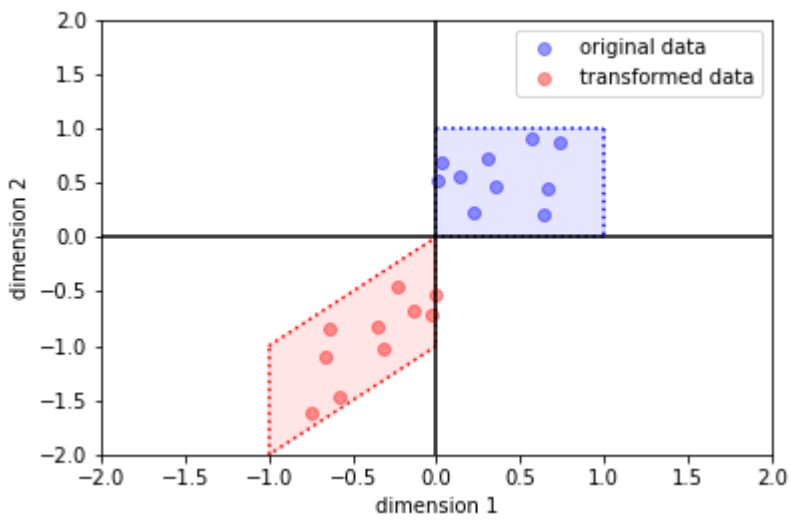
you can use the `tfb.ScaleMatvecTriL` bijector as follows:

In [5]:
```
# Create the ScaleMatvecTriL bijector
bijector = tfb.ScaleMatvecTriL(scale_tril = [[-1., 0.], [-1., -1.]])
```

In [6]:
```
# Apply the bijector to the sample x
y = bijector(x)
y
```

Out[6]:
```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([-0.79194343, -1.2050827 ], dtype=float32)>
```

A graphical overview of this change is:



Inverse and composition

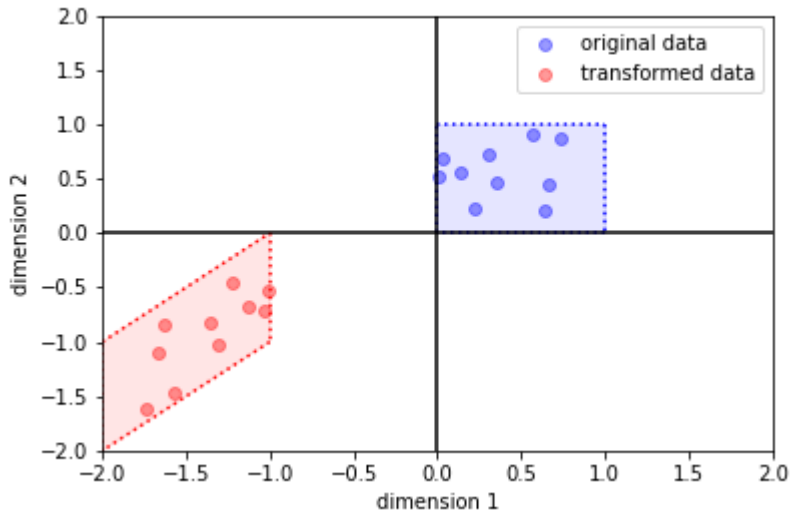
Scale transformations always map the point $[0, 0]$ to itself and are only one particular class of bijectors. As you saw before, you can create more complicated bijections by composing one with another. This works just like you would expect. For example, you can compose a scale transformation with a shift to the left (by one unit) as follows:

In [7]:
```
# Create a scale and shift bijector
scale_bijector = tfb.ScaleMatvecTriL \
    (scale_tril = [[-1., 0.], [-1., -1.]])
shift_bijector = tfb.Shift([-1., 0.])
bijector = shift_bijector(scale_bijector)
```

In [8]:
```
# Apply the bijector to the sample x
y = bijector(x)
y
```

Out[8]:
```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([-1.7919434, -1.2050827], dtype=float32)>
```

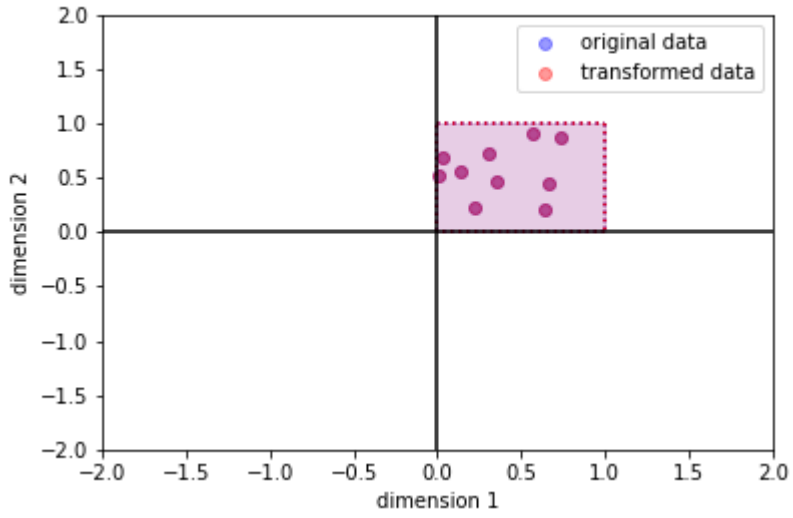
which has the expected result:



Furthermore, bijectors are always invertible (with just a few special cases, see e.g. `Absolute Value` (https://www.tensorflow.org/probability/api_docs/python/tfp/bijectors/AbsoluteValue)), and these scale transformations are no exception. For example, running

In [9]:
```
# Apply the inverse transformation to the image of x
bijector = tfb.ScaleMatvecTriL \
    (scale_tril = [[-1., 0.], [-1., -1.]])
y = bijector.inverse(bijector(x))
```

recovers `x` :



so that the original and transformed data is the same.

In [10]:
```
# Check that all y and x values are the same
tf.reduce_all(y == x)
```

Out[10]:
```
<tf.Tensor: shape=(), dtype=bool, numpy=True>
```

The `LinearOperator` class and `ScaleMatvecLinearOperator` bijector

The examples you just saw used the `ScaleMatvecDiag` and `ScaleMatvecTriL` bijectors, whose transformations can be represented by diagonal and lower-triangular matrices respectively. These are convenient since it's easy to check whether such matrices are invertible (a requirement for a bijector). However, this comes at a cost of generality: there are acceptable bijectors whose matrices are not diagonal or lower-triangular. To construct these more general bijectors, you can use the `ScaleMatvecLinearOperator` class, which operates on instances of `tf.linalg.LinearOperator` .

The `LinearOperator` is a class that allows the creation and manipulation of linear operators in TensorFlow. It's rare to call the class directly, but its subclasses represent many of the common linear operators. It's programmed in a way to have computational advantages when working with big linear operators, although we won't discuss these here. What matters now is that we can use these linear operators to define bijectors using the `ScaleMatvecLinearOperator` class. Let's see how this works.

The `LinearOperatorDiag` class

First, let's use this framework to recreate our first bijector, represented by the diagonal matrix

$$B = \begin{bmatrix} 1.5 & 0 \\ 0 & -0.5 \end{bmatrix}.$$

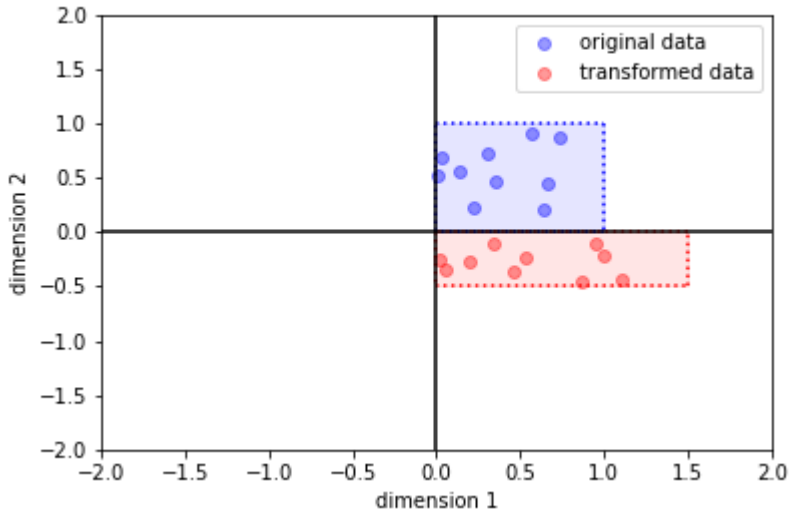
You can do this using the `ScaleMatvecLinearOperator` as follows. First, we'll create the linear operator that represents the scale transformation using

In [11]:
```
scale = tf.linalg.LinearOperatorDiag(diag = [1.5, -0.5])
```

where `LinearOperatorDiag` is one of the subclasses of `LinearOperator` . As the name suggests, it implements a diagonal matrix. We then use this to create the bijector using the `tfb.ScaleMatvecLinearOperator` :

In [12]:
```
# Create the ScaleMatvecLinearOperator bijector
bijector = tfb.ScaleMatvecLinearOperator(scale)
```

This bijector is the same as the first one above:



In [13]:
```
# Apply the bijector to the sample x
y = bijector(x)
y
```

Out[13]:
```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([ 1.1879151 , -0.20656961], dtype=float32)>
```

The `LinearOperatorFullMatrix` class

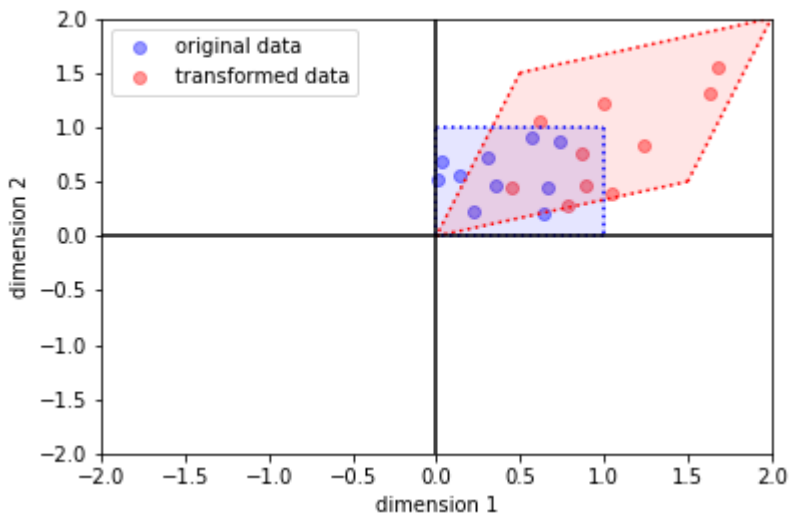
We can also use this framework to create a bijector represented by a custom matrix. Suppose we have the matrix

$$B = \begin{bmatrix} 0.5 & 1.5 \\ 1.5 & 0.5 \end{bmatrix}$$

which is neither diagonal nor lower-triangular. We can implement a bijector for it using the `ScaleMatvecLinearOperator` class by using another subclass of `LinearOperator`, namely the `LinearOperatorFullMatrix`, as follows:

In [14]:
```
# Create a ScaleMatvecLinearOperator bijector
B = [[0.5, 1.5], [1.5, 0.5]]
scale = tf.linalg.LinearOperatorFullMatrix(matrix = B)
bijector = tfb.ScaleMatvecLinearOperator(scale)
```

which leads to the following transformation:



In [15]:
```
# Apply the bijector to the sample x
y = bijector(x)
```

Out[15]:
```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([1.0156806, 1.3944848], dtype=float32)>
```

Batch operations and broadcasting

As you've seen before, it's important to be very careful with shapes in TensorFlow Probability. That's because there are three possible components to a shape: the event shape (dimensionality of the random variable), sample shape (dimensionality of the samples drawn) and batch shape (multiple distributions can be considered in one object). This subtlety is especially important for bijectors, but can be harnassed to make powerful, and very computationally efficient, transformations of spaces. Let's examine this a little bit in this section.

In the previous examples, we applied a bijector to a two-dimensional data point $\mathbf{x}$ to create a two-dimensional data point $\mathbf{y}$. This was done using $\mathbf{y} = B\mathbf{x}$ where B is the 2×2 matrix that represents the scale bijector. This is simply matrix multiplication. To implement this, we created a tensor `x` with `x.shape == [2]` and a bijector using a matrix of shape `B.shape == [2, 2]`. This generalises straightforwardly to higher dimensions: if $\mathbf{x}$ is n-dimensional, the bijection matrix must be of shape $n \times n$ for some $n > 0$. In this case, $\mathbf{y}$ is n-dimensional.

But what if you wanted to apply the same bijection to ten $\mathbf{x}$ values at once? You can then arrange all these samples into a single tensor `x` with `x.shape == [10, 2]` and create a bijector as usual, with a matrix of shape `B.shape == [2, 2]`.

In [16]:
```
# Create 10 samples from the uniform distribution
x = uniform.sample(10)
x
```

Out[16]:
```
<tf.Tensor: shape=(10, 2), dtype=float32, numpy=
array([[0.6960416 , 0.00880921],
```

```
       [0.48266065, 0.3516494 ],
       [0.06291747, 0.3056345 ],
       [0.37396407, 0.43012333],
       [0.14035296, 0.29497242],
       [0.19567811, 0.7274327 ],
       [0.69855285, 0.28542566],
       [0.5855237 , 0.06669152],
       [0.53690004, 0.43457592],
       [0.34628356, 0.70255816]], dtype=float32)>
```

In [17]:
```
# Recreate the diagonal matrix transformation with LinearOperatorDiag
scale = tf.linalg.LinearOperatorDiag(diag = [1.5, -0.5])
scale.to_dense()
```

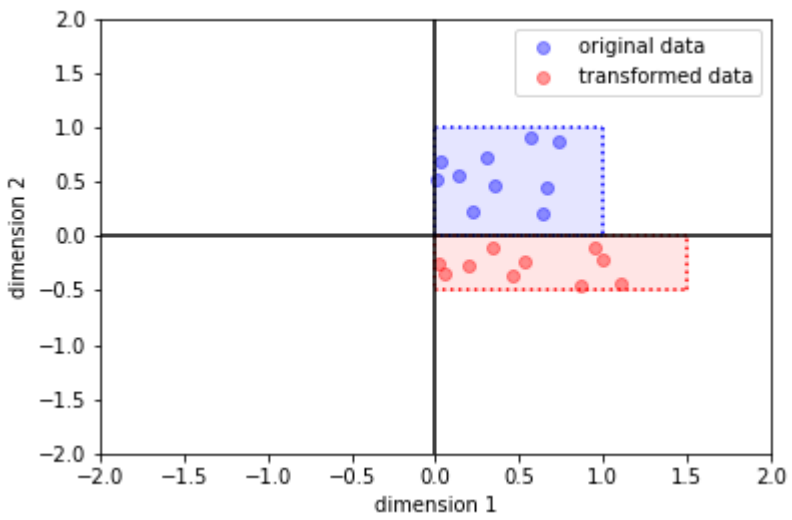
Out[17]:
```
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[ 1.5,  0. ],
       [ 0. , -0.5]], dtype=float32)>
```

In [18]:
```
# Create the ScaleMatvecLinearOperator bijector
bijector = tfb.ScaleMatvecLinearOperator(scale)
```

In [19]:
```
# Apply the bijector to the 10 samples
y = bijector(x)
y
```

Out[19]:
```
<tf.Tensor: shape=(10, 2), dtype=float32, numpy=
array([[ 1.0440624 , -0.0044046 ],
       [ 0.723991  , -0.1758247 ],
       [ 0.09437621, -0.15281725],
       [ 0.5609461 , -0.21506166],
       [ 0.21052945, -0.14748621],
       [ 0.29351717, -0.36371636],
       [ 1.0478293 , -0.14271283],
       [ 0.8782856 , -0.03334576],
       [ 0.80535007, -0.21728796],
       [ 0.51942533, -0.35127908]], dtype=float32)>
```

This gives us the same plot we had before:



For matrix multiplication to work, we need `B.shape[-1] == x.shape[-1]`, and the output tensor has last dimension `y.shape[-1] == B.shape[-2]`. For invertibility, we also need the matrix `B` to be square. Any dimensions except for the last one on `x` become sample/batch dimensions: the operation is broadcast across these dimensions as we are used to. It's probably easiest to understand through a table of values, where `s`, `b`, `m`, and `n` are positive integers and `m != n`:

`B.shape`	`x.shape`	`y.shape`
`(2, 2)`	`(2)`	`(2)`
`(n, n)`	`(m)`	`ERROR`
`(n, n)`	`(n)`	`(n)`
`(n, n)`	`(s, n)`	`(s, n)`
`(b, n, n)`	`(n)`	`(b, n)`
`(b, n, n)`	`(b, n)`	`(b, n)`
`(b, n, n)`	`(s, 1, n)`	`(s, b, n)`

These rules and the ability to broadcast make batch operations easy.

We can also easily apply multiple bijectors. Suppose we want to apply both these bijectors:

$$B_1 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \qquad B_2 = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}.$$

We can do this using the batched bijector

In [20]:
```
# Create a batched ScaleMatvecLinearOperator bijector
diag = tf.stack \
    ((tf.constant([1, -1.]), tf.constant([-1, 1.])))  # (2, 2)
scale = tf.linalg.LinearOperatorDiag(diag = diag)  # (2, 2, 2)
bijector = tfb.ScaleMatvecLinearOperator(scale = scale)
```

and we can broadcast the samples across both bijectors in the batch, as well as broadcasting the bijectors across all samples. For this, we need to include a batch dimension in the samples Tensor.

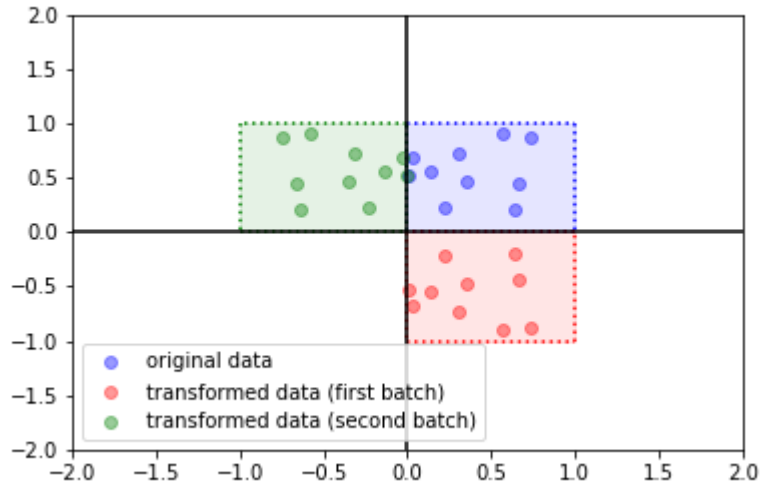
In [21]:
```
# Add a singleton batch dimension to x
x = tf.expand_dims(x, axis = 1)
x.shape
```

Out[21]:
```
TensorShape([10, 1, 2])
```

In [22]:
```
# Apply the batched bijector to x
y = bijector(x)
y.shape  # (S, B, E) shape semantics
```

Out[22]:
```
TensorShape([10, 2, 2])
```

which gives two batches of forward values for each sample:



Conclusion

In this reading, you saw how to construct scale bijectors in two dimensions using the various `ScaleMatvec` classes. You also had a quick introduction to the general `LinearOperators` class and some of its subclasses. Finally, you saw how batching makes large computations clean and efficient. Be careful to keep track of the tensor shapes, as broadcasting and the difference between batch shapes and event shapes makes errors easy. Finally, note that these bijectors are still amenable to composition (via `Chain` or simply feeding one into another) and inversion, which retains the same syntax you're used to. Enjoy using this powerful tool!

Further reading and resources

- `ScaleMatvec` bijectors:
  - https://www.tensorflow.org/probability/api_docs/python/tfp/bijectors/ScaleMatvecDiag\n (https://www.tensorflow.org/probability/api_docs/python/tfp/bijectors/ScaleMatvecDiag\n)",
  - https://www.tensorflow.org/probability/api_docs/python/tfp/bijectors/ScaleMatvecLinearOperator\n (https://www.tensorflow.org/probability/api_docs/python/tfp/bijectors/ScaleMatvecLinearOperator\n)",
  - https://www.tensorflow.org/probability/api_docs/python/tfp/bijectors/ScaleMatvecLU\n (https://www.tensorflow.org/probability/api_docs/python/tfp/bijectors/ScaleMatvecLU\n)",
  - https://www.tensorflow.org/probability/api_docs/python/tfp/bijectors/ScaleMatvecTriL\n (https://www.tensorflow.org/probability/api_docs/python/tfp/bijectors/ScaleMatvecTriL\n)",
- `LinearOperator` class (see also subclasses)
  - https://www.tensorflow.org/api_docs/python/tf/linalg/LinearOperator (https://www.tensorflow.org/api_docs/python/tf/linalg/LinearOperator)

Change of variables

This reading reviews the change of variables formula for continuous random variables. This important formula is fundamental to the theory of normalising flows.

Introduction

The change of variables formula tells us how to compute the probability density function of a random variable under a smooth invertible transformation.

In this reading notebook we will review the statement of the change of variables formula in various forms. We will then look at a simple example of a linear change of variables in two dimensions, where the probability density function of the transformed variable can easily be written by inspection and checked against the change of variables formula. In the following section we provide a sketch of the proof of the formula in one dimension. Finally, we will conclude the reading by discussing how the change of variables formula is applied to normalising flows.

Statement of the formula

Let $Z := (z_1, \ldots, z_d) \in \mathrm{R}^d$ be a d-dimensional continuous random variable, and suppose that $f: \mathrm{R}^d \to \mathrm{R}^d$ is a smooth, invertible transformation. Now consider the change of variables $X = f(Z)$, with $X = (x_1, \ldots, x_d)$, and denote the probability density functions of the random variables Z and X by p_Z and p_X respectively.

The change of variables formula states that

$$p_X(x) = p_Z(z) \cdot \left| \det J_f(z) \right|^{-1}, \qquad (1)$$

where $J_f(z)$ is the *Jacobian* of the transformation f, given by the matrix of partial derivatives

$$J_f(z) = \begin{bmatrix} \frac{\partial f_1}{\partial z_1} & \cdots & \frac{\partial f_1}{\partial z_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_d}{\partial z_1} & \cdots & \frac{\partial f_d}{\partial z_d} \end{bmatrix},$$

and $\left| \det J_f(z) \right|$ is the absolute value of the determinant of the Jacobian matrix. Note that (1) can also be written in the log-form

$$\log p_X(x) = \log p_Z(z) - \log \left| \det J_f(z) \right|. \qquad (2)$$

Furthermore, we can equivalently consider the transformation $Z = f^{-1}(X)$. Then the change of variables formulae can be written as

$$p_Z(z) = p_X(x) \cdot \left| \det J_{f^{-1}}(x) \right|^{-1}, \qquad (3)$$

$$\log p_Z(z) = \log p_X(x) - \log \left| \det J_{f^{-1}}(x) \right|. \qquad (4)$$

A simple example

We will demonstrate the change of variables formula with a simple example. Let $Z = (z_1, z_2)$ be a 2-dimensional random variable that is uniformly distributed on the unit square $[0, 1]^2 =: \Omega_Z$. We also define the transformation $f: \mathrm{R}^2 \to \mathrm{R}^2$ as

$$f(z_1, z_2) = (\lambda z_1, \mu z_2)$$

for some nonzero $\lambda, \mu \in \mathrm{R}$. The random variable $X = (x_1, x_2)$ is given by $X = f(Z)$.

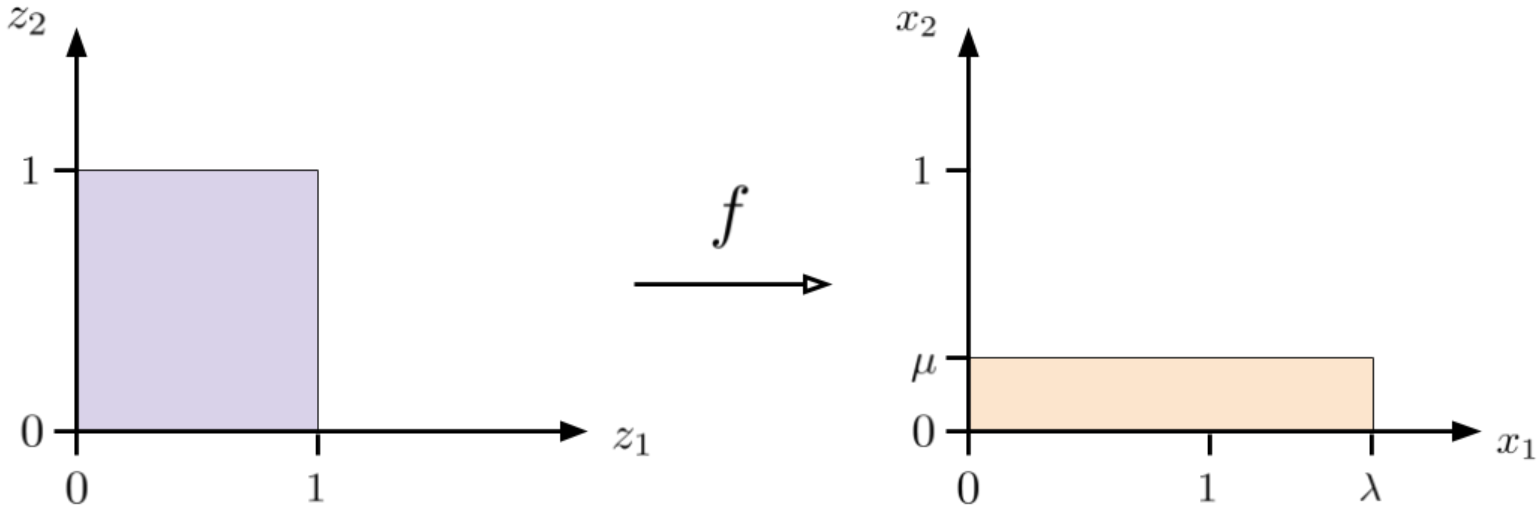


Figure 1. Linearly transformed uniformly distributed random variable.

Since $\int_{\Omega_Z} p_Z(z) dz = 1$ and Z is uniformly distributed, we have that

$$p_Z(z) = 1 \quad \text{for} \quad z \in \Omega_Z.$$

The random variable X is uniformly distributed on the region $\Omega_X = f(\Omega_Z)$ as shown in the figure above (for the case $\lambda, \mu > 0$). Since again $\int_{\Omega_X} p_X(x) dx = 1$, the probability density function for X must be given by

$$p_X(x) = \frac{1}{|\Omega_X|} = \frac{1}{|\lambda\mu|} \quad \text{for} \quad x \in \Omega_X.$$

This result corresponds to the equations (1)-(4) above. In this simple example, the transformation f is linear, and the Jacobian matrix is given by

$$J_f(z) = \begin{bmatrix} \lambda & 0 \\ 0 & \mu \end{bmatrix}.$$

The absolute value of the determinant is $\left| \det J_f(x) \right| = |\lambda\mu| \neq 0$. Equation (1) then implies

$$\begin{aligned} p_X(x) &= p_Z(z) \cdot \left| \det J_f(z) \right|^{-1} \\ &= \frac{1}{|\lambda\mu|}. \end{aligned}$$

Writing in the log-form as in equation (2) gives

$$\begin{aligned} \log p_X(x) &= \log p_Z(z) - \log \left| \det J_f(z) \right| \\ &= \log(1) - \log|\lambda\mu| \\ &= -\log|\lambda\mu|. \end{aligned}$$

Sketch of proof in 1-D

We now provide a sketch of the proof of the change of variables formula in one dimension. Let Z and X be random variables such that $X = f(Z)$, where $f: \mathrm{R} \to \mathrm{R}$ is a C^k diffeomorphism with $k \geq 1$. The change of variables formula in one dimension can be written

$$p_X(x) = p_Z(z) \cdot \left| \frac{d}{dz} f(z) \right|^{-1}, \qquad \text{(cf. equation (1))}$$

or equivalently as

$$p_X(x) = p_Z(z) \cdot \left| \frac{d}{dx} f^{-1}(x) \right|. \qquad \text{(cf. equation (3))}$$

Sketch of proof. For f to be invertible, it must be strictly monotonic. That means that for all $x^{(1)}, x^{(2)} \in \mathrm{R}$ with $x^{(1)} < x^{(2)}$, we have $f(x^{(1)}) < f(x^{(2)})$ (strictly monotonically increasing) or $f(x^{(1)}) > f(x^{(2)})$ (strictly monotonically decreasing).

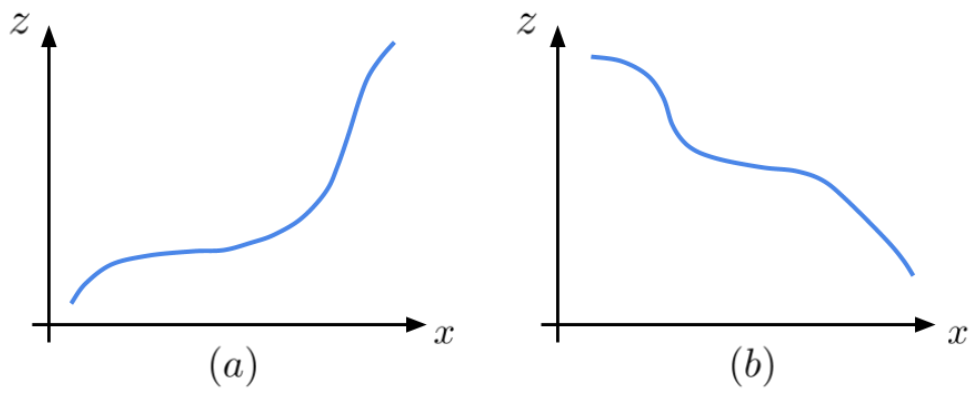


Figure 2. Sketch of monotonic functions: (a) strictly increasing, (b) strictly decreasing.

Suppose first that f is strictly increasing. Also let F_X and F_Z be the cumulative distribution functions of the random variables X and Z respectively. Then we have

$$\begin{aligned} F_X(x) &= P(X \leq x) \\ &= P(f(Z) \leq x) \\ &= P(Z \leq f^{-1}(x)) \qquad \text{(since } f \text{ is monotonically increasing)} \\ &= F_Z(f^{-1}(x)) \end{aligned}$$

By differentiating on both sides with respect to x, we obtain the probability density function:

$$\begin{aligned} p_X(x) &= \frac{d}{dx} F_X(x) \\ &= \frac{d}{dx} F_Z(f^{-1}(x)) \\ &= \frac{d}{dz} F_Z(z) \cdot \frac{d}{dx} f^{-1}(x) \\ &= p_Z(z) \frac{d}{dx} f^{-1}(x) \qquad (5) \end{aligned}$$

Now suppose first that f is strictly decreasing. Then

$$\begin{aligned} F_X(x) &= P(X \leq x) \\ &= P(f(Z) \leq x) \\ &= P(Z \geq f^{-1}(x)) \qquad \text{(since } f \text{ is monotonically decreasing)} \\ &= 1 - F_Z(f^{-1}(x)) \end{aligned}$$

Again differentiating on both sides with respect to x:

$$\begin{aligned} p_X(x) &= \frac{d}{dx} F_X(x) \\ &= -\frac{d}{dx} F_Z(f^{-1}(x)) \\ &= -F'_Z(f^{-1}(x)) \frac{d}{dx} f^{-1}(x) \\ &= -p_Z(z) \frac{d}{dx} f^{-1}(x) \qquad (6) \end{aligned}$$

Now note that the inverse of a strictly monotonically increasing (resp. decreasing) function is again strictly monotonically increasing (resp. decreasing). This implies that the quantity $\frac{d}{dx} f^{-1}(x)$ is positive in (5) and negative in (6), and so these two equations can be combined into the single equation:

$$p_X(x) = p_Z(z) \left| \frac{d}{dx} f^{-1}(x) \right|$$

which completes the proof.

Normalising flows

Normalising flows are a class of models that exploit the change of variables formula to estimate an unknown target data density.

Suppose we have data samples $\mathrm{D} := \{x^{(1)}, \ldots, x^{(n)}\}$, with each $x^{(i)} \in \mathrm{R}^d$, and assume that these samples are generated i.i.d. from the underlying distribution p_X.

A normalising flow models the distribution p_X using a random variable Z (also of dimension d) with a simple distribution p_Z (e.g. an isotropic Gaussian), such that the random variable X can be written as a change of variables $X = f_\theta(Z)$, where θ is a parameter vector that parameterises the smooth invertible function f_θ.

The function f_θ is modelled using a neural network with parameters θ, which we want to learn from the data. An important point is that this neural network must be designed to be invertible, which is not the case in general with deep learning models. In practice, we often construct the neural network by composing multiple simpler blocks together. In TensorFlow Probability, these simpler blocks are the *bijectors* that we will study in the first part of the week.

In order to learn the optimal parameters θ, we apply the principle of maximum likelihood and search for θ_{ML} such that

$$\begin{aligned}\theta_{ML} &:= \arg\max_\theta P(\mathrm{D};\theta) \\ &= \arg\max_\theta \log P(\mathrm{D};\theta).\end{aligned}$$

In order to compute $\log P(\mathrm{D};\theta)$ we can use the change of variables formula:

$$P(\mathrm{D};\theta) = \prod_{x\in\mathrm{D}} p_Z(f_\theta^{-1}(x)) \cdot \left|\det J_{f_\theta^{-1}}(x)\right|$$

$$\log P(\mathrm{D};\theta) = \sum_{x\in\mathrm{D}} \log p_Z(f_\theta^{-1}(x)) + \log\left|\det J_{f_\theta^{-1}}(x)\right| \qquad (7)$$

The term $p_Z(f_\theta^{-1}(x))$ can be computed for a given data point $x \in \mathrm{D}$ since the neural network f_θ is designed to be invertible, and the distribution p_Z is known. The term $\det J_{f_\theta^{-1}}(x)$ is also computable, although this also highlights another important aspect of normalising flow models: they should be designed such that the determinant of the Jacobian can be efficiently computed.

The log-likelihood (7) is usually optimised as usual in minibatches, with gradient-based optimisation methods.

Further reading and resources

Some general resources related to the content of this reading are:

- https://en.wikipedia.org/wiki/Probability_density_function (https://en.wikipedia.org/wiki/Probability_density_function)
- https://en.wikipedia.org/wiki/Cumulative_distribution_function (https://en.wikipedia.org/wiki/Cumulative_distribution_function)
- https://en.wikipedia.org/wiki/Monotonic_function (https://en.wikipedia.org/wiki/Monotonic_function)

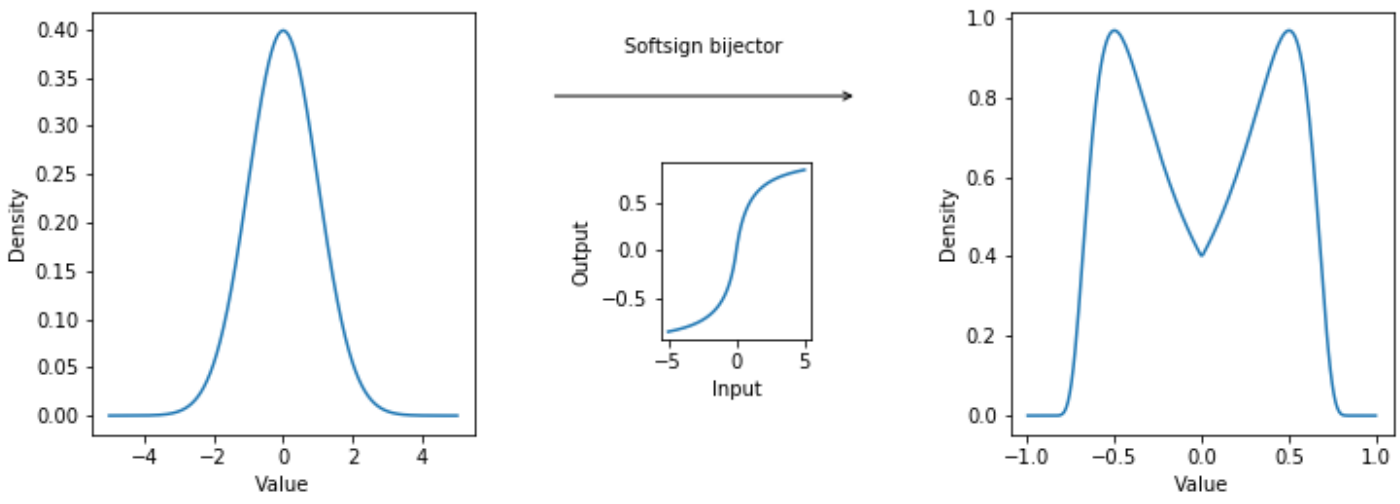
Autoregressive flows and RealNVP

This reading contains an overview of normalising flows, and introduces two popular normalising flow models: masked autoregressive flow (MAF) and RealNVP.

You'll also learn about the considerations in different architectures and the tradeoff between computational complexity and learning power.

Introduction

Before any theory, we'll discuss an example of how normalizing flows work. Suppose you have a standard normal distribution (mean 0, variance 1). It has a single mode at 0, so, even after scaling and shifting, can't be fit well to data with two modes. However, you've seen how bijectors applied to distributions can create other distributions. A natural question is then: can we create a bimodal distribution (one with two modes) from a bijector applied to a standard normal distribution? It turns out that this is possible with the `Softsign` bijector. This is a differentiable approximation to the sign function (1 if x is nonnegative, -1 if x is negative). Passing a standard normal distribution through this bijector transforms the probability distribution as follows:



As you can see, the bijector created a bimodal distribution from a standard normal one! This is just one from a huge class of possible bijectors available in TensorFlow Probability. Furthermore, since you can chain them together, it's possible to create very complicated bijectors that change standard distributions (e.g. a normal) to very complicated ones. This is how a normalizing flow works: it creates a complicated distribution by applying a bijector to a simple, well-understood and computationally implemented distribution (such as a Gaussian). In this reading, you'll learn how this works and see some implementations from previous research.

Normalizing flows

The one-dimensional case

The main idea of normalizing flows is to create a random variable X (with complicated distribution P) by applying a bijector f to a random variable Z (with a simple distribution). For example, suppose $Z \sim N(0, 1)$ has a standard normal distribution. The goal is to find a bijector f so that $X = f(Z) \sim P$ for some target distribution P. Under this transformation, we can calculate the log-density using the change of variables equation:

$$\log p(x) = \log p(z) - \log \left| \frac{\partial f}{\partial z}(z) \right|$$

where $z = f^{-1}(x)$.

Finding an f that changes the distribution as required is not trivial: if the target distribution P is very complex, a simple f (such as a scale or shift) won't do the trick. However, we know that composing bijectors with one another creates more bijectors. Hence, one approach is to combine multiple simple bijectors f_k to create a more complicated f:

$$f = f_K \circ f_{K-1} \circ \ldots \circ f_1.$$

This series, where a base distribution is transformed by a series of bijectors after each other, is called a *normalizing flow*:

$$z_0 = z$$

$$z_k = f_k(z_{k-1}) \quad k = 1, \ldots, K.$$

$$x = z_K$$

Furthermore, the log-probability can be calculated by summing the contributions from each of the bijectors:

$$\log p(x) = \log p(z) - \sum_{k=1}^{K} \log \left| \frac{\partial f_k}{\partial z_{k-1}}(z_{k-1}) \right|$$

This, however, still doesn't answer the question of how to construct the f_k. Usually, this is done by giving each f_k some simple functional form, such as a scale and shift followed by a simple nonlinearity such as a sigmoid or ReLU. Each f_k will have some parameters (such as the scale and shift values), and these can be learned via standard methods such as maximum likelihood estimation given some training data.

The higher-dimensional case

The results above generalise straightforwardly to higher dimensions. Suppose that $\mathbf{z} \sim N(0, \mathbf{I})$ is distributed according to a multivariate unit Gaussian. The normalizing flow is then

$$\mathbf{z}_0 = \mathbf{z}$$

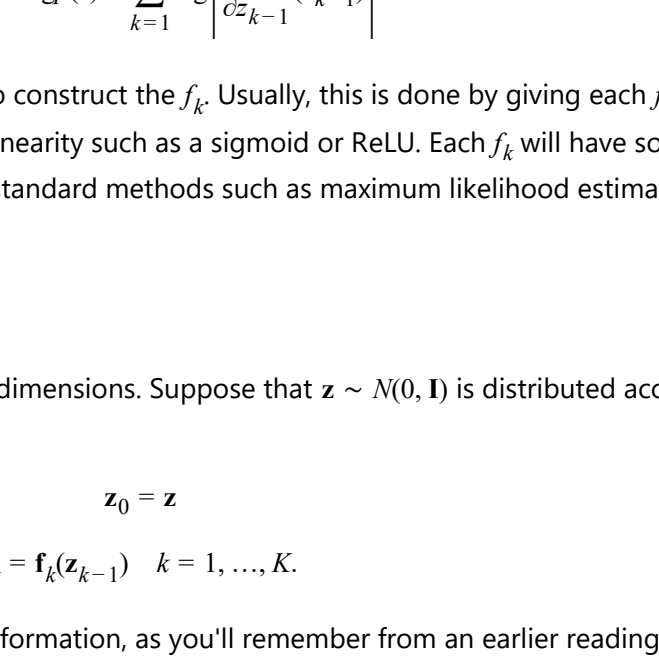
$$\mathbf{z}_k = \mathbf{f}_k(\mathbf{z}_{k-1}) \quad k = 1, \ldots, K.$$

The log-probability involves the determinant of the transformation, as you'll remember from an earlier reading:

$$\log p(\mathbf{x}) = \log p(\mathbf{z}) - \sum_{k=1}^{K} \log \left(\left| \det \left(\frac{\partial \mathbf{f}_k}{\partial \mathbf{z}_{k-1}}(\mathbf{z}_{k-1}) \right) \right| \right)$$

where we use the shorthand notation $\frac{\partial \mathbf{a}}{\partial \mathbf{b}}$ for the matrix with components $\frac{\partial \mathbf{a}_i}{\partial \mathbf{b}_j}$, where i and j index the components of $\mathbf{a}$ and $\mathbf{b}$ respectively.

Let's see an example of this from an early research paper (https://arxiv.org/abs/1505.05770). In the figure below, the left column is the density of the target distribution P, and the right columns are the normalizing flow approximations with K=2, 8 and 32 bijectors (each with a simple form and some trainable parameters).



As you can see, the approximation improves as the number of bijectors in the flow increases.

The reason this is useful is that it allows us to "learn" a complex distribution from data and then manipulate it. For example, to draw a new sample from the learned distribution, simply draw $\mathbf{z}$ from a standard unit Gaussian and transform it to the correct space using the normalizing flow (the series of bijectors).

Note: Throughout this reading, we use the index k to refer to the bijector in the normalizing flow and indices i and j to refer to dimensions of the probability space (from 1 to D). From here on, for clarity, we consider a normalizing flow formed of only one bijector (with $K = 1$), so that we may drop the indices k. The equation becomes $\mathbf{x} = \mathbf{f}(\mathbf{z})$. The reason for doing this is that we now use indices to refer to components of the vectors (e.g. $\mathbf{x} = [x_1, \ldots, x_D]^T$) where D is the dimensionality. For normalizing flows with $K > 1$, the results apply for each k.

Computational concerns

The above theory provides, in principle, a framework to learn and manipulate complex distributions by building them up from a simple one. There is one key difficulty, however, when going to a practical implementation. This comes from the need to calculate the determinant $\left| \det \left(\frac{\partial \mathbf{f}}{\partial \mathbf{z}} \right) \right|$ to determine the density of the transformed variable $\mathbf{x}$. The computational cost (number of operations) to calculate a determinant for a general matrix with D dimensions scales as $\mathrm{O}(D^3)$. This makes general normalizing flow density calculations intractable, and some simplifications, as outlined below, are required.

Autoregressive flows

For some matrices, calculating a determinant is easy. For example, for a lower or upper triangular matrix, the determinant is the product of the diagonal elements, of which there are D, meaning the determinant calculation scales linearly. Hence, to attain a linear scaling of the determinant in the number of dimensions, it is enough to enforce that $\frac{\partial f_i}{\partial z_j} = 0$ whenever $j > i$. In other words, the component f_i depends only on $z_1, \ldots z_i$.

Autoregressive models can be reinterpreted as normalising flows that fulfil this requirement. These are models that model the joint density $p(\mathbf{x})$ as the product of conditionals $\prod_i p(x_i \mid \mathbf{x}_{1:i-1})$. For example, the conditionals could be parameterised as Gaussians:

$$\begin{aligned} p(x_i \mid \mathbf{x}_{1:i-1}) &= \mathrm{N}(x_i \mid \mu_i, \exp(\sigma_i)^2), \\ \text{where} \qquad \mu_i &= f_{\mu_i}(\mathbf{x}_{1:i-1}) \\ \text{and} \qquad \sigma_i &= f_{\sigma_i}(\mathbf{x}_{1:i-1}). \end{aligned}$$

In the above equations, the mean and standard deviations of each conditional distribution are computed using (parameterised) functions of all previous variables. The above can alternatively be written as:

$$x_i = \mu_i(\mathbf{x}_{1:i-1}) + \exp(\sigma_i(\mathbf{x}_{1:i-1}))z_i \qquad i = 1, \ldots, D$$

where $z_i \sim N(0, 1)$ is sampled from a unit Gaussian. This last equation shows how the autoregressive model can be viewed as a transformation f from the random variables $\mathbf{z} \in \mathrm{R}^D$ to the data $\mathbf{x} \in \mathrm{R}^D$.

This is an example of an *autoregressive* process where x_i depends only on the components of $\mathbf{z}$ that are lower than or equal to i but not any of the higher ones. The dependence on lower dimensions of $\mathbf{z}$ happens indirectly through the x_i dependence in the f_{μ_i} and f_{σ_i}.

Implementations

Masked Autoregressive Flow (MAF)

An implementation of the above autoregressive flow appears in the following paper:

- George Papamakarios, Theo Pavlakou, Iain Murray (2017). Masked Autoregressive Flow for Density Estimation (http://papers.nips.cc/paper/6828-masked-autoregressive-flow-for-density-estimation.pdf). In *Advances in Neural Information Processing Systems*, 2017.

Here, the authors use the above equations, using a masked autoencoder for distribution estimation (MADE (http://proceedings.mlr.press/v37/germain15.pdf)) to implement the functions f_{μ_i} and f_{σ_i}. For clarity, let's see how $\mathbf{x}$ is sampled. This is done as follows:

1. $x_1 = f_{\mu_1} + \exp(f_{\sigma_1})z_1$ for $z_1 \sim N(0, 1)$
2. $x_2 = f_{\mu_2}(x_1) + \exp(f_{\sigma_2}(x_1))z_2$ for $z_2 \sim N(0, 1)$
3. $x_3 = f_{\mu_3}(x_1, x_2) + \exp(f_{\sigma_3}(x_1, x_2))z_3$ for $z_3 \sim N(0, 1)$

and so on. For the f_{μ_i} and f_{σ_i}, they use the same MADE network across the i, but mask the weights so that x_i depends on x_j for all $j < i$ but not any others. By re-using the same network, weights can be shared and the total number of parameters is significantly lower.

A note on computational complexity: determining $\mathbf{x}$ from $\mathbf{z}$ is relatively slow, since this must be done sequentially: first x_1, then x_2, and so on up to x_D. However, determining $\mathbf{z}$ from $\mathbf{x}$ is fast: each of the above equations can be solved for z_i at the same time:

$$z_i = \frac{x_i - f_{\mu_i}}{\exp(f_{\sigma_i})} \qquad i = 0, \ldots, D - 1$$

Hence, the *forward* pass through the bijector (sampling $\mathbf{x}$) is relatively slow, but the *inverse* pass (determining $\mathbf{z}$), which is used in the likelihood calculations used to train the model, is fast.

Inverse Autoregressive Flow (IAF)

The inverse autoregressive flow reverses the dependencies to make the forward pass parallelisable but the inverse pass sequential. Details can be found in the following paper:

- Diederik Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, Max Welling (2016). Improved Variational Inference with Inverse Autoregressive Flow (http://papers.nips.cc/paper/6581-improved-variational-inference-with-inverse-autoregressive-flow.pdf). In *Advances in Neural Information Processing Systems*, 2016.

It uses the same equations:

$$x_i = \mu_i + \exp(\sigma_i) z_i \qquad i = 1, \ldots, D$$

but has the scale and shift functions depend on the z_i instead of the x_i:

$$\mu_i = f_{\mu_i}(z_1, \ldots, z_{i-1}) \qquad \sigma_i = f_{\sigma_i}(z_1, \ldots, z_{i-1}).$$

Note that now the forward equation (determining $\mathbf{x}$ from $\mathbf{z}$) can be parallelised, but the reverse transformations require determining z_1, followed by z_2, etc. and must hence be solved in sequence.

Real-NVP and NICE

A further simplification of these approaches can be found in these papers:

- Laurent Dinh, Jascha Sohl-Dickstein, Samy Bengio (2016). Density estimation using Real NVP (https://arxiv.org/abs/1605.08803).
- Laurent Dinh, David Krueger, Yoshua Bengio (2014). NICE: Non-linear Independent Components Estimation (https://arxiv.org/abs/1410.8516).

The first uses a reduced versions of the above equations, for some chosen L:

$$x_i = z_i \qquad i = 1, \ldots, d$$
$$x_i = \mu_i + \exp(\sigma_i) z_i \qquad i = d+1, \ldots D$$

where

$$\mu_i = f_{\mu_i}(z_1, \ldots, z_d)$$
$$\sigma_i = f_{\sigma_i}(z_1, \ldots, z_d)$$

Hence, nothing happens for the first d dimensions, but the z_i values across these values transform the x_i values for remaining $D - d$. Note that, in this case, both the forward and backward pass of the flow can be done fully in parallel. The second paper is even simpler, and omits the scale term altogether.

There is, of course, a catch: such a simple form means the flow typically needs a lot of bijections (a high K value) to be able to describe complicated distributions. Furthermore, the dimensions that are transformed ($D - L$ in total) and not transformed (L in total) must be permuted in the different bijections: otherwise the first L dimensions of $\mathbf{z}$ are never changed throughout the whole normalizing flow, which greatly limits the expressive power. You'll be creating such a normalizing flow in this week's programming assignment.

Further reading and resources

Besides the papers cited above, there are two great blog posts that explain that material as well:

- Normalizing Flows (http://akosiorek.github.io/ml/2018/04/03/norm_flows.html) by Adam Kosiorek
- Normalizing Flows Tutorial (https://blog.evjang.com/2018/01/nf1.html) by Eric Jang.

Both of these offer slightly more detail than we have space for here. They also have some great visuals. Happy reading and have fun implementing these ideas in the next few lessons!

Bijectors And Normalising Flows

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions
tfb = tfp.bijectors
tfpl = tfp.layers
```

1. Bijectors

2. The TransformedDistribution class

3. Subclassing bijectors

4. Normalising flows

Bijectors

In [2]:
```
# Define base distribution
normal = tfd.Normal(loc = 0., scale = 1.)
```

In [3]:
```
# Sample from base distribution
n = 10000
z = normal.sample(n)
```

Scale and shift bijector

In [4]:
```
# Define scale and shift
scale = 4.5
shift = 7
```

In [5]:
```
# Define chain bijector
scale_and_shift = tfb.Chain([tfb.Shift(shift), tfb.Scale(scale)])
```

In [6]:
```
# We can also use call methods
scale_transf = tfb.Scale(scale)
shift_transf = tfb.Shift(shift)
scale_and_shift = shift_transf(scale_transf)
```

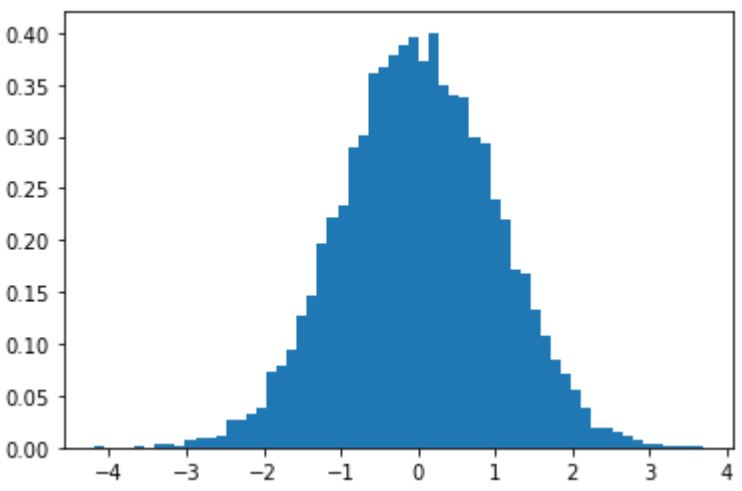
In [7]:
```
# Apply the forward transformation
x = scale_and_shift.forward(z)
```

In [8]:
```
# Check the forward transformation
tf.norm(x - (scale * z + shift))
```

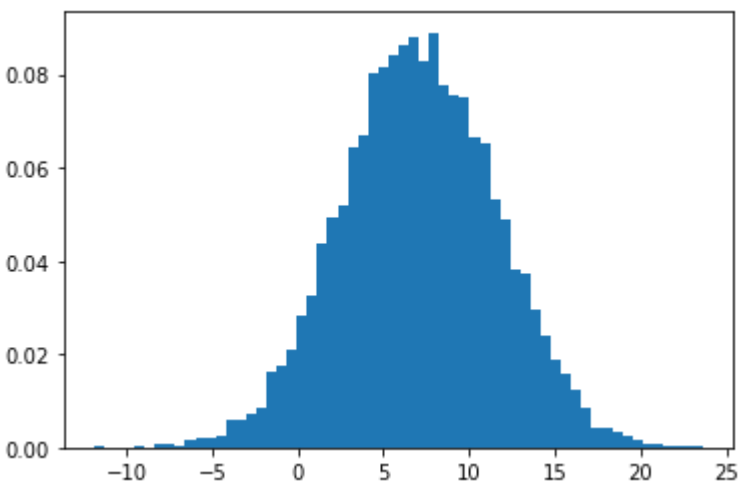
Out[8]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=0.0>
```

Plots

In [9]:
```
# Plot z density
plt.hist(z.numpy(), bins = 60, density = True)
plt.show()
```



In [10]:
```
# Plot x density
plt.hist(x.numpy(), bins = 60, density = True)
plt.show()
```



Inverse transformation

In [11]:
```
# Apply inverse transformation
inv_x = scale_and_shift.inverse(x)
```

In [12]:
```
# Check inverse transformation
tf.norm(inv_x - z)
```

Out[12]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=0.0>
```

Log probability

In [13]:
```
# Compute log prob for x
log_prob_x = normal.log_prob(z) - scale_and_shift.forward_log_det_jacobian(z, event_ndims = 0)
log_prob_x
```

Out[13]:
```
<tf.Tensor: shape=(10000,), dtype=float32, numpy=
array([-3.2562919, -5.2344127, -2.4325273, ..., -2.5546515, -2.4498444,
       -2.755146 ], dtype=float32)>
```

In [14]:
```
# We can also use the inverse transformation
log_prob_x = normal.log_prob(scale_and_shift.inverse(x)) + \
    scale_and_shift.inverse_log_det_jacobian(x, event_ndims = 0)
log_prob_x
```

Out[14]:
```
<tf.Tensor: shape=(10000,), dtype=float32, numpy=
array([-3.2562919, -5.2344127, -2.4325273, ..., -2.5546515, -2.4498444,
       -2.755146 ], dtype=float32)>
```

Broadcasting

In [15]:
```
x = tf.random.normal(shape = (100, 1))
```

In [16]:
```
# Softfloor bijector
softfloor = tfb.Softfloor(temperature = 0.01)
y = softfloor.forward(x)
y.shape
```

Out[16]:
```
TensorShape([100, 1])
```

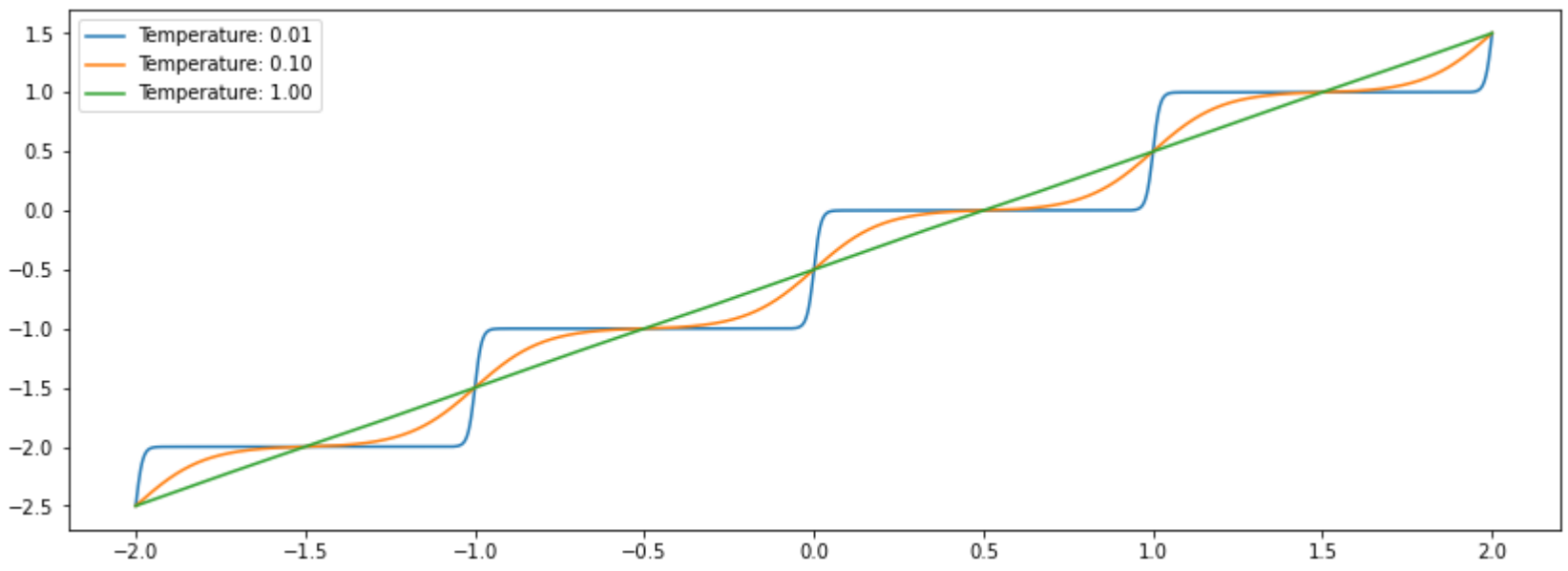
In [17]:
```
# Softfloor bijector using broadcasting
softfloor = tfb.Softfloor(temperature = [0.2, 1.])
y = softfloor.forward(x)
y.shape
```

Out[17]:
```
TensorShape([100, 2])
```

In [18]:
```
# Softfloor bijector using broadcasting
softfloor = tfb.Softfloor(temperature = [0.01, 0.1, 1.])
```

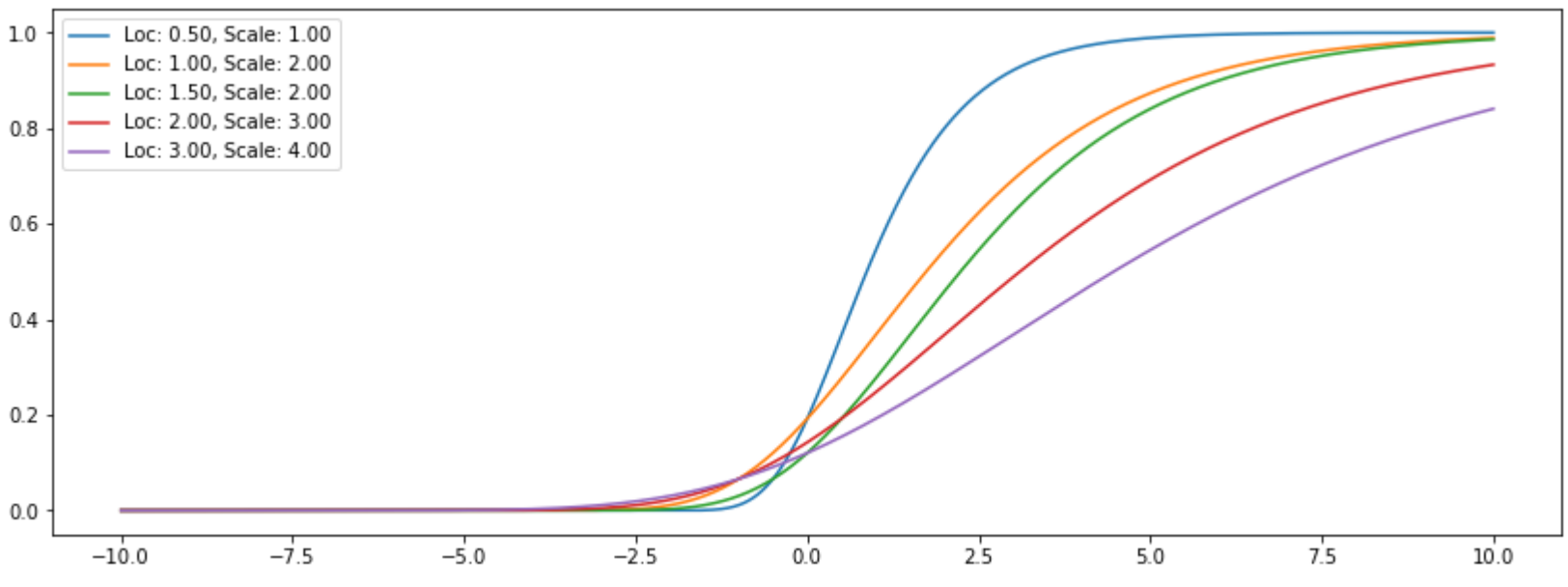
In [19]:
```
# Plot routine
def _plot(nparams, bijector, params, x):
    bijector_params = tuple(getattr(bijector, name) for name in params)
    upper_params = [name[0].upper() + name[1:] for name in params]
    fig = plt.figure(figsize=(14, 5))
    lines = plt.plot(np.tile(x, nparams), bijector.forward(x))
    for l in zip(lines, *bijector_params):
        labels = ": {:.2f}, ".join(upper_params) + ': {:.2f}'
        l[0].set_label(labels.format(*l[1:]))
    plt.legend()
    plt.show()
```

In [20]:
```
# Plot
x = np.linspace(-2, 2, 2000)[..., np.newaxis]
_plot(3, softfloor, ['temperature'], x)
```



In [21]:
```
# Gumbel bijector using broadcasting
exps = tfb.GumbelCDF(loc = [0.5, 1, 1.5, 2, 3], scale = [1, 2, 2, 3, 4])
```

In [22]:
```
# Plot
x = np.linspace(-10, 10, 2000, dtype = np.float32)[..., np.newaxis]
_plot(5, exps, ['loc', 'scale'], x)
```



The TransformedDistribution class

TransformedDistribution

In [23]:
```
# Parameters
n = 10000
loc = 0
scale = 0.5
```

In [24]:
```
# Normal distribution
normal = tfd.Normal(loc = loc, scale = scale)
```

In [25]:
```
# Display event and batch shape
print('batch shape: ', normal.batch_shape)
print('event shape: ', normal.event_shape)
```

```
batch shape:  ()
event shape:  ()
```

In [26]:
```
# Exponential bijector
exp = tfb.Exp()
```

In [27]:
```
# Log normal transformed distribution using exp and normal bijectors
log_normal_td = exp(normal)
```

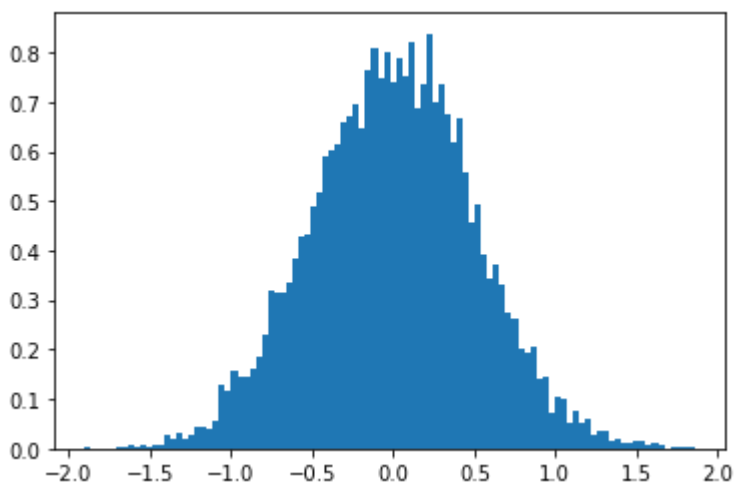
In [28]:
```
# Display event and batch shape
print('batch shape: ', log_normal_td.batch_shape)
print('event shape: ', log_normal_td.event_shape)
```

```
batch shape:  ()
event shape:  ()
```

In [29]:
```
# Base distribution
z = normal.sample(n)
```

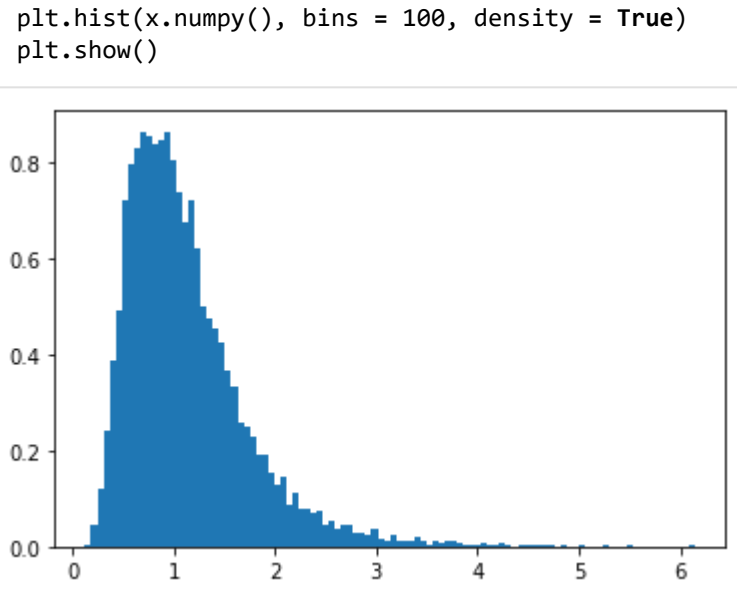
Plots

In [30]:
```
# Plot z density
plt.hist(z.numpy(), bins = 100, density = True)
plt.show()
```



In [31]:
```
# Transformed distribution
x = log_normal_td.sample(n)
```

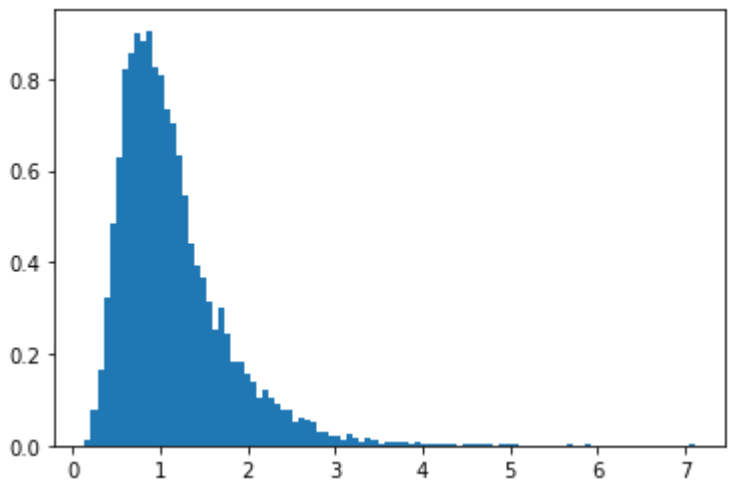
In [32]:
```
# Plot x density
plt.hist(x.numpy(), bins = 100, density = True)
plt.show()
```



In [33]:
```
# Define log normal distribution
log_normal = tfd.LogNormal(loc = loc, scale = scale)
```

In [34]:
```
# Sample log_normal
l = log_normal.sample(n)
```

In [35]:
```
# Plot l density
plt.hist(l.numpy(), bins = 100, density = True)
plt.show()
```



Log probability

In [36]:
```
# Log prob of LogNormal
log_prob = log_normal.log_prob(x)
```

In [37]:
```
# Log prob of log normal transformed distribution
log_prob_td = log_normal_td.log_prob(x)
```

In [38]:
```
# Check log probs
tf.norm(log_prob - log_prob_td)
```

Out[38]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=1.01100395e-05>
```

Event shape and batch shape

In [39]:
```
# Set a scaling lower triangular matrix
tril = tf.random.normal((2, 4, 4))
scale_low_tri = tf.linalg.LinearOperatorLowerTriangular(tril)
```

In [40]:
```
# View of scale_low_tri
scale_low_tri.to_dense()
```

Out[40]:
```
<tf.Tensor: shape=(2, 4, 4), dtype=float32, numpy=
array([[[-0.44597602,  0.        ,  0.        ,  0.        ],
        [ 0.05159349, -0.09687567,  0.        ,  0.        ],
        [-0.46078765,  0.0207627 , -0.27233383,  0.        ],
        [-1.1664037 , -1.2637694 ,  0.5337775 ,  1.393418  ]],

       [[ 0.6948613 ,  0.        ,  0.        ,  0.        ],
        [ 1.1749753 ,  0.4086611 ,  0.        ,  0.        ],
        [-0.08414282,  0.4017596 , -0.17369917,  0.        ],
        [-0.4325606 , -0.18971127, -0.43916336, -1.9362649 ]]],
      dtype=float32)>
```

In [41]:
```
# Define scale linear operator
scale_lin_op = tfb.ScaleMatvecLinearOperator(scale_low_tri)
```

In [42]:
```
# Define scale linear operator transformed distribution with a batch and event shape
mvn = tfd.TransformedDistribution(normal, scale_lin_op, batch_shape = [2], event_shape = [4])
mvn
```

Out[42]:
```
<tfp.distributions.TransformedDistribution 'scale_matvec_linear_operatorNormal' batch_shape=[2] event_shape=[4] dtype=
float32>
```

In [43]:
```
# Display event and batch shape
print('batch shape: ', mvn.batch_shape)
print('event shape: ', mvn.event_shape)
```

```
batch shape:  (2,)
event shape:  (4,)
```

In [44]:
```
# Sample
y1 = mvn.sample(sample_shape = (n,))
y1.shape
```

Out[44]:
```
TensorShape([10000, 2, 4])
```

In [45]:
```
# Define a MultivariateNormalLinearOperator distribution
mvn2 = tfd.MultivariateNormalLinearOperator(loc = 0, scale = scale_low_tri)
mvn2
```

Out[45]:
```
<tfp.distributions.MultivariateNormalLinearOperator 'MultivariateNormalLinearOperator' batch_shape=[2] event_shape=[4]
dtype=float32>
```

In [46]:
```
# Display event and batch shape
print('batch shape: ', mvn2.batch_shape)
print('event shape: ', mvn2.event_shape)
```

```
batch shape:  (2,)
event shape:  (4,)
```

In [47]:
```
# Sample
y2 = mvn2.sample(sample_shape = (n,))
y2.shape
```

Out[47]:
```
TensorShape([10000, 2, 4])
```

In [48]:
```
# Check
xn = normal.sample((n, 2, 4))
tf.norm(mvn.log_prob(xn) - mvn2.log_prob(xn)) / tf.norm(mvn.log_prob(xn))
```

Out[48]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=0.74677587>
```

Subclassing bijectors

In [49]:
```
# Define a new bijector: Cubic
class Cubic(tfb.Bijector):

    def __init__(self, a, b, validate_args = False, name = 'Cubic'):
        self.a = tf.cast(a, tf.float32)
        self.b = tf.cast(b, tf.float32)
        if validate_args:
            assert tf.reduce_mean(tf.cast(tf.math.greater_equal(tf.abs(self.a), 1e-5), tf.float32)) == 1.0
            assert tf.reduce_mean(tf.cast(tf.math.greater_equal(tf.abs(self.b), 1e-5), tf.float32)) == 1.0
        super(Cubic, self).__init__ \
            (validate_args = validate_args, forward_min_event_ndims = 0, name = name)

    def _forward(self, x):
        x = tf.cast(x, tf.float32)
        return tf.squeeze(tf.pow(self.a * x + self.b, 3))

    def _inverse(self, y):
        y = tf.cast(y, tf.float32)
        return (tf.math.sign(y) * tf.pow(tf.abs(y), 1 / 3) - self.b) / self.a

    def _forward_log_det_jacobian(self, x):
        x = tf.cast(x, tf.float32)
        return tf.math.log(3. * tf.abs(self.a)) + 2. * tf.math.log(tf.abs(self.a * x + self.b))
```

In [50]:
```
# Cubic bijector
cubic = Cubic([1., -2.], [-1., 0.4], validate_args = True)
```

In [51]:
```
# Apply forward transformation
x = tf.constant([[1, 2], [3, 4]])
y = cubic.forward(x)
y
```

Out[51]:
```
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[   0.     ,  -46.656  ],
       [   8.     , -438.97598]], dtype=float32)>
```

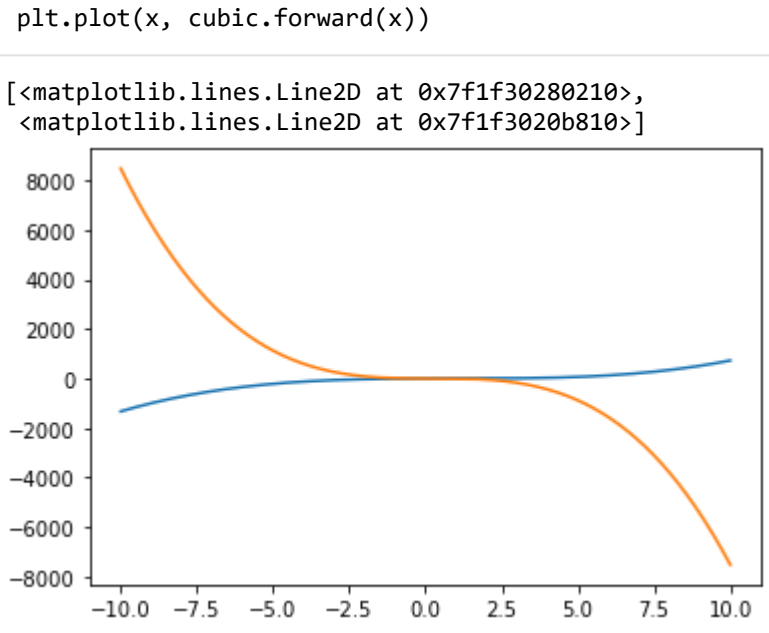
In [52]:
```
# Check inverse
np.linalg.norm(x - cubic.inverse(y))
```

Out[52]:
```
0.0
```

Function plots

In [53]:
```
# Plot the forward transformation
x = np.linspace(-10, 10, 500).reshape(-1, 1)
plt.plot(x, cubic.forward(x))
```

Out[53]:
```
[<matplotlib.lines.Line2D at 0x7f1f30280210>,
 <matplotlib.lines.Line2D at 0x7f1f3020b810>]
```

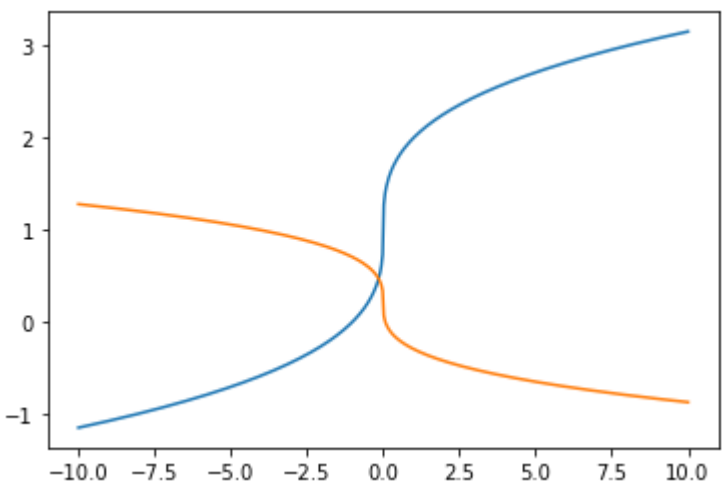


In [54]:
```
# Display shape
cubic.forward(x).shape
```

Out[54]:
```
TensorShape([500, 2])
```

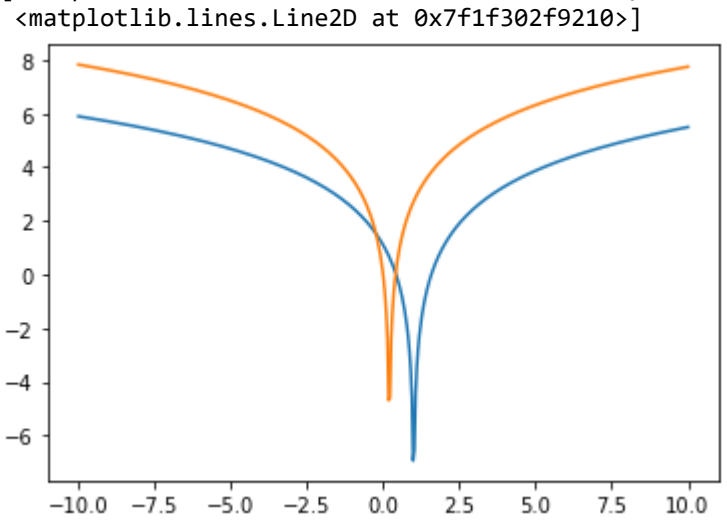
In [55]:
```
# Plot the inverse
plt.plot(x, cubic.inverse(x))
```

Out[55]:
```
[<matplotlib.lines.Line2D at 0x7f1f302e1b50>,
 <matplotlib.lines.Line2D at 0x7f1f3021c510>]
```



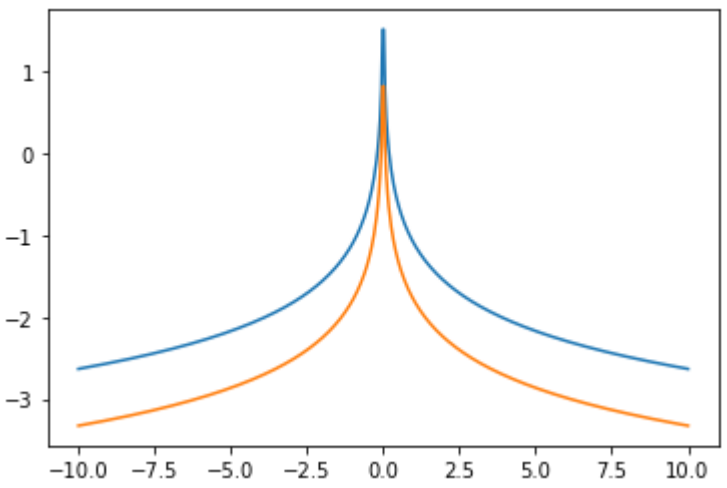
In [56]:
```
# Plot the forward log Jacobian determinant
plt.plot(x, cubic.forward_log_det_jacobian(x, event_ndims = 0))
```

Out[56]:
```
[<matplotlib.lines.Line2D at 0x7f1f303d7710>,
 <matplotlib.lines.Line2D at 0x7f1f302f9210>]
```



In [57]:
```
# Plot the inverse log Jacobian determinant
plt.plot(x, cubic.inverse_log_det_jacobian(x, event_ndims = 0))
```

Out[57]:
```
[<matplotlib.lines.Line2D at 0x7f1f30576dd0>,
 <matplotlib.lines.Line2D at 0x7f1f305cff50>]
```



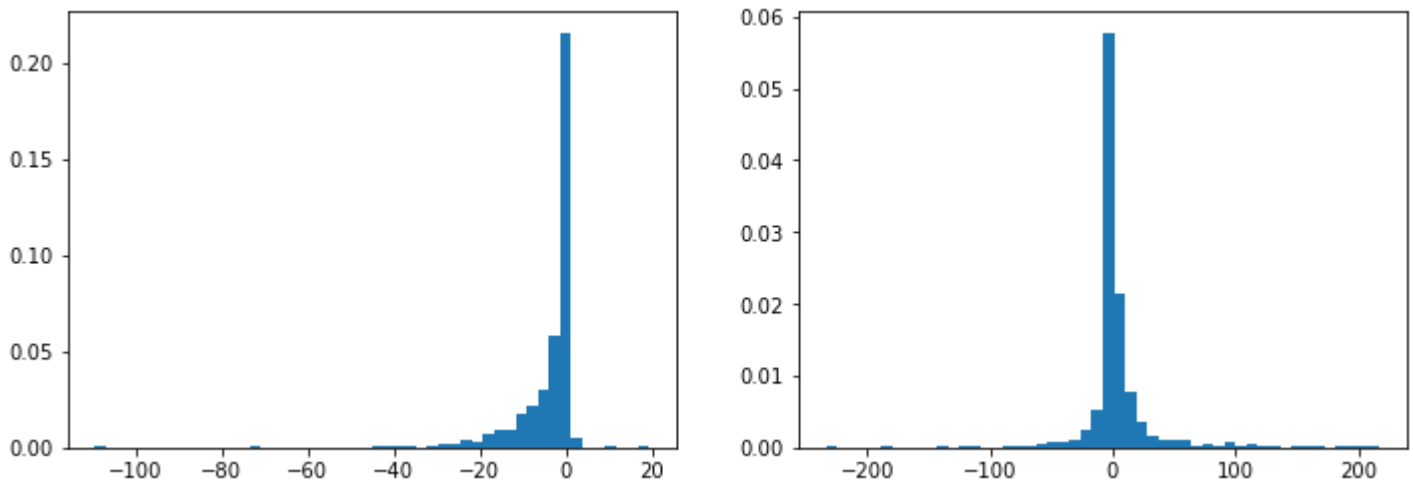
TransformedDistribution and plots

In [58]:
```
# Create a transformed distribution with Cubic
normal = tfd.Normal(loc = 0., scale = 1.)
cubed_normal = tfd.TransformedDistribution(normal, cubic, event_shape = [2])
```

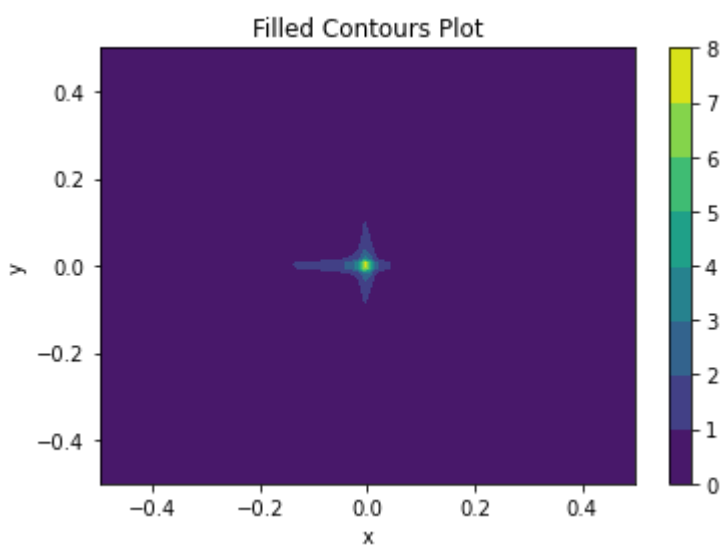
In [59]:
```
# Sample cubed_normal
n = 1000
g = cubed_normal.sample(n)
g.shape
```

Out[59]:
```
TensorShape([1000, 2])
```

In [60]:
```
# Plot histograms
plt.figure(figsize = (12, 4))
plt.subplot(1, 2, 1)
plt.hist(g[..., 0].numpy(), bins = 50, density = True)
plt.subplot(1, 2, 2)
plt.hist(g[..., 1].numpy(), bins = 50, density = True)
plt.show()
```



In [61]:
```
# Make contour plot
xx = np.linspace(-0.5, 0.5, 100)
yy = np.linspace(-0.5, 0.5, 100)
X, Y = np.meshgrid(xx, yy)

fig, ax = plt.subplots(1, 1)
Z = cubed_normal.prob(np.dstack((X, Y)))
cp = ax.contourf(X, Y, Z)
fig.colorbar(cp) # Add a colorbar to a plot
ax.set_title('Filled Contours Plot')
ax.set_xlabel('x')
ax.set_ylabel('y')
plt.show()
```

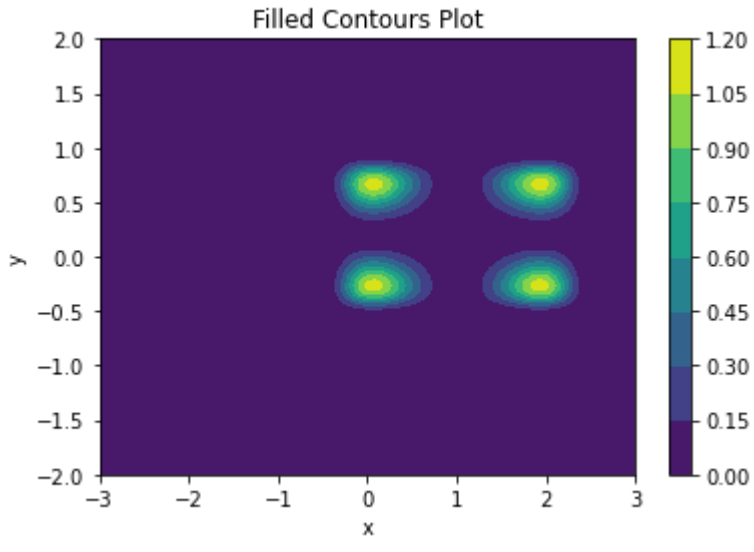


In [62]:
```
# Create a transformed distribution with the inverse of Cube
inverse_cubic = tfb.Invert(cubic)
inv_cubed_normal = inverse_cubic(normal, event_shape = [2])
```

In [63]:
```
# Samble inv_cubed_normal
n = 1000
g = inv_cubed_normal.sample(n)
g.shape
```

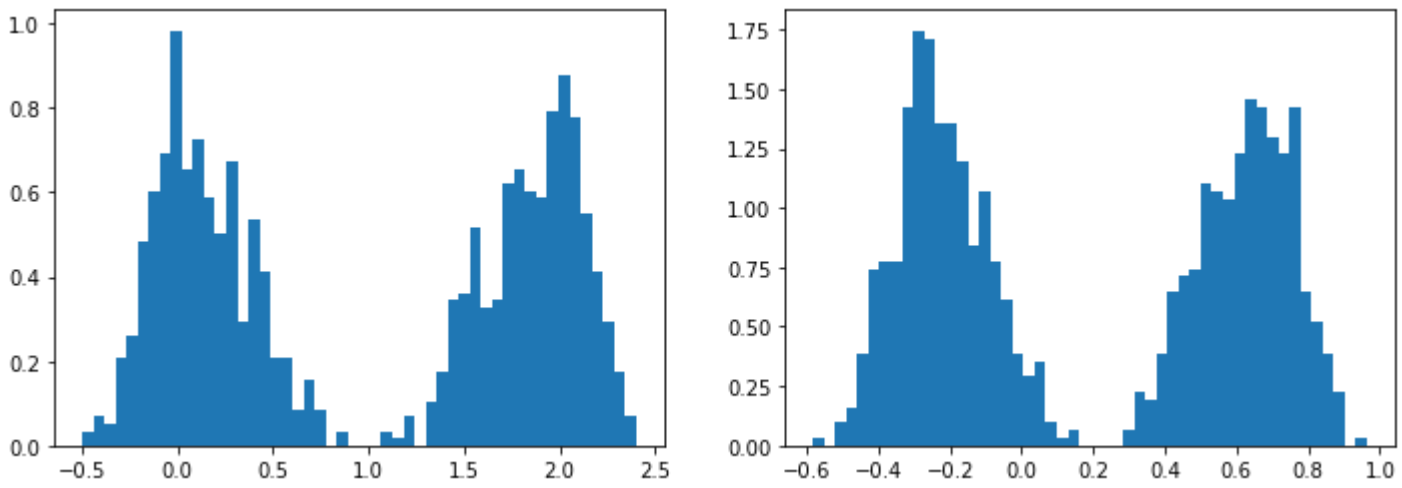
Out[63]:
```
TensorShape([1000, 2])
```

In [64]:
```
# Make contour plot
xx = np.linspace(-3.0, 3.0, 100)
yy = np.linspace(-2.0, 2.0, 100)
X, Y = np.meshgrid(xx, yy)

fig, ax = plt.subplots(1, 1)
Z = inv_cubed_normal.prob(np.dstack((X, Y)))
cp = ax.contourf(X, Y, Z)
fig.colorbar(cp) # Add a colorbar to a plot
ax.set_title('Filled Contours Plot')
ax.set_xlabel('x')
ax.set_ylabel('y')
plt.show()
```



In [65]:
```
# Plot histograms
plt.figure(figsize = (12, 4))
plt.subplot(1, 2, 1)
```

```
plt.hist(g[..., 0].numpy(), bins = 50, density = True)
plt.subplot(1, 2, 2)
plt.hist(g[..., 1].numpy(), bins = 50, density = True)
plt.show()
```



Training the bijector

In [66]:
```
# Create a mixture of four Gaussians
probs = [0.45, 0.55]
mix_gauss = tfd.Mixture (
    cat = tfd.Categorical(probs = probs),
    components = [tfd.Normal(loc = 2.3, scale = 0.4), tfd.Normal(loc = 0.8, scale = 0.4)]
)
```

In [67]:
```
# Create the dataset
x_train = mix_gauss.sample(10000)
x_train = tf.data.Dataset.from_tensor_slices(x_train)
x_train = x_train.batch(128)

x_valid = mix_gauss.sample(1000)
x_valid = tf.data.Dataset.from_tensor_slices(x_valid)
x_valid = x_valid.batch(128)

print(x_train.element_spec)
print(x_valid.element_spec)
```

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

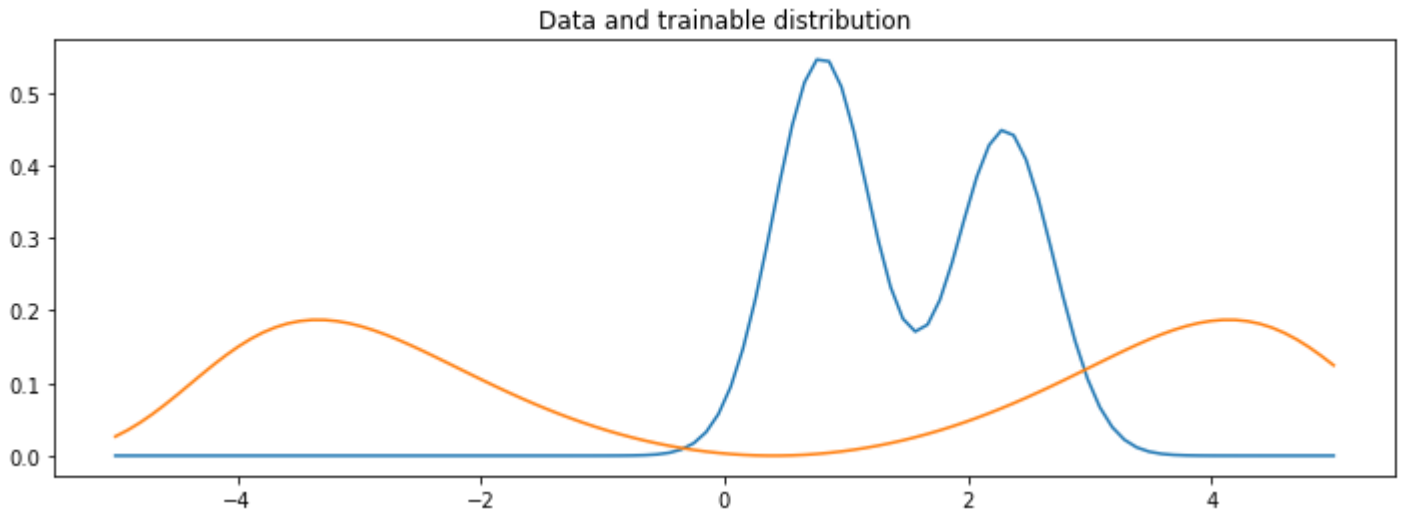
In [68]:
```
# Plot the data distribution
x = np.linspace(-5.0, 5.0, 100)
plt.plot(x, mix_gauss.prob(x))
plt.title('Data distribution')
plt.show()
```


In [69]:
```
# Make a trainable bijector
trainable_inv_cubic = tfb.Invert(Cubic(tf.Variable(0.25), tf.Variable(-0.1),))
trainable_inv_cubic.trainable_variables
```

Out[69]:
```
(<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=0.25>,
 <tf.Variable 'Variable:0' shape=() dtype=float32, numpy=-0.1>)
```

In [70]:
```
# Make a trainable transformed distribution
trainable_dist = tfd.TransformedDistribution(normal, trainable_inv_cubic)
```

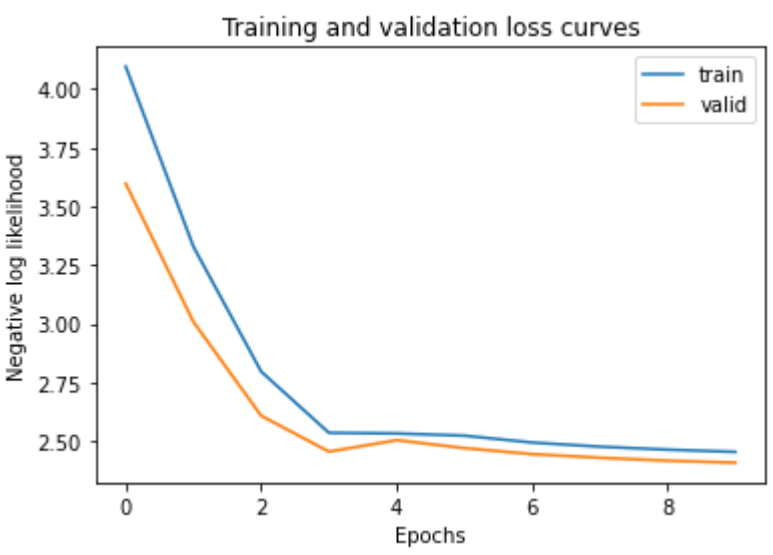
In [71]:
```
# Plot the data and learned distributions
x = np.linspace(-5.0, 5.0, 100)
plt.figure(figsize = (12, 4))
plt.plot(x, mix_gauss.prob(x), label = 'data')
plt.plot(x, trainable_dist.prob(x), label = 'trainable')
plt.title('Data and trainable distribution')
plt.show()
```



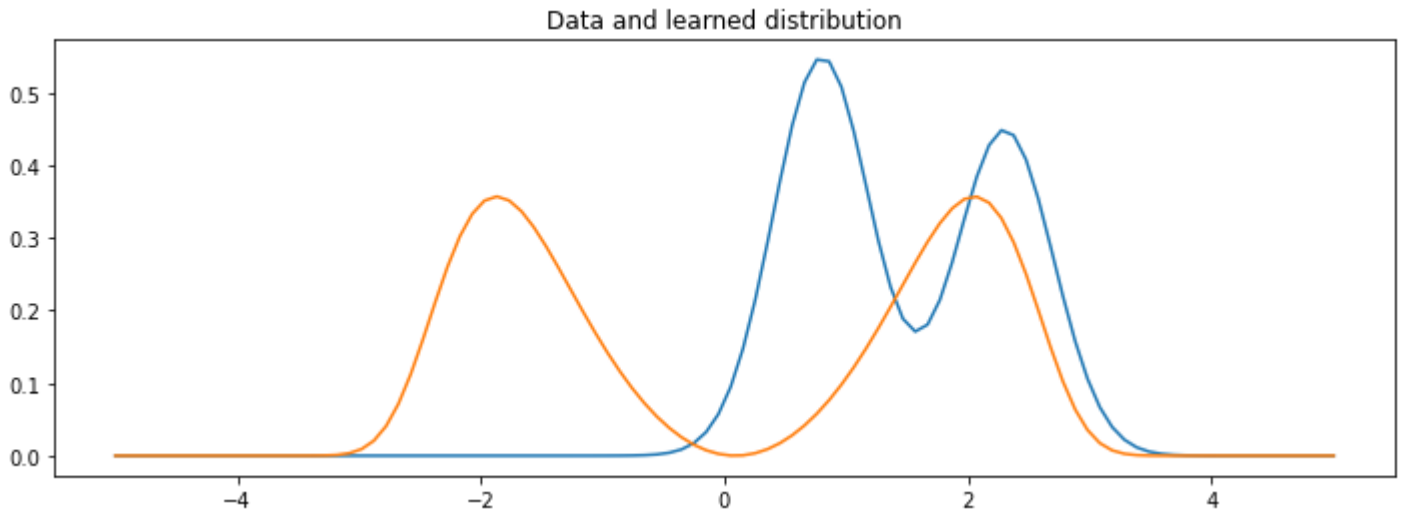
In [72]:
```
# Train the bijector
num_epochs = 10
opt = tf.keras.optimizers.Adam()
train_losses = []
valid_losses = []

for epoch in range(num_epochs):
    print("Epoch {}...".format(epoch))
    train_loss = tf.keras.metrics.Mean()
    val_loss = tf.keras.metrics.Mean()
    for train_batch in x_train:
        with tf.GradientTape() as tape:
            tape.watch(trainable_inv_cubic.trainable_variables)
            loss = -trainable_dist.log_prob(train_batch)
        train_loss(loss)
        grads = tape.gradient(loss, trainable_inv_cubic.trainable_variables)
        opt.apply_gradients(zip(grads, trainable_inv_cubic.trainable_variables))
    train_losses.append(train_loss.result().numpy())

    # Validation
    for valid_batch in x_valid:
        loss = -trainable_dist.log_prob(valid_batch)
        val_loss(loss)
    valid_losses.append(val_loss.result().numpy())
```

```
Epoch 0...
Epoch 1...
Epoch 2...
Epoch 3...
Epoch 4...
Epoch 5...
Epoch 6...
Epoch 7...
Epoch 8...
Epoch 9...
```

In [73]:
```
# Plot the learning curves
plt.plot(train_losses, label = 'train')
plt.plot(valid_losses, label = 'valid')
plt.legend()
plt.xlabel("Epochs")
plt.ylabel("Negative log likelihood")
plt.title("Training and validation loss curves")
plt.show()
```



In [74]:
```
# Plot the data and learned distributions
x = np.linspace(-5.0, 5.0, 100)
plt.figure(figsize = (12, 4))
plt.plot(x, mix_gauss.prob(x), label = 'data')
plt.plot(x, trainable_dist.prob(x), label = 'learned')
plt.title('Data and learned distribution')
plt.show()
```



In [75]:
```
# Display trainable variables
trainable_inv_cubic.trainable_variables
```

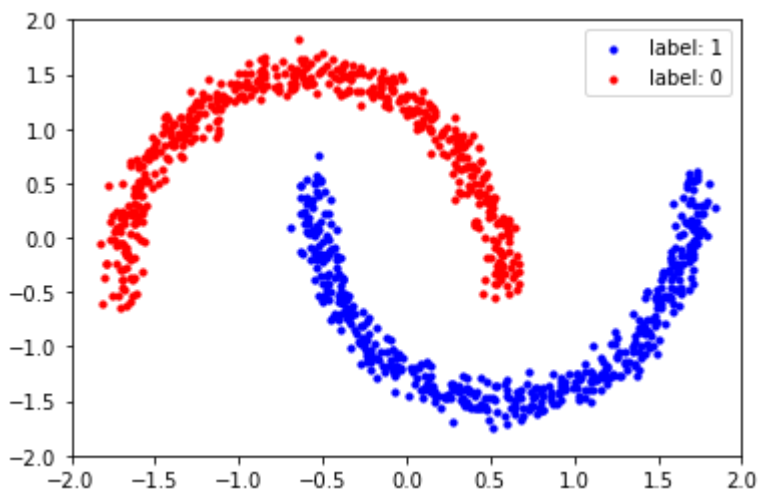
Out[75]:
```
(<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=0.47675532>,
 <tf.Variable 'Variable:0' shape=() dtype=float32, numpy=-0.04277669>)
```

Normalising flows

In [76]:
```
# Load dataset
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
n_samples = 1000
noisy_moons = datasets.make_moons(n_samples = n_samples, noise = .05)
X, y = noisy_moons
X_data = StandardScaler().fit_transform(X)
xlim, ylim = [-2, 2], [-2, 2]
```

In [77]:
```
# Plot with labels
y_label = y.astype(np.bool)
X_train, Y_train = X_data[..., 0], X_data[..., 1]
plt.scatter(X_train[y_label], Y_train[y_label], s = 10, color = 'blue')
plt.scatter(X_train[y_label == False], Y_train[y_label == False], s = 10, color = 'red')
plt.legend(['label: 1', 'label: 0'])
plt.xlim(xlim)
plt.ylim(ylim)
```

Out[77]:
```
(-2.0, 2.0)
```



In [78]:
```
# Define base distribution
base_distribution = tfd.Normal(loc = 0, scale = 1)
```

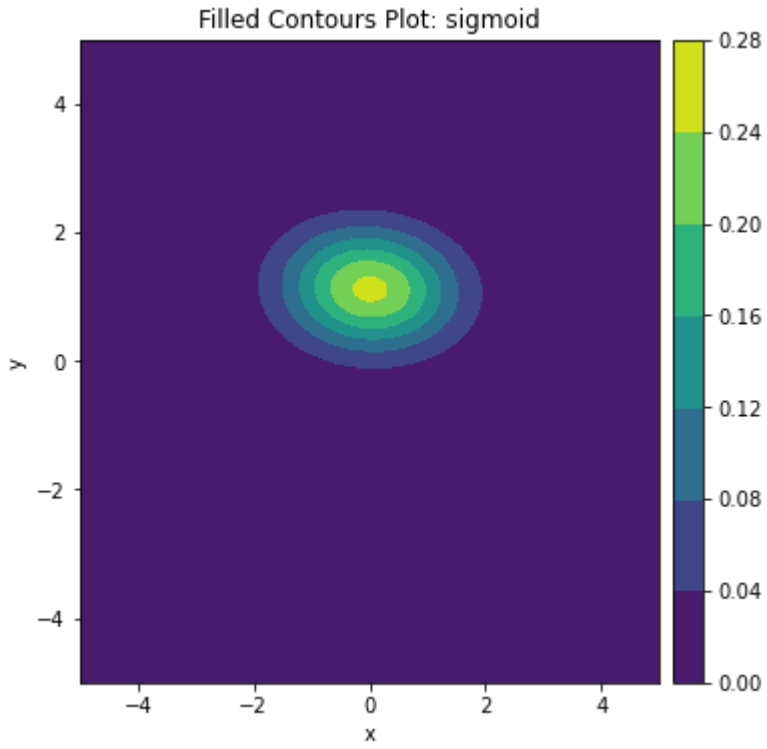
In [79]:
```
# Define the trainable distribution
def make_masked_autoregressive_flow(hidden_units = [16, 16], activation = 'relu'):
    made = tfb.AutoregressiveNetwork \
        (params = 2, event_shape = [2], hidden_units = hidden_units, activation = activation)
    return tfb.MaskedAutoregressiveFlow(shift_and_log_scale_fn = made)

trainable_distribution = tfd.TransformedDistribution \
    (base_distribution, make_masked_autoregressive_flow(), event_shape = [2])
```

In [80]:
```
from mpl_toolkits.axes_grid1 import make_axes_locatable
from tensorflow.compat.v1 import logging
logging.set_verbosity(logging.ERROR)
```

In [81]:
```
# Define a plot contour routine
def plot_contour_prob(dist, rows = 1, title = [''], scale_fig = 4):
    cols = int(len(dist) / rows)
    xx = np.linspace(-5.0, 5.0, 100)
    yy = np.linspace(-5.0, 5.0, 100)
    X, Y = np.meshgrid(xx, yy)

    fig, ax = plt.subplots(rows, cols, figsize = (scale_fig * cols, scale_fig * rows))
    fig.tight_layout(pad = 4.5)

    i = 0
    for r in range(rows):
        for c in range(cols):
```

```
            Z = dist[i].prob(np.dstack((X, Y)))
            if len(dist) == 1:
                axi = ax
            elif rows == 1:
                axi = ax[c]
            else:
                axi = ax[r, c]

            # Plot contour
            p = axi.contourf(X, Y, Z)

            # Add a colorbar
            divider = make_axes_locatable(axi)
            cax = divider.append_axes("right", size = "5%", pad = 0.1)
            cbar = fig.colorbar(p, cax = cax)

            # Set title and labels
            axi.set_title('Filled Contours Plot: ' + str(title[i]))
            axi.set_xlabel('x')
            axi.set_ylabel('y')

            i += 1
    plt.show()
```

In [82]:
```
# Plot contour
activation = 'sigmoid'
maf = tfd.TransformedDistribution \
    (base_distribution, make_masked_autoregressive_flow(activation = activation), event_shape = [2])
plot_contour_prob([maf], scale_fig = 6, title = [activation])
```

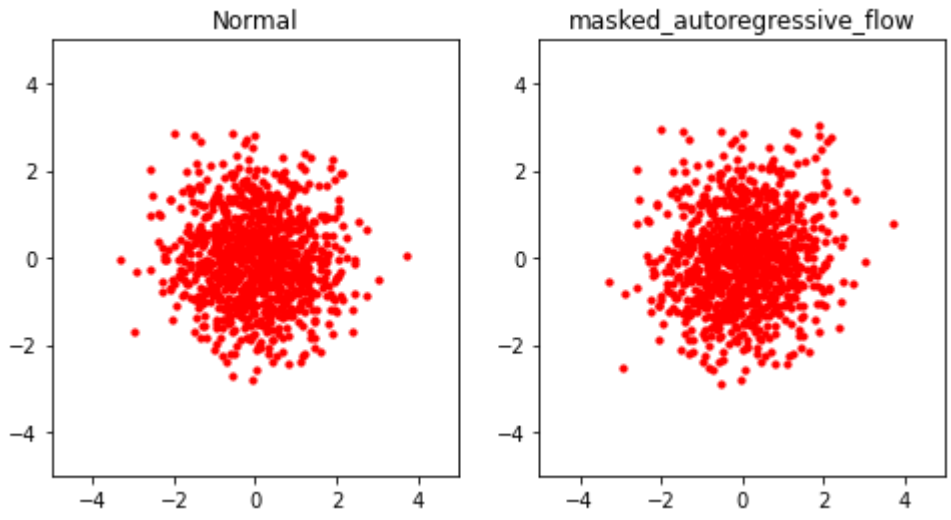


In [83]:
```
from tensorflow.keras.layers import Input
from tensorflow.keras import Model
```

In [84]:
```
# Make samples
x = base_distribution.sample((1000, 2))
names = [base_distribution.name, trainable_distribution.bijector.name]
samples = [x, trainable_distribution.bijector.forward(x)]
```

In [85]:
```
# Define a scatter plot routine for the bijectors
def _plot(results, rows=1, legend = False):
    cols = int(len(results) / rows)
    f, arr = plt.subplots(rows, cols, figsize = (4 * cols, 4 * rows))
    i = 0
    for r in range(rows):
        for c in range(cols):
            res = results[i]
            X, Y = res[..., 0].numpy(), res[..., 1].numpy()
            if rows == 1:
                p = arr[c]
            else:
                p = arr[r, c]
            p.scatter(X, Y, s = 10, color = 'red')
            p.set_xlim([-5, 5])
            p.set_ylim([-5, 5])
            p.set_title(names[i])

            i += 1
```

In [86]:
```
# Plot
_plot(samples)
```



Training a MaskedAutoregressiveFlow bijector

In [87]:
```
from tensorflow.keras.callbacks import LambdaCallback
```

In [88]:
```
# Define a training routine
def train_dist_routine(trainable_distribution, n_epochs = 200, batch_size = None, n_disp = 100):
    x_ = Input(shape = (2,), dtype = tf.float32)
    log_prob_ = trainable_distribution.log_prob(x_)
    model = Model(x_, log_prob_)
    model.compile(optimizer = tf.optimizers.Adam(), loss = lambda _, log_prob: -log_prob)

    ns = X_data.shape[0]
    if batch_size is None:
        batch_size = ns

    # Display the loss every n_disp epoch
    epoch_callback = LambdaCallback (
        on_epoch_end = lambda epoch, logs:
            print('\n Epoch {}/{}'.format(epoch + 1, n_epochs, logs),
                  '\n\t ' + (': {:.4f}, '.join(logs.keys()) + ': {:.4f}').format(*logs.values()))
            if epoch % n_disp == 0 else False
    )

    history = model.fit (
        x = X_data, y = np.zeros((ns, 0), dtype=np.float32),
        batch_size = batch_size, epochs = n_epochs,
        validation_split = 0.2, shuffle = True, verbose = False,
        callbacks = [epoch_callback]
    )
    return history
```

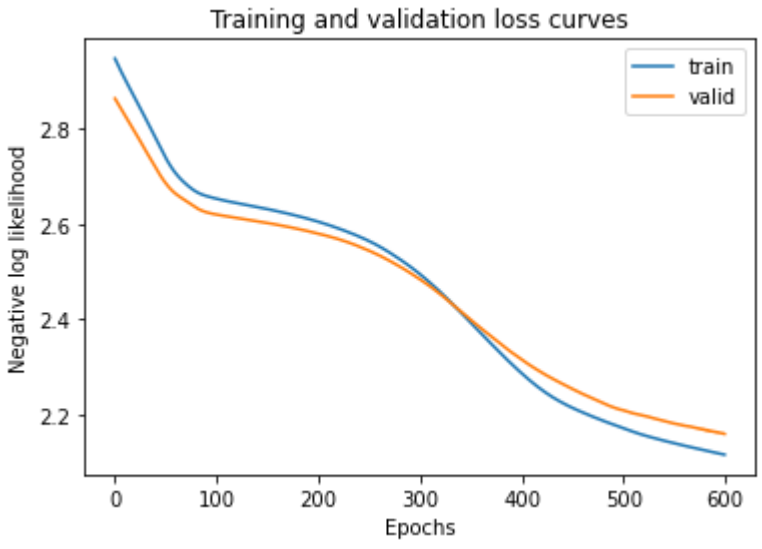
In [89]:
```
# Train the distribution
history = train_dist_routine(trainable_distribution, n_epochs = 600, n_disp = 50)
```

```
Epoch 1/600
        loss: 2.9460, val_loss: 2.8624

Epoch 51/600
        loss: 2.7376, val_loss: 2.6847

Epoch 101/600
        loss: 2.6527, val_loss: 2.6195

Epoch 151/600
        loss: 2.6311, val_loss: 2.6019

Epoch 201/600
        loss: 2.6045, val_loss: 2.5796

Epoch 251/600
        loss: 2.5639, val_loss: 2.5437

Epoch 301/600
        loss: 2.4945, val_loss: 2.4838

Epoch 351/600
        loss: 2.3925, val_loss: 2.3987

Epoch 401/600
        loss: 2.2868, val_loss: 2.3155

Epoch 451/600
        loss: 2.2146, val_loss: 2.2542

Epoch 501/600
        loss: 2.1721, val_loss: 2.2096

Epoch 551/600
        loss: 2.1409, val_loss: 2.1822
```

In [90]:
```
# Get losses
train_losses = history.history['loss']
valid_losses = history.history['val_loss']
```

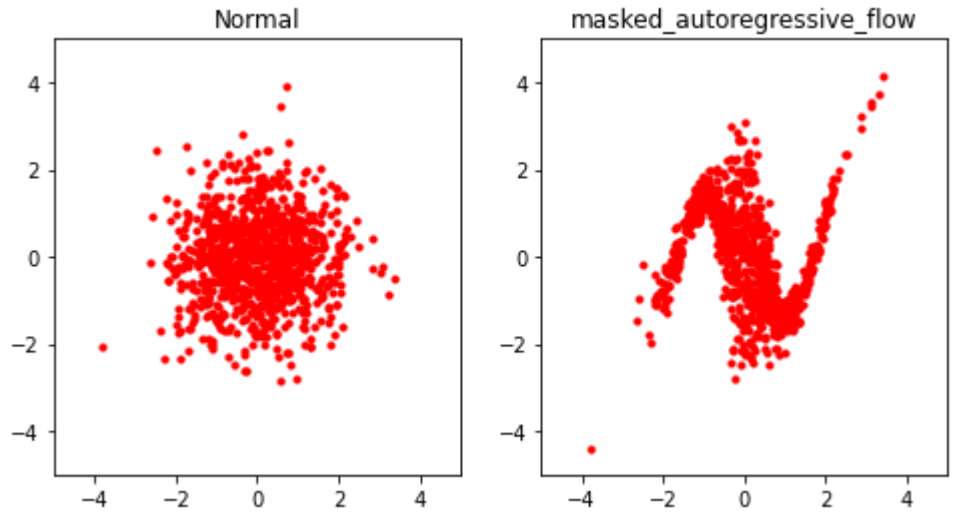
In [91]:
```
# Plot loss vs epoch
plt.plot(train_losses, label = 'train')
plt.plot(valid_losses, label = 'valid')
plt.legend()
plt.xlabel("Epochs")
```

```
plt.ylabel("Negative log likelihood")
plt.title("Training and validation loss curves")
plt.show()
```

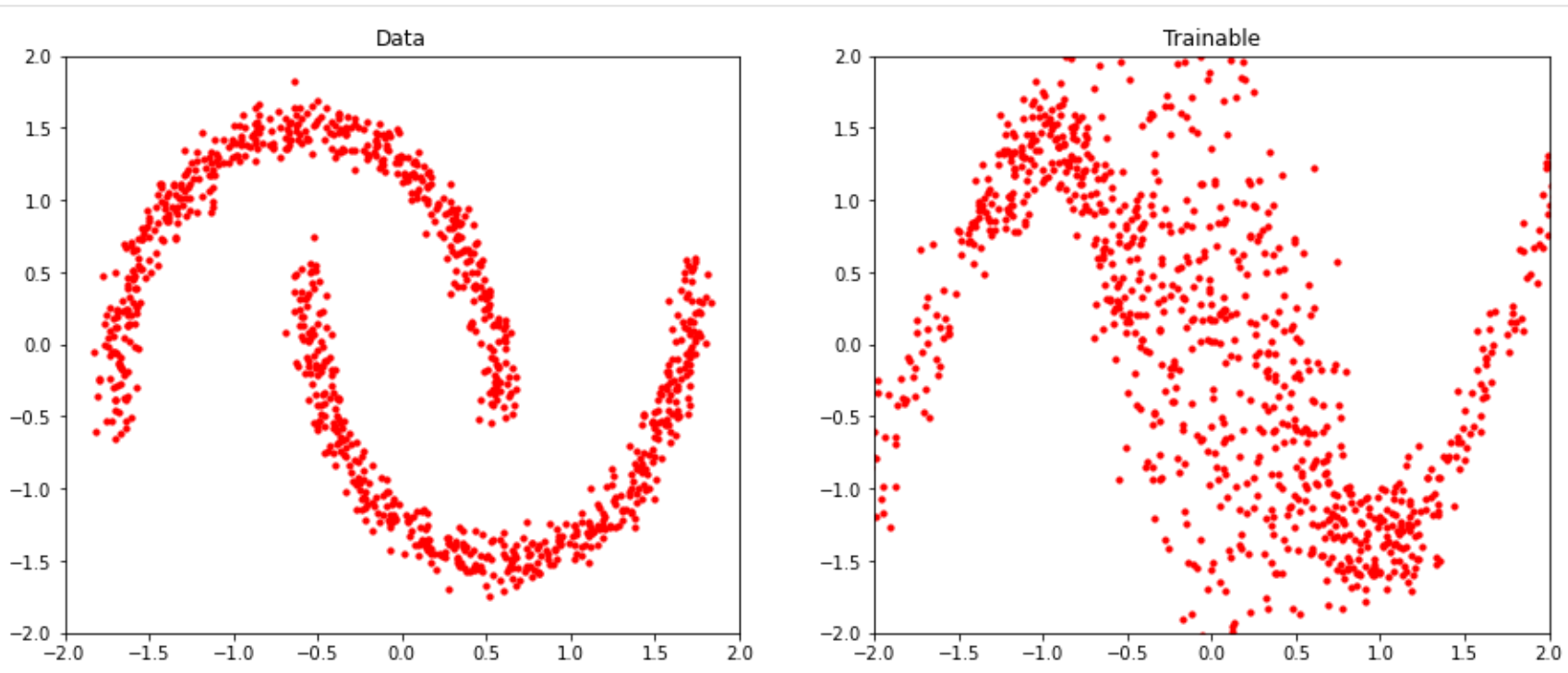


In [92]:
```
# Make samples
x = base_distribution.sample((1000, 2))
names = [base_distribution.name, trainable_distribution.bijector.name]
samples = [x, trainable_distribution.bijector.forward(x)]
```

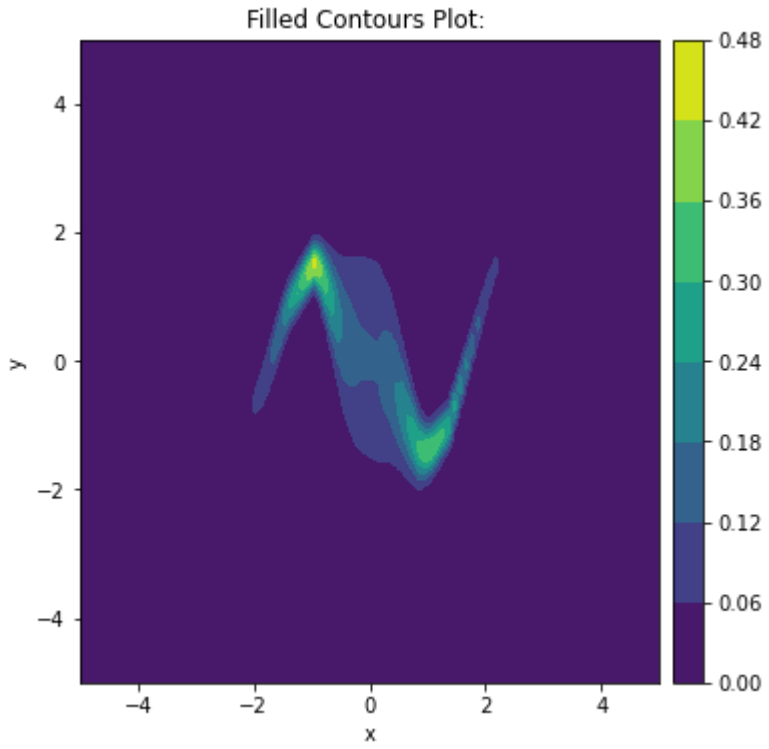
In [93]:
```
# Plot
_plot(samples)
```



In [94]:
```
# Define a plot routine
def visualize_training_data(samples):
    f, arr = plt.subplots(1, 2, figsize = (15, 6))
    names = ['Data', 'Trainable']
    samples = [tf.constant(X_data), samples[-1]]

    for i in range(2):
        res = samples[i]
        X, Y = res[..., 0].numpy(), res[..., 1].numpy()
        arr[i].scatter(X, Y, s = 10, color = 'red')
        arr[i].set_xlim([-2, 2])
        arr[i].set_ylim([-2, 2])
        arr[i].set_title(names[i])

visualize_training_data(samples)
```



In [95]:
```
# Plot contour
plot_contour_prob([trainable_distribution], scale_fig = 6)
```



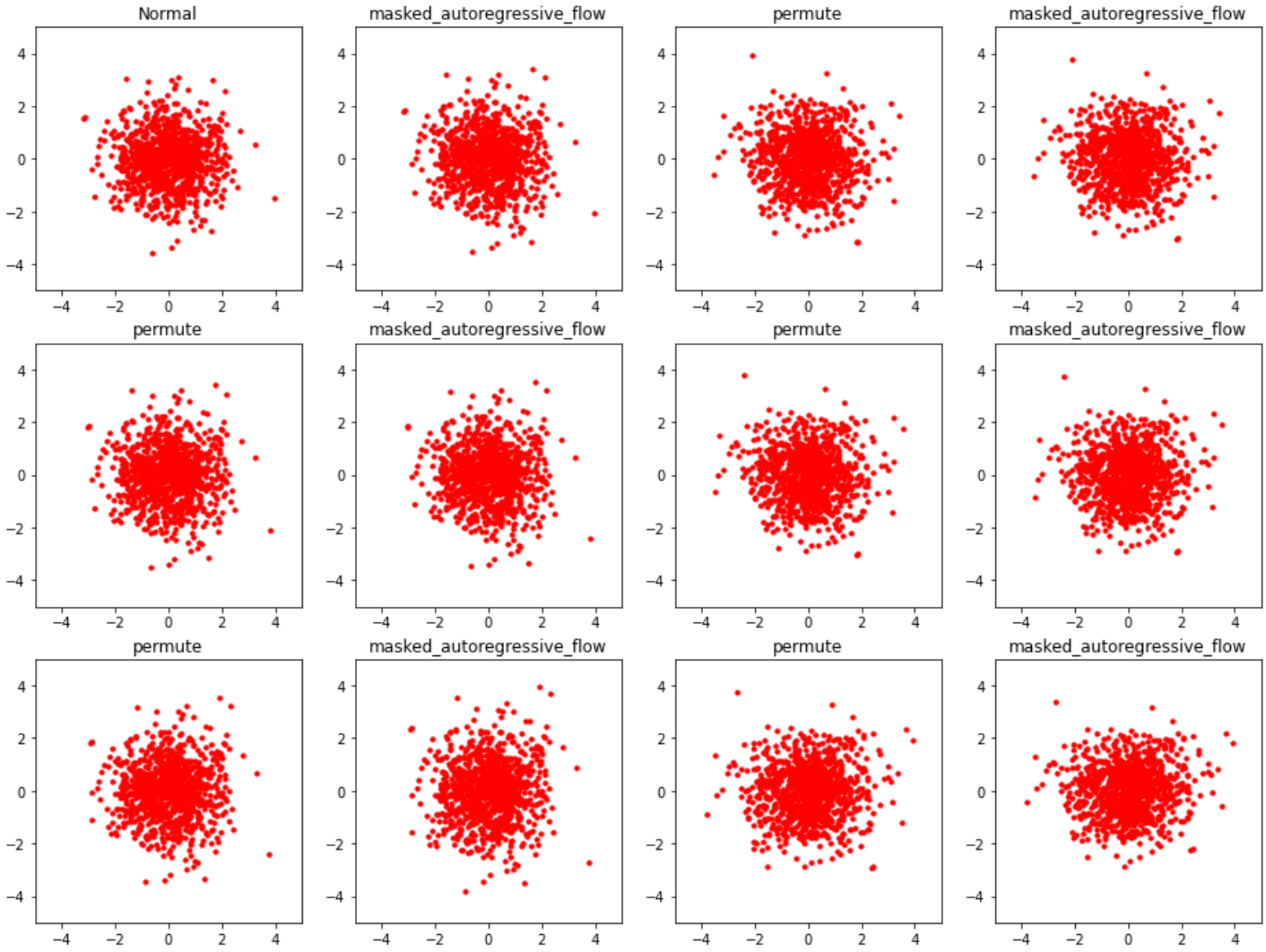
Training a chain of MaskedAutoregressiveFlow bijectors

In [96]:
```
# Define a more expressive model
num_bijectors = 6
bijectors = []

for i in range(num_bijectors):
    masked_auto_i = make_masked_autoregressive_flow(hidden_units = [256, 256], activation = 'relu')
    bijectors.append(masked_auto_i)
    bijectors.append(tfb.Permute(permutation = [1, 0]))

flow_bijector = tfb.Chain(list(reversed(bijectors[:-1])))
```

In [97]:
```
# Define the trainable distribution
trainable_distribution = tfd.TransformedDistribution \
    (distribution = base_distribution, bijector = flow_bijector, event_shape = [2])
```

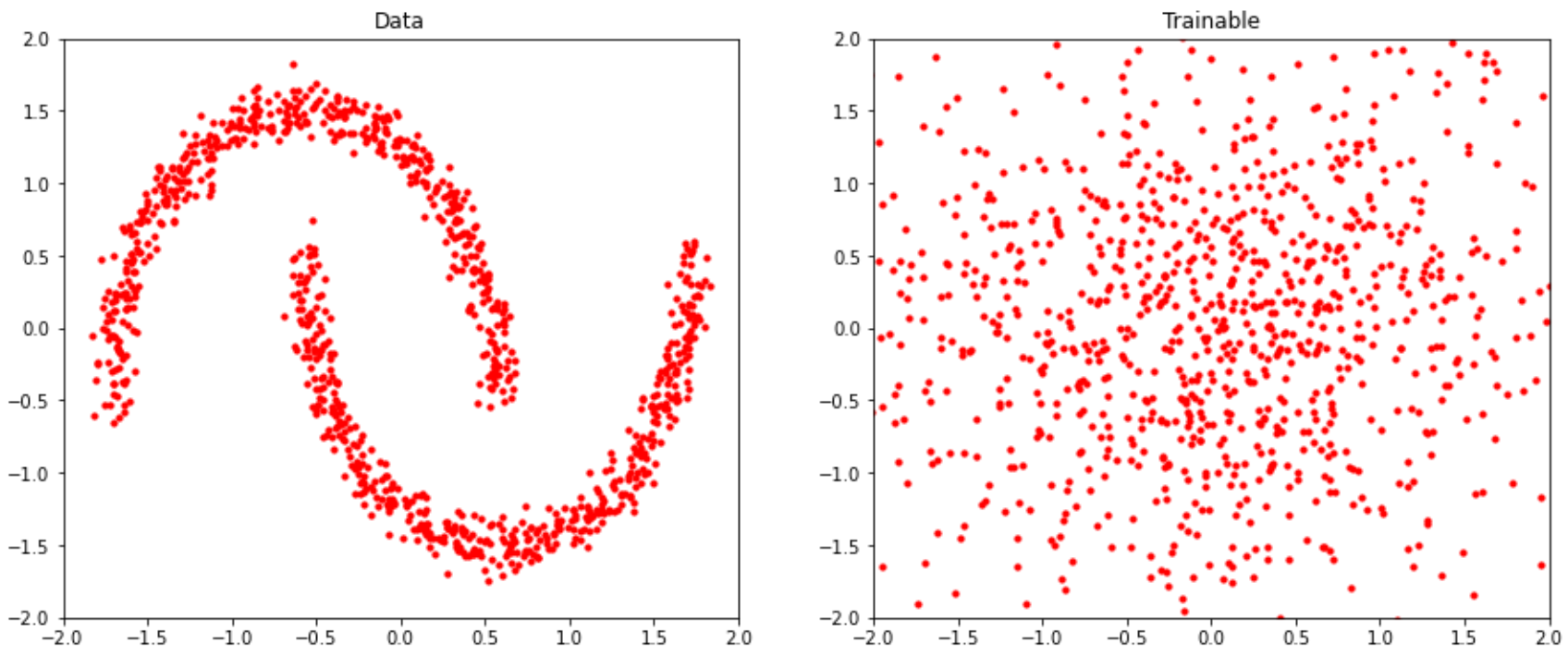
In [98]:
```
# Make samples
def make_samples():
    x = base_distribution.sample((1000, 2))
    samples = [x]
    names = [base_distribution.name]
    for bijector in reversed(trainable_distribution.bijector.bijectors):
        x = bijector.forward(x)
        samples.append(x)
        names.append(bijector.name)
    return names, samples

names, samples = make_samples()
```

In [99]:
```
# Plot
_plot(samples, 3)
```



In [100…

```
# Plot
visualize_training_data(samples)
```



In [101…

```
# Train the distribution
history = train_dist_routine(trainable_distribution, n_epochs = 600, n_disp = 50)
```

```
Epoch 1/600
        loss: 2.9169, val_loss: 2.6941

Epoch 51/600
        loss: 2.1496, val_loss: 2.1582

Epoch 101/600
        loss: 1.8858, val_loss: 1.9126

Epoch 151/600
        loss: 2.0485, val_loss: 2.0945

Epoch 201/600
        loss: 1.9392, val_loss: 1.8634

Epoch 251/600
        loss: 1.3229, val_loss: 1.4462

Epoch 301/600
        loss: 1.2615, val_loss: 1.3872

Epoch 351/600
        loss: 1.1725, val_loss: 1.3308

Epoch 401/600
        loss: 1.1442, val_loss: 1.3434

Epoch 451/600
```

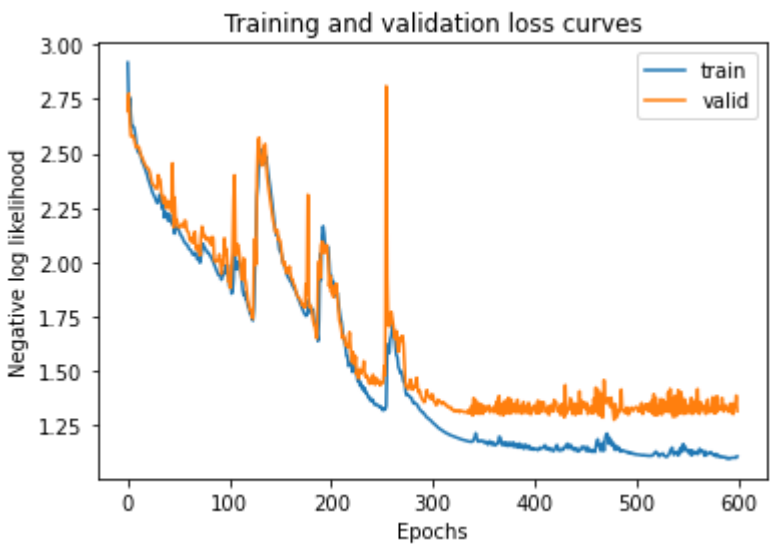
```
        loss: 1.1349, val_loss: 1.3283

Epoch 501/600
        loss: 1.1160, val_loss: 1.3346

Epoch 551/600
        loss: 1.1374, val_loss: 1.3436
```

In [102…

```
# Get losses
train_losses = history.history['loss']
valid_losses = history.history['val_loss']
```

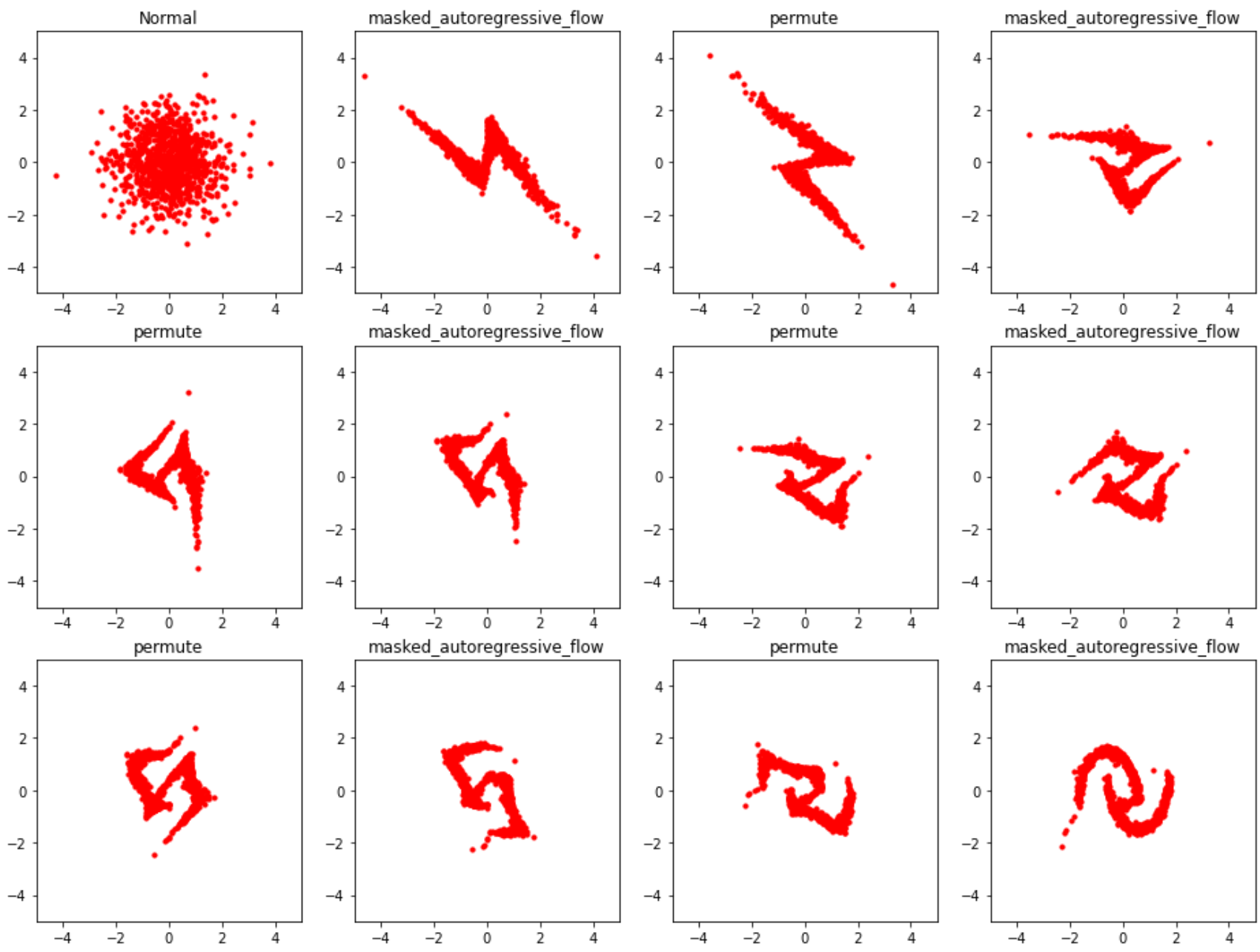
In [103…

```
# Plot loss vs epoch
plt.plot(train_losses, label = 'train')
plt.plot(valid_losses, label = 'valid')
plt.legend()
plt.xlabel("Epochs")
plt.ylabel("Negative log likelihood")
plt.title("Training and validation loss curves")
plt.show()
```



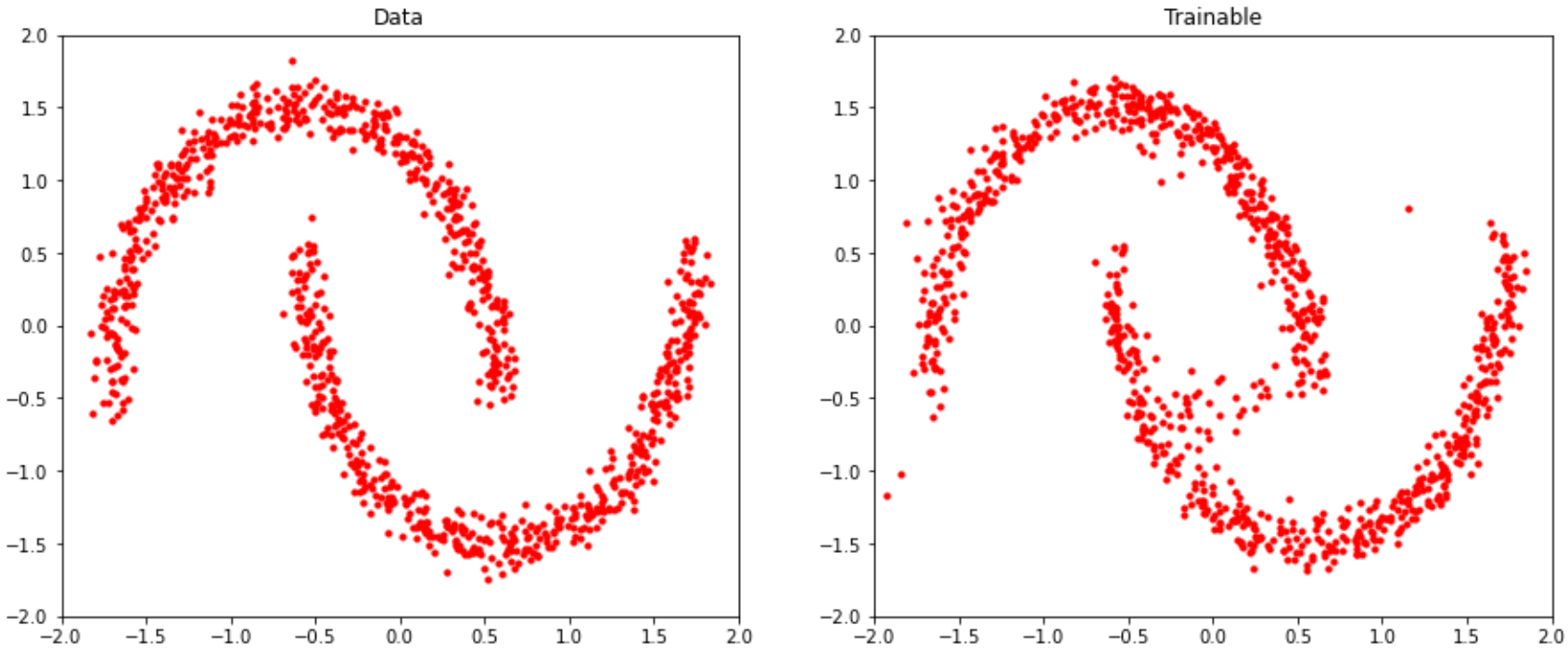
In [104…

```
# Make samples and plot
names, samples = make_samples()
_plot(samples, 3)
```



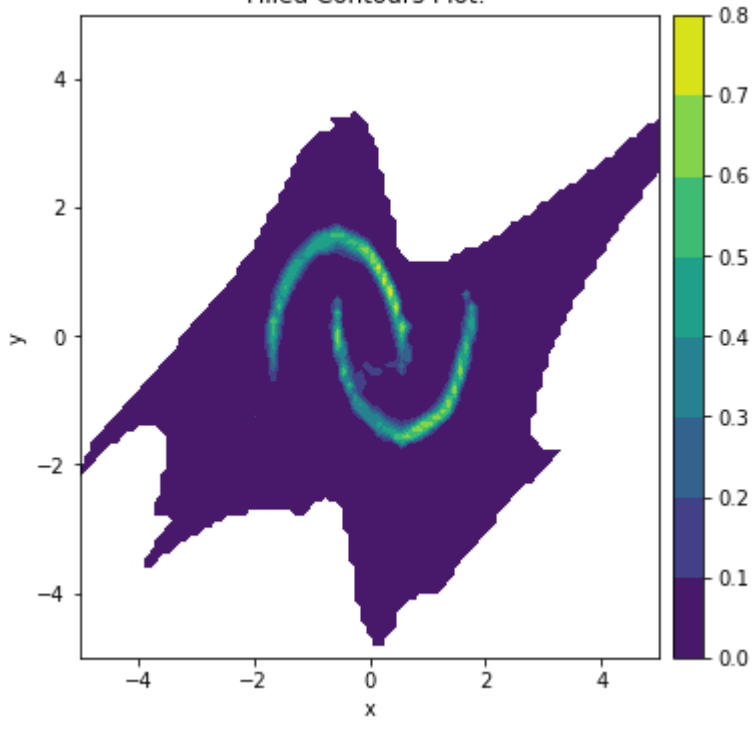
In [105…

```
# Plot
visualize_training_data(samples)
```



In [106…

```
# Plot
plot_contour_prob([trainable_distribution], scale_fig = 6)
```



Variational autoencoders

This reading is a review of the variational autoencoder (VAE) algorithm, that we will be working with this week.

It is split into four sections:

- Section 1 This section motivates and describes the structure of a VAE, which is that of a latent variable model. A VAE is a latent variable model in which the 'encoder' and 'decoder' are neural networks
- Section 2 The second section derives the evidence lower bound (ELBO). The ELBO is the objective function that is used to fit a VAE
- Section 3 In practice, a reparameterisation trick, detailed in the third section, is used when estimating the ELBO
- Section 4 The reading concludes by highlighting that the ELBO can be estimated more precisely if part of it is evaluated analytically, and by providing a recipe for fitting a VAE.

The sections of this reading summarise the main results of the following reference:

- D. P. Kingma and M. Welling. "Auto-Encoding Variational Bayes", 2014. https://arxiv.org/abs/1312.6114 (https://arxiv.org/abs/1312.6114)

Variational autoencoders have been used for anomaly detection, data compression, image denoising, and for reducing dimensionality in preparation for some other algorithm or model. These applications vary in their use of a trained VAE's encoder and decoder: some use both, while others use only one.

The key point of similarity between a VAE and an autoencoder is that they both use neural networks for tasks that can be interpreted as compression and reconstruction. Additionally, a term in the ELBO resembles the reconstruction error of an autoencoder. Apart from these similarities, VAEs are quite different from autoencoders. Crucially, a VAE is an unsupervised generative model, whereas an autoencoder is not. An autoencoder is sometimes described as being 'self-supervised'. A VAE on the other hand describes the variability in the observations, and can be used to synthesise observations.

Latent variables and the latent variable model

A latent variable is a random variable that cannot be conditioned on for inference because its value is not known. 'Latent' means hidden. Latent variables do not need to correspond to real quantities. Sometimes models that outwardly do not involve latent quantities are more conveniently expressed by imagining that they do. A perfect example of this is the mixture of Gaussians model: observations can be generated by sampling a label from a categorical distribution, then drawing from the Gaussian in the mixture that has that label.

A latent variable model underlies the variational autoencoder: some latent random variable Z is assumed to have distribution p_{θ_*}, and the observation X is assumed to be distributed according to the conditional distribution $p_{\theta_*}(x|z)$. X may be either continuous or discrete.

Given some data, our objective is to obtain a maximum likelihood estimate for θ, denoted θ_{ML}. Once θ_{ML} is available, then the distribution of the observable given the latent variable, $p_{\theta_{ML}}(x|z)$, and the marginal likelihood of an observation, $p_{\theta_{ML}}(x)$, can be used.

This model could be fit by maximising the marginal likelihood,

$$p_\theta(x) = \int p_\theta(x|z)p_\theta(z)dz,$$

If this likelihood or its gradient can be efficiently evaluated or approximated, then maximising it with respect to θ is straightforward. Alternatively, the marginal likelihood may be intractable while the posterior $p_\theta(z|x)$ is known or can be efficiently approximated, in which case the EM algorithm could be used.

A simple approach to estimating $p_\theta(x)$ is to take samples z_i ($i \in I$) from $p_\theta(z)$, then take the average of their $p_\theta(x|z_i)$ values. The problem with this method is that if z is high-dimensional, then a very large sample is required to estimate $p_\theta(x)$ well.

Variational inference provides an alternative approach to fitting the model. The high-level idea is this: approximate $p_\theta(z|x)$, then use this approximation to estimate a lower bound on $\log p_\theta(x)$. θ can then be updated based on this lower bound.

The first step in this variational approach is to introduce an approximating distribution for $p_\theta(z|x)$. Call this approximating distribution $q_\phi(z|x)$, where ϕ is its parameter. q_ϕ is fit to p_θ by minimising the Kullback- Leibler divergence

$$D_{KL}(q_\phi(z|x)||p_\theta(z|x))$$

The reasons for this choice of objective function are discussed in more detail in a reading exclusively on the KL divergence later in the week. Its most important properties for now are that it is non-negative, and is zero if and only if q_ϕ and p_θ are equal almost everywhere.

A bound on the marginal log-likelihood

The marginal log-likelihood of a single observation x can be written

$$\log p_\theta(x) = -\log p_\theta(z|x) + \log p_\theta(x,z)$$

Adding and subtracting $\log q_\phi(z|x)$ to the right-hand side of this equation, rearranging the logs, then taking the expectation of both sides under $q_\phi(z|x)$, results in

$$\log p_\theta(x) = E_{z\sim q_\phi}\left[\log \frac{q_\phi(z|x)}{p_\theta(z|x)}\right] + E_{Z\sim q_\phi}\left[\log p_\theta(x,z) - \log q_\phi(z|x)\right], \qquad (1)$$

$$D_{KL}(\widetilde{q_\phi}||p_\theta)$$

where the definition of the KL divergence $D_{KL}(f||g)$ for distributions f and g where $g(x) = 0 \Rightarrow f(x) = 0$ is given by

$$D_{KL}(f||g) = E_{x\sim f}\left[\log \frac{f(X)}{g(X)}\right].$$

If θ is held fixed in (1), then $\log p_\theta(x)$ is fixed too. Because the Kullback-Leibler divergence is non-negative, increasing

$$E_{z\sim q_\phi}\left[\log p_\theta(x,z) - q(z|x)\right]$$

with respect to ϕ will reduce $D_{KL}(q_\phi||p_\theta)$, improving our approximating distribution. Additionally, we have the inequality

$$\log p_\theta(x) \geq E_{Z\sim q_\phi}\left[\log p_\theta(x,z) - q_\phi(z|x)\right] =: \mathrm{L}(\theta,\phi;x) \qquad (2)$$

This lower bound on the marginal log-likelihood, $\mathrm{L}(\theta,\phi;x)$, is the objective function maximised in variational inference. It is known as the evidence lower bound (ELBO), since the marginal likelihood is the Bayesian evidence of posterior inference in the latent variable model. Notice that L does not involve evaluating $p_\theta(z|x)$, which we assumed was intractable.

Usually, an analytic expression for the entire ELBO is unavailable. Instead, a Monte Carlo estimate of it can be made. Two estimators for the ELBO are described in Kingma and Welling's original paper. The simplest uses a samples $\{z_j\}_{j=1}^L$ from $q_\phi(z|x)$:

$$\hat{\mathrm{L}}^A(\theta,\phi;x) := \frac{1}{L}\sum_{j=1}^{L} \log p_\theta(x,z_j) - \log q_\phi(z_j|x) \qquad (3)$$

In principle, θ and ϕ can now be updated via stochastic gradient ascent using the derivatives of L . Unfortunately, there is a fly in the ointment: the z_j values are not differentiable functions of ϕ, since they are samples. To remove this obstacle to evaluating the gradients, a trick is used.

The reparameterisation trick

The reparameterisation trick enables derivatives to be propagated to the parameters of a distribution that is sampled from when computing the objective. The essence of the trick is to change how sampling is executed. Rather than sampling from $q_\phi(z|x)$ directly, we instead sample *auxiliary variables* ϵ_j from a distribution $p(\epsilon)$ that is not parameterised by ϕ, then pass them through a ϕ-dependent

deterministic transformation $g_\phi(\epsilon, x)$.

We therefore need to choose the distribution $p(\epsilon)$ and transformation $g_\phi(\epsilon, x)$ so that $q_\phi(z|x)$ has the same distribution as $g_\phi(\epsilon; x)$, where $\epsilon \sim p(\epsilon)$, i.e. our sampling procedure is equivalent to sampling from $q_\phi(z|x)$.

For the time being, assume that we know of a $g_\phi(\epsilon, x)$ and $p(\epsilon)$ that satisfy this criterion.

We can then re-write an estimate from the $\hat{\mathrm{L}}^A$ as expressed in (3) in terms of the auxiliary samples:

$$\hat{\mathrm{L}}^A(\theta, \phi; x) := \frac{1}{L}\sum_{j=1}^{L} \log p_\theta(x, z_j) - \log q_\phi(z_j|x), \qquad \text{where } z_j = g_\phi(\epsilon_j, x)$$

and the ϵ_j have been sampled from $p(\epsilon)$. This quantity is differentiable with respect to both ϕ and θ, so it can be used for parameter updates in a minibatch gradient ascent algorithm.

For some distributions $q_\phi(z|x)$, an obvious choice of $p(\epsilon)$ and g_ϕ is available. For instance, if $q_\phi(z|x)$ is the density of the multivariate normal $N(\mu, \Sigma)$ with $\phi = (\mu, \Sigma)$, then

$$p(\epsilon) = N(\mathbf{0}, \mathbf{I}), \quad g_\phi(\epsilon, x) = \mu + L\epsilon, \quad \text{where } LL^T = \Sigma$$

results in $q_\phi(z|x)$ and $g_\phi(\epsilon; x)$ being equal in distribution, and g_ϕ being differentiable with respect to ϕ.

A lower-variance estimator for the ELBO

Referring back to equation (2), we can see that the ELBO can be re-written as

$$\mathrm{L}(\theta, \phi; x) = -D_{KL}(q_\phi(z|x)||p_\theta(z)) + E_{z\sim q_\phi}\left[\log p_\theta(x|z)\right] \qquad (4)$$

If the KL divergence term, an integral, in this expression can be evaluated analytically, then we might expect the estimator

$$\hat{\mathrm{L}}^B(\theta, \phi; x) := -D_{KL}(q_\phi(z|x)||p_\theta(z)) + \frac{1}{L}\sum_{j=1}^{L} \log p_\theta(x|z_j)$$

where $z_j = g_\phi(\epsilon_j, x)$, $\epsilon_j \sim p(\epsilon)$, to have lower variance than $\hat{\mathrm{L}}^A$. This is the second ELBO estimator introduced in Kingma and Welling's paper. Usually $q_\phi(z|x)$ and $p_\theta(z)$ are chosen to be Gaussians, meaning that an analytic expression for the divergence can be computed.

Equation (4) helps us to understand the components of the ELBO. The negative of the KL divergence between $q_\phi(z|x)$ and $p_\theta(z)$ penalises $q_\phi(z|x)$ for placing probability mass in locations where $p_\theta(z)$ does not. This has the effect of regularizing $q_\phi(z|x)$.

The second term favours parameter values for which the reconstruction error is small. Given an input $\bar{x}$, an encoding z is sampled from $q_\phi(z|\bar{x})$, then the probability density of a perfect reconstruction is $p_\theta(\bar{x}|z)$. Averaging over the encodings via $E_{z\sim q_\phi}$ results in utility being placed on parameters that yield probable reconstruction of the input $\bar{x}$.

Conclusion

To specify and fit a variational autoencoder, choose $p_\theta(z)$, $p_\theta(x|z)$, and $q_\phi(x|z)$, then repeat:

1. Sample a minibatch of observations $x_1, x_2, \ldots, x_n$ and evaluate an estimate of its ELBO,

$$\sum_{j=1}^{n} \hat{\mathrm{L}}(\theta, \phi; x_j)$$

   where $\hat{\mathrm{L}}$ is either $\hat{\mathrm{L}}^A$ or $\hat{\mathrm{L}}^B$. In either case, for each x_j, L samples from $q_\phi(z|x_j)$ will be required. These samples should be taken using the reparameterisation trick.

2. Use the gradients of the ELBO estimate to update the parameters θ and ϕ.

Often $q_\phi(z|x)$ is chosen to be a multivariate normal distribution, with a neural network mapping x to its mean and covariance matrix, and ϕ is the parameter vector of the neural network. For continuous data, the distribution $p_\theta(x|z)$ is often also a multivariate normal distribution. Again, a neural network maps z to a mean and covariance matrix, and this neural network is parameterised by θ. For discrete data, often Bernoulli or categorical distributions are used. Typically $p_\theta(z)$ is fixed as a standard multivariate normal distribution.

The model can be sampled from by drawing z from $p_\theta(z)$, then sampling x from $p_\theta(x|z)$. Encodings associated with an observation x can be retrieved by sampling from $q_\phi(z|x)$.

Further reading and resources

In addition to the Kingma and Welling paper (https://arxiv.org/abs/1312.6114) cited above, the following is a general introduction to variational inference, which you may find is useful context for VAEs.

- D. M. Blei, A. Kucukelbir, and J. D. McAuliffe. "Variational Inference: A Review for Statisticians", 2016. https://arxiv.org/abs/1601.00670 (https://arxiv.org/abs/1601.00670)

Kullback-Leibler divergence

This reading will review the definition of the Kullback-Leibler (or KL) divergence, look at some of its important properties, see how it can be computed in practice with TensorFlow Probability.

In [1]:
```
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions
print("TF version:", tf.__version__)
print("TFP version:", tfp.__version__)

# Additional packages for the reading
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.patches import Ellipse
```

```
TF version: 2.3.0
TFP version: 0.11.0
```

Introduction

As you have already seen, the KL divergence is used in variational inference to score the dissimilarity between two distributions. In this reading, we will examine KL divergence more closely. We will see the definition of the KL divergence and some important properties, as well as how it can be computed using `tfd.kl_divergence` and Monte Carlo estimation.

Definition of the Kullback-Leibler divergence

Given two probability density or mass functions $q(x)$ and $p(x)$, the Kullback-Leibler divergence between them is defined as

$$D_{KL}\left[q \,||\, p\right] = \begin{cases} \mathrm{E}_{X\sim q}\left[\log q(X) - \log p(X)\right] & \text{if } p(x) = 0 \implies q(x) = 0, \\ \infty & \text{otherwise.} \end{cases}$$

The condition $p(x) = 0 \implies q(x) = 0$ - *absolute continuity* - ensures that the $\log$ in the expectation is well-defined for all x in the support of q.

As was mentioned, the KL divergence is a score for the disagreement of two distributions in their placement of probability mass. A smaller score indicates a greater degree of agreement.

Properties

The Kullback-Leibler divergence is asymmetric. In general,

$$D_{KL}\left[q \,||\, p\right] \neq D_{KL}\left[p \,||\, q\right]$$

In variational inference, q is the approximating distribution, while p is the distribution being approximated. The other KL divergence - $D_{KL}[p \,||\, q]$ - is also sometimes used as a loss function, for reasons that will become clear later in this reading.

Gibbs' inequality

A crucial property of the KL divergence is that for all q and p,

$$D_{KL}\left[q \,||\, p\right] \geq 0,$$

with equality if and only if $q(x) = p(x)$ almost everywhere. This property is very useful when we are trying to learn a q that is similar to a p: if $D_{KL}[q \,||\, p] = 0$, then we know that q is identical to p.

What causes KL divergence to increase?

As an example, take $q(x)$ and $p(x)$ to be probability mass functions, and let X be q's support. Provided q is absolutely continuous with respect to p, we have

$$D_{KL}\left[q \,||\, p\right] = \sum_{x\in \mathrm{X}} q(x) \log \frac{q(x)}{p(x)}.$$

Values of x that p assigns mass to but q does not do not feature in this sum. Superficially, this may suggest that divergence is not increased if q fails to place mass where p does. However, q is a probability mass function, so will inevitably place more mass than p at some other value(s) of x. At those other locations, $\log q(x)/p(x) > 0$, so the divergence is increased.

On the other hand, if q places probability mass where p does not, then $D_{KL}\left[q \,||\, p\right]$ is $+\infty$ - the KL divergence severely penalizes q for locating probability mass where p does not!

From this combination of effects, we can conclude that

$$\text{support}(q) \subseteq \text{support}(p) \implies D_{KL}\left[q \,||\, p\right] < \infty,$$

while

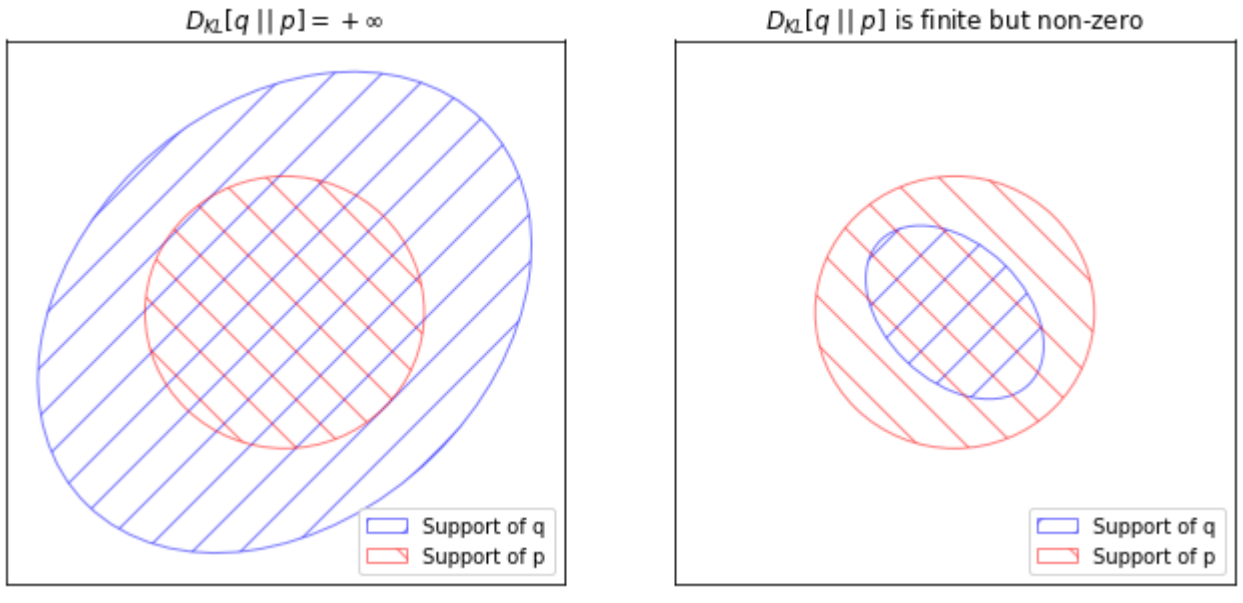
$$\text{support}(p) \subset \text{support}(q) \implies D_{KL}\left[q \,||\, p\right] = \infty$$

Consequently, the KL divergence favours distributions q that have a support contained in the target distribution's (i.e. p's).

The diagram below illustrates how the KL divergence is affected by the support of two bivariate density functions q and p. The hatched regions indicate the support of either function.

In [2]:
```
_, axs = plt.subplots(1, 2, sharex = True, sharey = True, figsize = (11, 5))

delta = 45.0  # degrees

q_ell_inf = Ellipse (
    (0, 0), 2, 1.5, 45, ec = 'blue', fc = 'none',
    alpha = 0.5, label = 'q(x)', hatch = '/'
)
q_ell_fin = Ellipse (
    (0, 0), 0.5, 0.75, 45, ec = 'blue', fc = 'none',
    alpha = 0.5, label = 'q(x)', hatch = '/'
)
p_ell_inf = Ellipse (
    (0, 0), 1, 1, 45, ec = 'red', fc = 'none',
    alpha = 0.5, label = 'p(x)', hatch = '\\'
)
p_ell_fin = Ellipse (
    (0, 0), 1, 1, 45, ec = 'red', fc = 'none',
    alpha = 0.5, label = 'p(x)', hatch = '\\'
)

# KL divergence is infinite
for ell in [q_ell_inf, p_ell_inf]:
    axs[0].add_artist(ell)
axs[0].legend([q_ell_inf, p_ell_inf], ['Support of q', 'Support of p'], loc = 'lower right')
axs[0].get_xaxis().set_ticks([])
axs[0].get_yaxis().set_ticks([])

# KL divergence is finite
for ell in [q_ell_fin, p_ell_fin]:
    axs[1].add_artist(ell)
axs[1].legend([q_ell_fin, p_ell_fin], ['Support of q', 'Support of p'], loc = 'lower right')
axs[1].get_xaxis().set_ticks([])
axs[1].get_yaxis().set_ticks([])

axs[0].set_title(r'$D_{KL}[q \ || \ p] = +\infty$')
axs[1].set_title(r'$D_{KL}[q \ || \ p]$ is finite but non-zero')
plt.xlim(-1, 1)
plt.ylim(-1, 1)
```

Out[2]: (-1.0, 1.0)



Computing KL divergence in TensorFlow

For some choices of q and p, the KL divergence can be evaluated to a closed-form expression.

`tfd.kl_divergence` computes the KL divergence between two distributions analytically, provided the divergence in question has been implemented in the TensorFlow Probability library.

Below is an example that uses `tfd.kl_divergence` to compute $D_{KL}\left[q \,||\, p\right]$ when q and p are univariate normal distributions.

In [3]:
```
# Simple example
mu_q = 0.
sigma_q = 1.
mu_p = 0.
sigma_p = 0.5
distribution_q = tfd.Normal(loc = mu_q, scale = sigma_q)
distribution_p = tfd.Normal(loc = mu_p, scale = sigma_p)

tfd.kl_divergence(distribution_q, distribution_p) # D_{KL}[q || p]
```

Out[3]: `<tf.Tensor: shape=(), dtype=float32, numpy=0.8068528>`

Let's check this value. The KL divergence between two univariate normal distributions can be derived directly from the definition of the KL divergence as

$$D_{KL}\left[q \,||\, p\right] = \frac{1}{2}\left(\frac{\sigma_q^2}{\sigma_p^2} + \frac{(\mu_q - \mu_p)^2}{\sigma_p^2} + 2\log\frac{\sigma_p}{\sigma_q} - 1\right)$$

The value of this function should be equal to that returned by `kl_divergence(distribution_q, distribution_p)`.

In [4]:
```
# Analytical expression for KL divergence between two univariate Normals
0.5 * ((sigma_q / sigma_p) ** 2 + ((mu_q - mu_p) / sigma_p) ** 2 + 2 * np.log(sigma_p / sigma_q) - 1)
```

Out[4]:
```
0.8068528194400546
```

Sure enough, it is.

If a batch of distributions is passed to `kl_divergence`, then a batch of divergences will be returned. `kl_divergence` also supports broadcasting.

In [5]:
```
# Batch example with broadcasting
distributions_q = tfd.Normal(loc = [0., 1.], scale = 1.)
distribution_p = tfd.Normal(loc = 0., scale = 0.5)
```

In [6]:
```
# Notice the batch_shape
distributions_q
```

Out[6]:
```
<tfp.distributions.Normal 'Normal' batch_shape=[2] event_shape=[] dtype=float32>
```

In [7]:
```
# [D_{KL}[q_1 || p], D_{KL}[q_2 || p]
tfd.kl_divergence(distributions_q, distribution_p)
```

Out[7]:
```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([0.8068528, 2.8068528], dtype=float32)>
```

`kl_divergence` provides a convenient way of computing the KL divergence for many TensorFlow distributions. As a rule of thumb, it will evaluate successfully provided you pass in two distributions of the same parametric family.

In [8]:
```
# An example with another distribution
beta_q = tfd.Beta(concentration1 = 12, concentration0 = 3)
beta_p = tfd.Beta(concentration1 = 9, concentration0 = 3)
tfd.kl_divergence(beta_q, beta_p)
```

Out[8]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=0.09615421>
```

In [9]:
```
# An example with a multivariate distribution
cov_q = np.array([[1., 0.5], [0.5, 1.]])
cov_p = np.array([[1., 0.], [0., 1.]])
mvtnormal_q = tfd.MultivariateNormalTriL(loc = [0., 0.], scale_tril = tf.linalg.cholesky(cov_q))
mvtnormal_p = tfd.MultivariateNormalTriL(loc = [0., 0.], scale_tril = tf.linalg.cholesky(cov_p))
tfd.kl_divergence(mvtnormal_q, mvtnormal_p)
```

Out[9]:
```
<tf.Tensor: shape=(), dtype=float64, numpy=0.14384103622589053>
```

To see a complete list of distributions for which a KL method is defined, refer to `help(tfd.kl_divergence)`.

If you pass `kl_divergence` a pair distributions for which a KL divergence method is not implemented, an error will be raised:

In [10]:
```
# uniform_q and beta_p are both uniform distributions with support [0, 1]
uniform_q = tfd.Uniform(low = 0., high = 1.)
beta_p = tfd.Beta(concentration1 = 0., concentration0 = 0.)
```

In [11]:
```
# kl_divergence has no method for computing their divergence
try:
    tfd.kl_divergence(uniform_q, beta_p)
except Exception as e:
    print(e)
```

```
No KL(distribution_a || distribution_b) registered for distribution_a type Uniform and distribution_b type Beta
```

When `kl_divergence` fails

If you do not have a closed-form expression for your KL divergence, and it is not implemented in `tfd.kl_divergence`, then you can make a Monte Carlo estimate of it. Simply sample n values $x_1, \ldots, x_n$ from q, then evaluate the estimate

$$\frac{1}{n}\sum_{i=1}^{n} \log\left[q(x_i)\right] - \log\left[p(x_i)\right]$$

In general, the Monte Carlo estimator is unbiased and its variance is inversely proportional to n.

To show how the variance of the Monte Carlo estimator varies with n, let's attempt to estimate $D_{KL}\left[q \mid p\right]$ when q and p are univariate normal distributions. We'll make many estimates for several values of n, then plot their absolute error as a function of n.

We'll start by evaluating the exact value $D_{KL}\left[q \mid p\right]$ using `kl_divergence`. Bear in mind that the Monte Carlo estimate will only be useful in situations where this not possible!

In [12]:
```
# Evaluate the exact KL divergence

distribution_q = tfd.Normal(loc=0., scale=1.)
distribution_p = tfd.Normal(loc=0., scale=0.5)

exact_kl_divergence = tfd.kl_divergence(distribution_q, distribution_p).numpy() # D_{KL}[q || p]
exact_kl_divergence
```

Out[12]:
```
0.8068528
```

Next, we'll define a function for making a Monte Carlo estimate for a given q, p, and n.

In [13]:
```
# Function to estimate the KL divergence with Monte Carlo samples
def monte_carlo_estimate_of_kl_divergence(n, q_sampler, q_density, p_density):
    '''
        Computes a Monte Carlo estimate of D_{KL}[q || p] using
        n samples from q_sampler.

        q_sampler is a function that receives a positive integer
        and returns as many samples from q.

        Given samples x_1, ..., x_n from q_sampler, the Monte Carlo
        estimate is

            \frac{1}{n}\sum_{i=1}^n \log(q(x_i)) - \log(p(x_i))

        where q and p are density/mass functions.
    '''
    x = q_sampler(n)
    KL_estimate = np.mean(np.log(q_density(x)) - np.log(p_density(x)))
    return(KL_estimate)
```

The code below shows how this function can be used to make a single estimate.

In [14]:
```
# Single MC estimate
n = 1000 # number of samples used in MC estimate
q_sampler = distribution_q.sample
q_density = distribution_q.prob
p_density = distribution_p.prob

monte_carlo_estimate_of_kl_divergence(n, q_sampler, q_density, p_density)
```

Out[14]:
```
0.8441007
```

To see how the estimator's variance decreases with increasing n, let's evaluate a few hundred estimates for each point in a grid of n values.

In [15]:
```
# Create a grid of 8 points
n_grid = 10 ** np.arange(1, 8)
samples_per_grid_point = 100  # Number of MC estimates to make for each value of n
```

In [16]:
```
# Array to store results
kl_estimates = np.zeros(shape = [samples_per_grid_point, len(n_grid), 2])
```

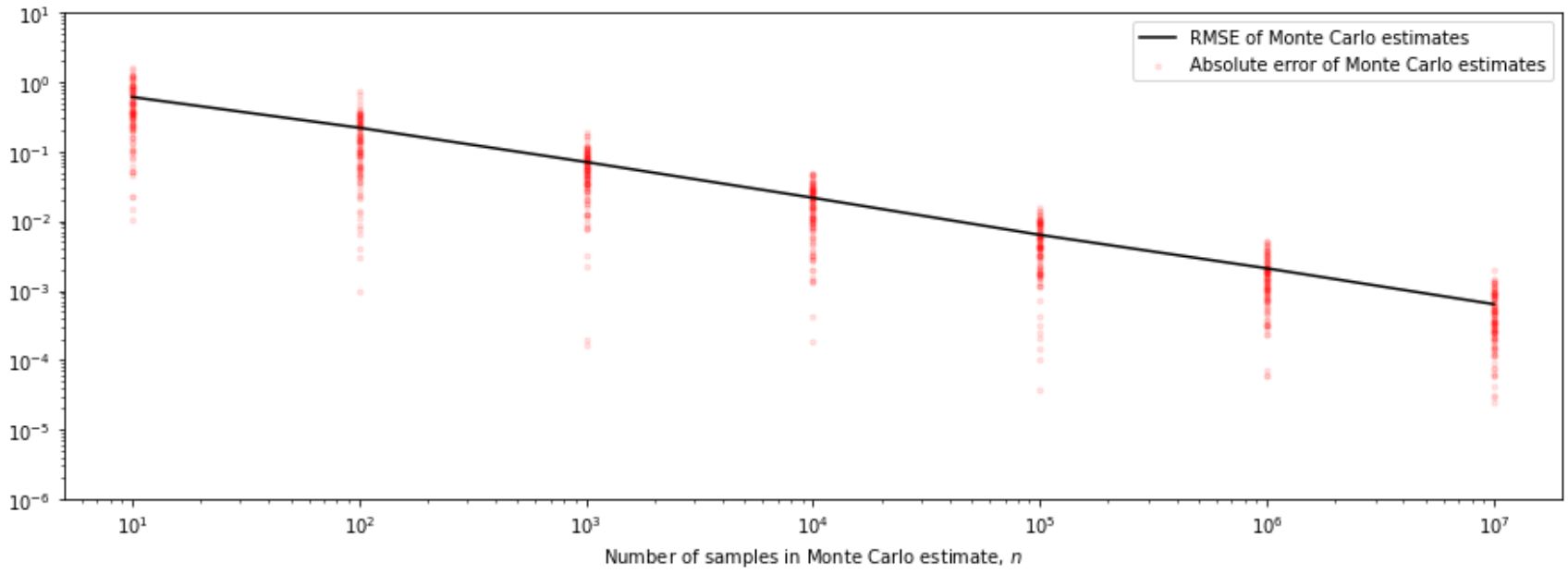
In [17]:
```
# Make 100 MC estimates for each value of n, store the results in kl_estimates
for sample_num in range(samples_per_grid_point):
    for grid_num, n in enumerate(n_grid):
        kl_estimates[sample_num, grid_num, 0] = n
        kl_estimates[sample_num, grid_num, 1] = \
            monte_carlo_estimate_of_kl_divergence(n, q_sampler, q_density, p_density)
```

In [18]:
```
# Compute RMSE of estimates (this is approximately equal to the standard deviation of the MC estimator)
rmse_of_kl_estimates = np.sqrt \
    (np.mean((kl_estimates[:, :, 1] - exact_kl_divergence) ** 2, axis = 0))
```

In [19]:
```
# Compute absolute error of the MC estimates
abs_error_of_kl_estimates = abs(kl_estimates[:, :, 1].flatten() - exact_kl_divergence)
```

In [20]:
```
# Plot the results
_, ax = plt.subplots(1, 1, figsize = (15, 5))
plt.xlabel(r'Number of samples in Monte Carlo estimate, $n$')
ax.scatter (
    kl_estimates[:, :, 0],
    abs_error_of_kl_estimates,
    marker = '.', color = 'red',
    alpha = 0.1, label = 'Absolute error of Monte Carlo estimates'
)
ax.plot(n_grid, rmse_of_kl_estimates, color = 'k', label = 'RMSE of Monte Carlo estimates')
ax.set_xscale('log')
ax.set_yscale('log')
ax.set_ylim([1e-6, 10])
ax.legend()
```

Out[20]:
```
<matplotlib.legend.Legend at 0x7fed1c260d10>
```



You can see that the gradient of the estimates' RMSE, an estimate of the MC estimator's standard devation, with respect to n is $-\frac{1}{2}$. This is unsurprising: the estimator's variance is inversely proportional to n, so its log standard deviation is a linear function of $\log n$ with gradient $-\frac{1}{2}$. As n increases, the Monte Carlo estimates approach the exact value of the KL divergence.

Summary

You should now feel confident about how the Kullback-Leibler divergence is motivated and defined, what its key properties and why it is used in variational inference, and how it can be computed or estimated in TensorFlow.

Further reading and resources

- TensorFlow documentation on `tfd.kl_divergence`: https://www.tensorflow.org/probability/api_docs/python/tfp/distributions/kl_divergence (https://www.tensorflow.org/probability/api_docs/python/tfp/distributions/kl_divergence)

Appendix

Information gain, relative entropy, and Bayesian inference

This section provides further context for the Kullback-Leibler divergence. It is not essential, but it will give you a more complete understanding of what the divergence measures.

The Kullback-Leibler divergence has its origins in information theory. The Shannon entropy, defined as

$$H(P) := E_{X \sim P(x)}[-\log P(X)]$$

is the greatest lower bound on the average number of nats ($\log 2$ nats are equal to 1 bit) required to losslessly encode an observation sampled from from $P(x)$. This is an informal statement of a result known as the *source coding theorem*. $-\log P(x)$ is the number of bits used to encode x in the lossless encoding scheme.

Say that a lossless compression algorithm instead encodes observations using a scheme that would be optimal for distribution $Q(x)$. Then the average number of of bits required to encode an observation sampled from $P(x)$ would be

$$H(P, Q) := E_{X \sim P(x)}[-\log Q(X)]$$

This is quantity is referred to as the *cross-entropy* between P and Q. Since $H(P)$ is the minimum average information for encoding observations from $P(x)$ by definition, it follows that $H(P, Q) \geq H(P)$.

The Kullback-Leibler divergence is defined as the average additional information required to encode observations from $P(x)$ using an optimal code for $Q(x)$:

$$\begin{aligned} D_{KL}(P \,||\, Q) &:= E_{X \sim P(x)}[-\log Q(X)] - E_{X \sim P(x)}[-\log P(X)] \\ &= H(P, Q) - H(P) \end{aligned}$$

The KL divergence therefore tells us how inefficient the optimal coding scheme for Q is when applied to data source P.

That KL divergence is the difference between a cross-entropy and a Shannon entropy sheds light on why the KL divergence has another moniker - *relative entropy*.

Alternatively, we might consider encoding observations in the context of Bayesian inference. Let $P(y)$ be the prior and $P(y|x)$ be the posterior. Then the Kullback-Leibler divergence

$$D_{KL}(P(y|x) \,||\, P(y)) = E_{Y \sim P(y|x)}[-\log P(Y)] - E_{Y \sim P(y|x)}[-\log P(Y|x)]$$

is the average number of bits saved if observations are encoded using an optimal code for the posterior rather than the prior. In this sense, the KL divergence tells us how much information is gained by conditioning on X.

Full covariance Gaussian approximation

This reading follows on from the previous coding tutorial, and shows how to set up a full covariance variational approximating Gaussian distribution.

In [1]:
```
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions
tfb = tfp.bijectors
print("TF version:", tf.__version__)
print("TFP version:", tfp.__version__)

# Additional packages for this reading
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import clear_output
```

```
TF version: 2.3.0
TFP version: 0.11.0
```

In the previous tutorial, you approximated a full covariance Gaussian with a diagonal covariance Gaussian. This reading notebook is a supplement that shows how you can configure a full covariance Gaussian as the approximating distribution.

The code below initializes the same target distribution that you saw in the tutorial.

In [2]:
```
# Define the target distribution
tf.random.set_seed(41)

p_mu = [0., 0.]
p_Sigma = tfp.bijectors.Chain \
    ([tfb.TransformDiagonal(tfb.Softplus()), tfb.FillTriangular()])(tf.random.uniform([3]))
p = tfd.MultivariateNormalTriL(loc = p_mu, scale_tril = p_Sigma)
```

To create a trainable normal distribution with full covariance matrix, we use the `MultivariateNormalTriL` Distribution.

In [3]:
```
# Define the approximating distribution
scale_tril_init = tfb.FillScaleTriL()(tf.random.normal([3]))
q = tfd.MultivariateNormalTriL (
    loc = tf.Variable(tf.random.normal([2])),
    scale_tril = tfp.util.TransformedVariable(scale_tril_init, bijector = tfb.FillScaleTriL())
)
```

As with `MultivariateNormalDiag`, `loc` is specified as a randomly intialized `tf.Variable`.

The lower-triangular matrix `scale_tril`, on the other hand, is initialized as

```
tfp.util.TransformedVariable(scale_tril_init,
                             bijector=tfb.FillScaleTriL())
```

Let's unpack this bit by bit. `tfp.util.TransformedVariable` is a class that allows us to initialize a Variable using a value for its bijection. Parameter updates take place on the unconstrained Variable, while the bijection enforces a constraint (e.g. positivity, shape, etc.).

In this case, we initialize `scale_tril` using a lower-triangular matrix, `scale_tril_init`.

The bijector handed to `TransformedVariable` is `tfb.FillScaleTriL`. This bijector is equivalent to a `tfb.Chain` of `tfb.FillTriangular` followed by `tfb.TransformDiagonal`.

`tfb.FillTriangular` inserts the elements of a vector into a lower-triangular matrix in a clockwise spiral. `tfb.TransformDiagonal` then applies a bijection to the diagonal of this matrix.

The diagonal bijection applied by `tfb.FillScaleTriL` can be specified via the `diag_bijector` argument. By default, it is a bijector chain of `tfb.Softplus` followed by addition of `1e-5`.

If you refer back to how `p_Sigma` is declared in the code cell above, you can see that it is initialized using a bijector chain that is similar to `tfb.FillScaleTriL` 's.

Fit the approximating distribution by minimising KL divergence

The target and trainable distributions have been initialized. All that remains is to fit the trainable distribution to the target.

The code below is identical to what you saw for fitting the diagonal covariance example in the coding tutorial.

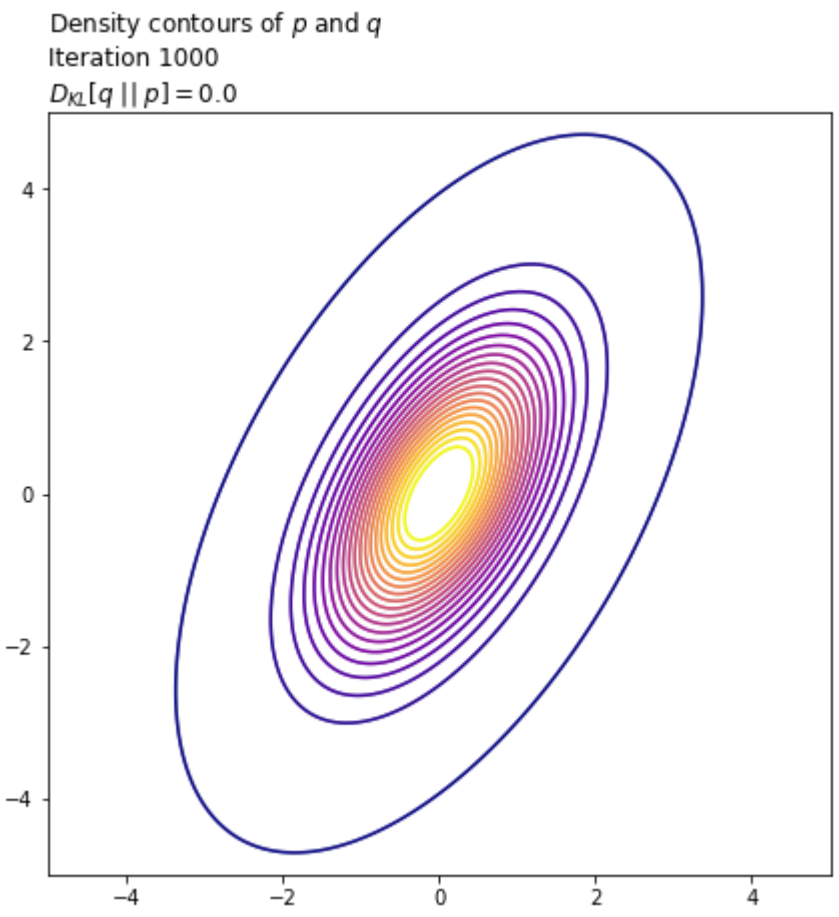
In [4]:
```
@tf.function
def loss_and_grads(dist_a, dist_b):
    '''
        Returns D_{KL}[dist_a || dist_b] and the gradients of this
        with respect to the trainable Variables of dist_a.
    '''
    with tf.GradientTape() as tape:
        loss = tfd.kl_divergence(dist_a, dist_b)
    return loss, tape.gradient(loss, dist_a.trainable_variables)
```

In [5]:
```
# Define function for graphics

def plot_density_contours(density, X1, X2, contour_kwargs, ax = None):
    '''
        Plots the contours of a bivariate TensorFlow density function (i.e. .prob()).
        X1 and X2 are numpy arrays of mesh coordinates.
    '''
    X = np.hstack([X1.flatten()[:, np.newaxis], X2.flatten()[:, np.newaxis]])
    density_values = np.reshape(density(X).numpy(), newshape = X1.shape)

    if ax == None:
```

```
        _, ax = plt.subplots(figsize = (7, 7))

    ax.contour(X1, X2, density_values, **contour_kwargs)
    return(ax)

x1 = np.linspace(-5, 5, 1000)
x2 = np.linspace(-5, 5, 1000)
X1, X2 = np.meshgrid(x1, x2)
contour_levels = np.linspace(1e-4, 10 ** (-0.8), 20)
```

In [6]:

```
# Set up and run a custom training loop to minimise the KL loss

num_train_steps = 1000
opt = tf.keras.optimizers.Adam(learning_rate = .01)
for i in range(num_train_steps):

    # Compute the KL divergence and its gradients
    q_loss, grads = loss_and_grads(q, p)

    # Update the trainable variables using the gradients via the optimizer
    opt.apply_gradients(zip(grads, q.trainable_variables))

    # Plot the updated density
    if ((i + 1) % 10 == 0):
        clear_output(wait = True)
        ax = plot_density_contours \
            (p.prob, X1, X2, {'levels': contour_levels, 'cmap': 'cividis', 'alpha': 0.5})
        ax = plot_density_contours \
            (q.prob, X1, X2, {'levels': contour_levels, 'cmap': 'plasma'}, ax = ax)
        ax.set_title (
            'Density contours of $p$ and $q$\n' +
            'Iteration ' + str(i + 1) + '\n' +
            '$D_{KL}[q \ || \ p] = ' +
            str(np.round(q_loss.numpy(), 4)) + '$',
        loc = 'left'
    )
    plt.pause(.01)
```



As you can see, using a trainable distribution that is in the same parametric family as the target enables the KL divergence to be minimised to zero, indicating that the target distribution has been learnt perfectly.

Summary

This notebook showed how you can initialize and train a normal distribution with full covariance matrix. `MultivariateNormalTriL` with a Variable transformed via `FillScaleTriL` should be your go-to for learnt full covariance matrices.

Variational Autoencoders

In [1]:

```
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions
tfpl = tfp.layers
tfb = tfp.bijectors
print("TF version:", tf.__version__)
print("TFP version:", tfp.__version__)
```

```
TF version: 2.3.0
TFP version: 0.11.0
```

1. Encoders and decoders

2. Minimising Kullback-Leibler divergence

3. Maximising the ELBO

4. KL divergence layers

Encoders and decoders

In [2]:
```
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import Dense, Flatten, Reshape
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
```

In [3]:
```
# Load Fashion MNIST
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.fashion_mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
class_names = np.array ([
    'T-shirt/top', 'Trouser/pants', 'Pullover shirt', 'Dress',
    'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag','Ankle boot'
])
```

In [4]:
```
# Display a few examples
n_examples = 1000
example_images = x_test[0:n_examples]
example_labels = y_test[0:n_examples]

f, axs = plt.subplots(1, 5, figsize = (15, 4))
for j in range(len(axs)):
    axs[j].imshow(example_images[j], cmap = 'binary')
    axs[j].axis('off')
```

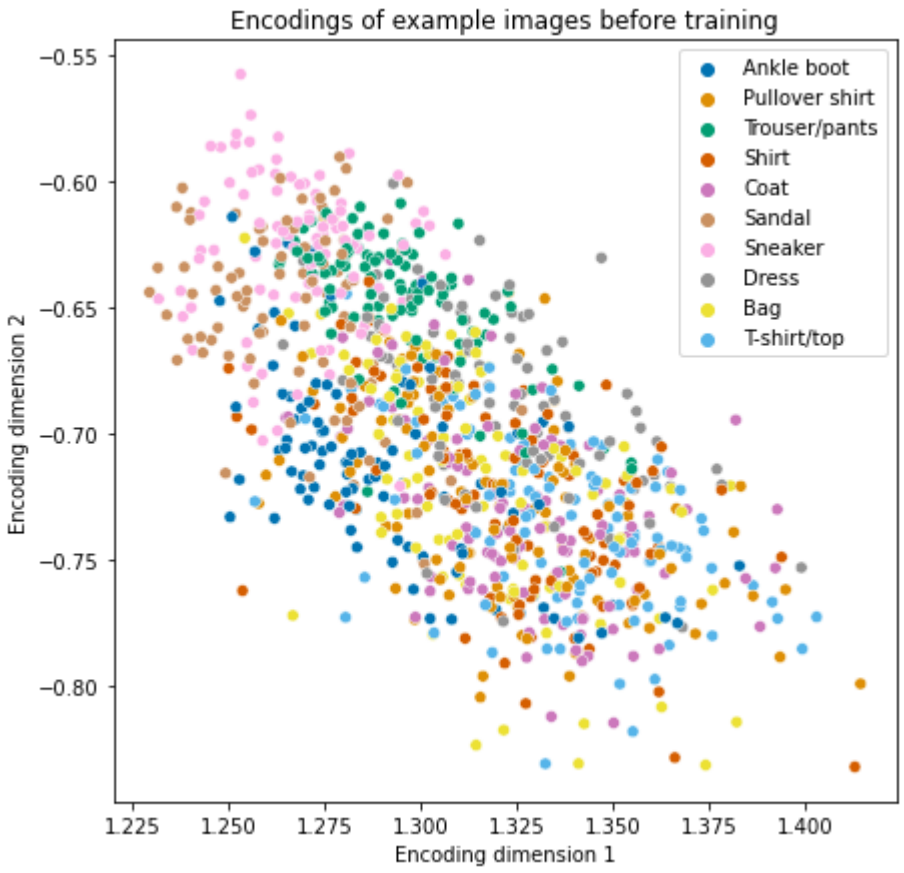
In [5]:
```
# Define the encoder
encoded_dim = 2
encoder = Sequential ([
    Flatten(input_shape = (28, 28)),
    Dense(256, activation = 'sigmoid'),
    Dense(64, activation = 'sigmoid'),
    Dense(encoded_dim)
])
```

In [6]:
```
# Encode examples before training
pretrain_example_encodings = encoder(example_images).numpy()
```

In [7]:
```
# Plot encoded examples before training
f, ax = plt.subplots(1, 1, figsize = (7, 7))
sns.scatterplot (
    pretrain_example_encodings[:, 0],
    pretrain_example_encodings[:, 1],
    hue = class_names[example_labels], ax = ax,
    palette = sns.color_palette("colorblind", 10)
)
ax.set_xlabel('Encoding dimension 1')
ax.set_ylabel('Encoding dimension 2')
ax.set_title('Encodings of example images before training')
```

```
/home/bacti/anaconda3/envs/tensor/lib/python3.7/site-packages/seaborn/_decorators.py:43: FutureWarning: Pass the follo
wing variables as keyword args: x, y. From version 0.12, the only valid positional argument will be `data`, and passin
g other arguments without an explicit keyword will result in an error or misinterpretation.
  FutureWarning
```

Out[7]: Text(0.5, 1.0, 'Encodings of example images before training')



In [8]:
```
# Define the decoder
decoder = Sequential ([
    Dense(64, activation = 'sigmoid', input_shape = (encoded_dim,)),
    Dense(256, activation = 'sigmoid'),
    Dense(28 * 28, activation = 'sigmoid'),
    Reshape((28, 28))
])
```

In [9]:
```
# Compile and fit the model
autoencoder = Model(inputs = encoder.inputs, outputs = decoder(encoder.outputs))

# Specify loss - input and output is in [0., 1.], so we can use a binary cross-entropy loss
autoencoder.compile(loss = 'binary_crossentropy')

# Fit model - highlight that labels and input are the same
autoencoder.fit(x = x_train, y = x_train, epochs = 10, batch_size = 32)
```

```
Epoch 1/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.3780
Epoch 2/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.3425
Epoch 3/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.3350
Epoch 4/10
1875/1875 [==============================] - 8s 5ms/step - loss: 0.3305
Epoch 5/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.3276
Epoch 6/10
1875/1875 [==============================] - 8s 5ms/step - loss: 0.3258
Epoch 7/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.3243
Epoch 8/10
1875/1875 [==============================] - 8s 5ms/step - loss: 0.3233
Epoch 9/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.3223
Epoch 10/10
1875/1875 [==============================] - 8s 5ms/step - loss: 0.3215
```

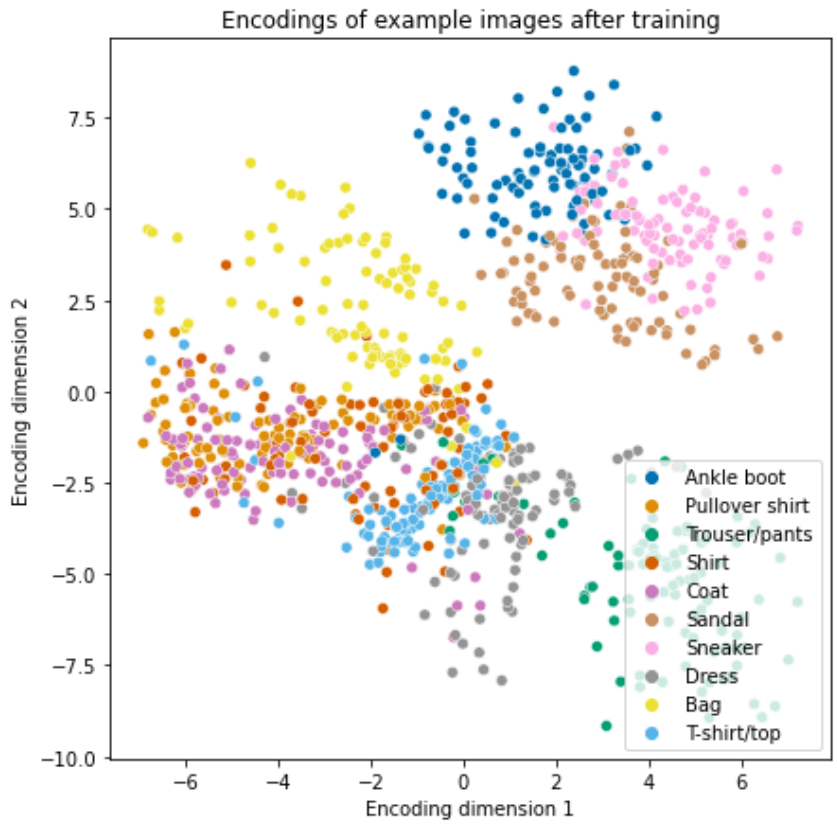
Out[9]:
```
<tensorflow.python.keras.callbacks.History at 0x7f867408a910>
```

In [10]:
```
# Compute example encodings after training
posttrain_example_encodings = encoder(example_images).numpy()
```

In [11]:
```
# Compare the example encodings before and after training
f, axs = plt.subplots(nrows = 1, ncols = 2, figsize = (15, 7))
sns.scatterplot (
    pretrain_example_encodings[:, 0],
    pretrain_example_encodings[:, 1],
    hue = class_names[example_labels], ax = axs[0],
    palette = sns.color_palette("colorblind", 10)
)
sns.scatterplot (
    posttrain_example_encodings[:, 0],
    posttrain_example_encodings[:, 1],
    hue = class_names[example_labels], ax = axs[1],
    palette = sns.color_palette("colorblind", 10)
)

axs[0].set_title('Encodings of example images before training')
axs[1].set_title('Encodings of example images after training')

for ax in axs:
    ax.set_xlabel('Encoding dimension 1')
    ax.set_ylabel('Encoding dimension 2')
    ax.legend(loc = 'lower right')
```

```
/home/bacti/anaconda3/envs/tensor/lib/python3.7/site-packages/seaborn/_decorators.py:43: FutureWarning: Pass the follo
wing variables as keyword args: x, y. From version 0.12, the only valid positional argument will be `data`, and passin
```

```
g other arguments without an explicit keyword will result in an error or misinterpretation.
  FutureWarning
/home/bacti/anaconda3/envs/tensor/lib/python3.7/site-packages/seaborn/_decorators.py:43: FutureWarning: Pass the follo
wing variables as keyword args: x, y. From version 0.12, the only valid positional argument will be `data`, and passin
g other arguments without an explicit keyword will result in an error or misinterpretation.
  FutureWarning
```



In [12]:
```
# Compute the autoencoder's reconstructions
reconstructed_example_images = autoencoder(example_images)
```

In [13]:
```
# Evaluate the autoencoder's reconstructions
f, axs = plt.subplots(2, 5, figsize = (15, 4))
for j in range(5):
    axs[0, j].imshow(example_images[j], cmap = 'binary')
    axs[1, j].imshow(reconstructed_example_images[j].numpy().squeeze(), cmap = 'binary')
    axs[0, j].axis('off')
    axs[1, j].axis('off')
```

Minimising Kullback-Leibler divergence

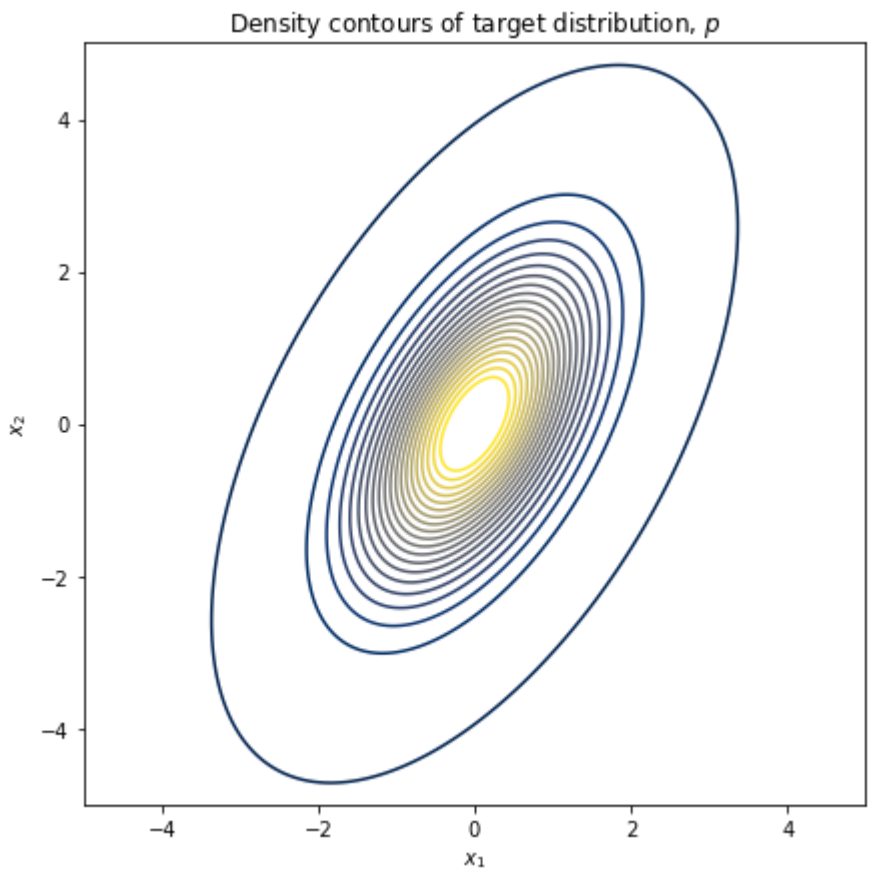
In [14]:
```
import matplotlib.pyplot as plt
import numpy as np
from IPython.display import clear_output
```

In [15]:
```
# Define a target distribution, p
tf.random.set_seed(41)
p_mu = [0., 0.]
p_L = tfb.Chain([tfb.TransformDiagonal(tfb.Softplus()), tfb.FillTriangular()]) \
    (tf.random.uniform([3]))
p = tfd.MultivariateNormalTriL(loc = p_mu, scale_tril = p_L)
```

In [16]:
```
# Plot the target distribution's density contours
def plot_density_contours(density, X1, X2, contour_kwargs, ax = None):
    '''
        Plots the contours of a bivariate TensorFlow density function (i.e. .prob()).
        X1 and X2 are numpy arrays of mesh coordinates.
    '''
    X = np.hstack([X1.flatten()[:, np.newaxis], X2.flatten()[:, np.newaxis]])
    density_values = np.reshape(density(X).numpy(), newshape = X1.shape)

    if ax == None:
        _, ax = plt.subplots(figsize = (7, 7))

    ax.contour(X1, X2, density_values, **contour_kwargs)
    return(ax)

x1 = np.linspace(-5, 5, 1000)
x2 = np.linspace(-5, 5, 1000)
X1, X2 = np.meshgrid(x1, x2)
f, ax = plt.subplots(1, 1, figsize = (7, 7))
```

```
# Density contours are linearly spaced
contour_levels = np.linspace(1e-4, 10 ** -0.8, 20) # specific to this seed
ax = plot_density_contours \
    (p.prob, X1, X2, {'levels': contour_levels,  'cmap': 'cividis'}, ax = ax)
ax.set_xlim(-5, 5)
ax.set_ylim(-5, 5)
ax.set_title('Density contours of target distribution, $p$')
ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
```

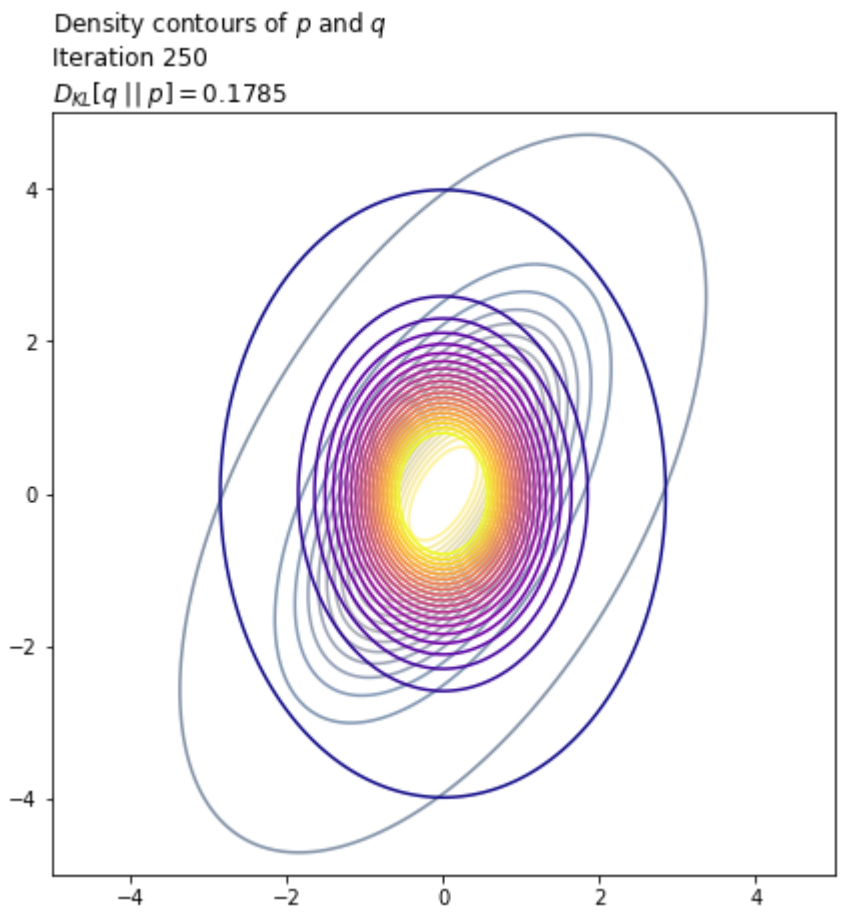
Out[16]: Text(0, 0.5, 'x_2')



In [17]:
```
# Initialize an approximating distribution, q, that has diagonal covariance
tf.random.set_seed(41)
q = tfd.MultivariateNormalDiag (
    loc = tf.Variable(tf.random.normal([2])),
    scale_diag = tfp.util.TransformedVariable(tf.random.uniform([2]), bijector = tfb.Exp())
)
```

In [18]:
```
# Define a function for the Kullback-Leibler divergence
@tf.function
def loss_and_grads(dist_a, dist_b, reverse = False):
    with tf.GradientTape() as tape:
        if not reverse:
            loss = tfd.kl_divergence(dist_a, dist_b)
        else:
            loss = tfd.kl_divergence(dist_b, dist_a)
    return loss, tape.gradient(loss, dist_a.trainable_variables)
```

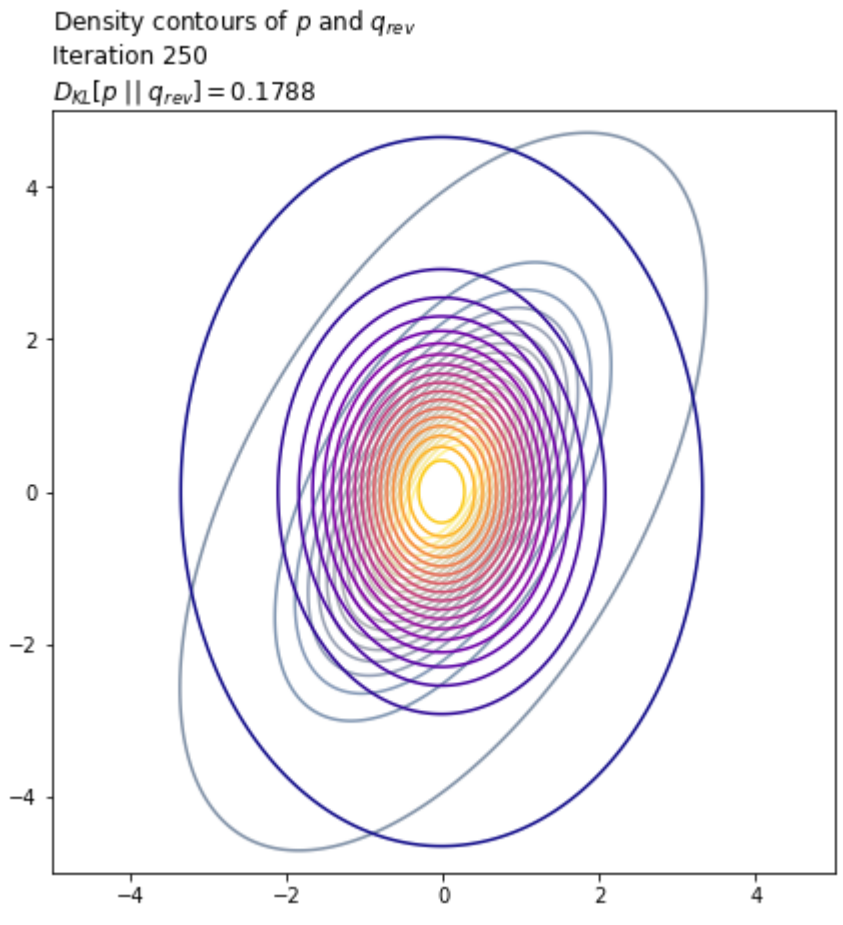
In [19]:
```
# Run a training loop that computes KL[q || p], updates q's parameters using its gradients
num_train_steps = 250
opt = tf.keras.optimizers.Adam(learning_rate = .01)

for i in range(num_train_steps):

    # Compute the KL divergence and its gradients
    q_loss, grads = loss_and_grads(q, p)

    # Update the trainable variables using the gradients via the optimizer
    opt.apply_gradients(zip(grads, q.trainable_variables))

    # Plot the updated density
    if ((i + 1) % 10 == 0):
        clear_output(wait = True)
        ax = plot_density_contours \
            (p.prob, X1, X2, {'levels': contour_levels, 'cmap': 'cividis', 'alpha': 0.5})
        ax = plot_density_contours \
            (q.prob, X1, X2, {'levels': contour_levels, 'cmap': 'plasma'}, ax = ax)
        ax.set_title (
            'Density contours of $p$ and $q$\n' +
                'Iteration ' + str(i + 1) + '\n' +
                '$D_{KL}[q \ || \ p] = ' +
                str(np.round(q_loss.numpy(), 4)) + '$'
            ,
            loc = 'left'
        )
        plt.pause(.01)
```



In [20]:
```
# Re-fit the distribution, this time fitting q_rev by minimising KL[p || q_rev]
tf.random.set_seed(41)
q_rev = tfd.MultivariateNormalDiag (
    loc = tf.Variable(tf.random.normal([2])),
    scale_diag = tfp.util.TransformedVariable(tf.random.uniform([2]), bijector = tfb.Exp())
)
```

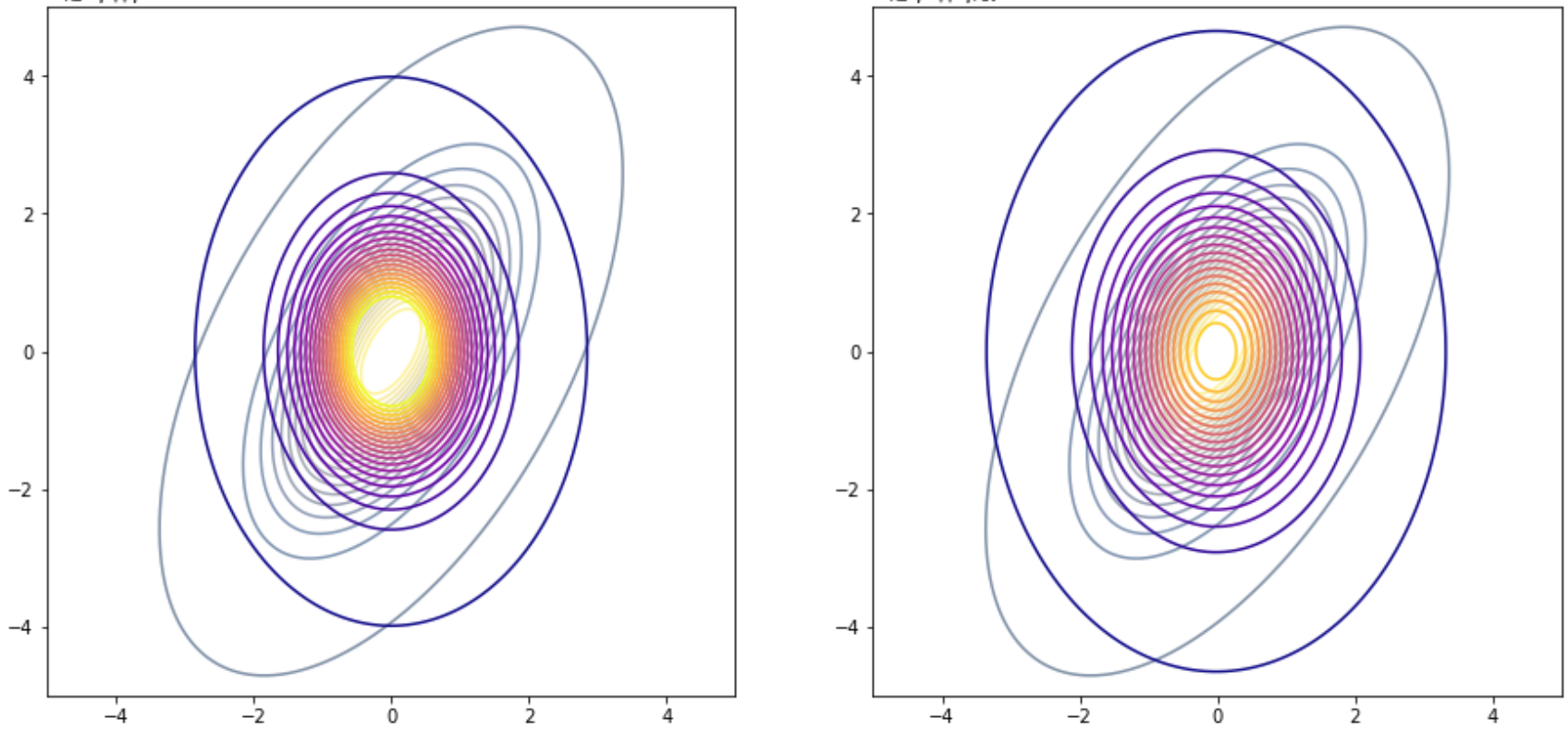
In [21]:
```
# Edit loss_and_grads function
```

In [22]:
```
# Re-initialize optimizer, run training loop
opt = tf.keras.optimizers.Adam(learning_rate = .01)
for i in range(num_train_steps):
    # Reverse the KL divergence terms - compute KL[p || q_rev]
    q_rev_loss, grads = loss_and_grads(q_rev, p, reverse = True)

    # Update the trainable variables using the gradients via the optimizer
    opt.apply_gradients(zip(grads, q_rev.trainable_variables))

    # Plot the updated density
    if ((i + 1) % 10 == 0):
        clear_output(wait=True)
        ax = plot_density_contours \
            (p.prob, X1, X2, {'levels': contour_levels, 'cmap': 'cividis', 'alpha': 0.5})
        ax = plot_density_contours \
            (q_rev.prob, X1, X2, {'levels': contour_levels, 'cmap': 'plasma'}, ax = ax)
        ax.set_title (
            'Density contours of $p$ and $q_{rev}$\n' +
                'Iteration ' + str(i + 1) + '\n' +
                '$D_{KL}[p \ || \ q_{rev}] = ' +
                str(np.round(q_rev_loss.numpy(), 4)) + '$'
            ,
            loc = 'left'
        )
        plt.pause(.01)
```



In [23]:
```
# Plot q and q_rev alongside one another
f, axs = plt.subplots(1, 2, figsize = (15, 7))

axs[0] = plot_density_contours \
    (p.prob, X1, X2, {'levels': contour_levels, 'cmap': 'cividis', 'alpha': 0.5}, ax = axs[0])
axs[0] = plot_density_contours \
    (q.prob, X1, X2, {'levels': contour_levels, 'cmap': 'plasma'}, ax = axs[0])
axs[0].set_title (
    'Density contours of $p$ and $q$\n' +
        '$D_{KL}[q \ || \ p] = ' + str(np.round(q_loss.numpy(), 4)) + '$'
    ,
    loc = 'left'
)

axs[1] = plot_density_contours \
    (p.prob, X1, X2, {'levels': contour_levels, 'cmap': 'cividis', 'alpha': 0.5}, ax = axs[1])
axs[1] = plot_density_contours \
    (q_rev.prob, X1, X2, {'levels': contour_levels, 'cmap': 'plasma'}, ax = axs[1])
axs[1].set_title (
    'Density contours of $p$ and $q_{rev}$\n' +
        '$D_{KL}[p \ || \ q_{rev}] = ' + str(np.round(q_rev_loss.numpy(), 4)) + '$'
    ,
    loc = 'left'
)
```

Out[23]:
```
Text(0.0, 1.0, 'Density contours of $p$ and $q_{rev}$\n$D_{KL}[p \\ || \\ q_{rev}] = 0.1788$')
```



Maximising the ELBO

Review of terminology:

- $p(z)$ = prior
- $q(z \mid x)$ = encoding distribution
- $p(x \mid z)$ = decoding distribution

$$\begin{aligned}\log p(x) &\geq \mathrm{E}_{Z \sim q(z \mid x)} \left[-\log q(Z \mid x) + \log p(x, Z) \right] \\ &= -\mathrm{KL} \left[q(z \mid x) \,||\, p(z) \right] + \mathrm{E}_{Z \sim q(z \mid x)} \left[\log p(x \mid Z) \right]\end{aligned}$$

In [24]:
```
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import Dense, Flatten, Reshape
import matplotlib.pyplot as plt
import numpy as np
```

In [25]:
```
# Import Fasion MNIST, make it a TensorFlow Dataset
(x_train, _), (x_test, _) = tf.keras.datasets.fashion_mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
example_x = x_test[:16]

batch_size = 64
x_train = tf.data.Dataset.from_tensor_slices(x_train).batch(batch_size)
```

In [26]:
```
# Define the encoding distribution, q(z|x)
latent_size = 2
event_shape = (28, 28)
encoder = Sequential ([
    Flatten(input_shape = event_shape),
    Dense(256, activation = 'relu'),
    Dense(128, activation = 'relu'),
    Dense(64, activation = 'relu'),
    Dense(32, activation = 'relu'),
```

```
    Dense(2 * latent_size),
    tfpl.DistributionLambda (
        lambda t: tfd.MultivariateNormalDiag \
            (loc = t[..., :latent_size], scale_diag = tf.math.exp(t[..., latent_size:]))
    )
])
```

```
WARNING:tensorflow:From /home/bacti/anaconda3/envs/tensor/lib/python3.7/site-packages/tensorflow/python/ops/linalg/lin
ear_operator_diag.py:166: calling LinearOperator.__init__ (from tensorflow.python.ops.linalg.linear_operator) with gra
ph_parents is deprecated and will be removed in a future version.
Instructions for updating:
Do not pass `graph_parents`.  They will  no longer be used.
```

In [27]:
```
# Pass an example image through the network - should return a batch of MultivariateNormalDiags.
encoder(example_x)
```

Out[27]:
```
<tfp.distributions.MultivariateNormalDiag 'sequential_2_distribution_lambda_MultivariateNormalDiag' batch_shape=[16] e
vent_shape=[2] dtype=float32>
```

In [28]:
```
# Define the decoding distribution, p(x|z)
decoder = Sequential ([
    Dense(32, activation = 'relu'),
    Dense(64, activation = 'relu'),
    Dense(128, activation = 'relu'),
    Dense(256, activation = 'relu'),
    Dense(tfpl.IndependentBernoulli.params_size(event_shape)),
    tfpl.IndependentBernoulli(event_shape)
])
```

In [29]:
```
# Pass a batch of examples to the decoder
decoder(tf.random.normal([16, latent_size]))
```

Out[29]:
```
<tfp.distributions.Independent 'sequential_3_independent_bernoulli_IndependentBernoulli_Independentsequential_3_indepe
ndent_bernoulli_IndependentBernoulli_Bernoulli' batch_shape=[16] event_shape=[28, 28] dtype=float32>
```

In [30]:
```
# Define the prior, p(z) - a standard bivariate Gaussian
prior = tfd.MultivariateNormalDiag(loc = tf.zeros(latent_size))
```

The loss function we need to estimate is

$$-\text{ELBO} = \text{KL}[\, q(z|x) \,||\, p(z)\,] - \text{E}_{Z\sim q(z|x)}[\log p(x|Z)]$$

where $x = (x_1, x_2, \ldots, x_n)$ refers to all observations, $z = (z_1, z_2, \ldots, z_n)$ refers to corresponding latent variables.

Assumed independence of examples implies that we can write this as

$$\sum_j \text{KL}[\, q(z_j|x_j) \,||\, p(z_j)\,] - \text{E}_{Z_j\sim q(z_j|x_j)}[\log p(x_j|Z_j)]$$

In [31]:
```
# Specify the loss function, an estimate of the -ELBO
def loss(x, encoding_dist, sampled_decoding_dist, prior):
    return tf.reduce_sum(tfd.kl_divergence(encoding_dist, prior) - sampled_decoding_dist.log_prob(x))
```

In [32]:
```
# Define a function that returns the loss and its gradients
@tf.function
def get_loss_and_grads(x):
    with tf.GradientTape() as tape:
        encoding_dist = encoder(x)
        sampled_z = encoding_dist.sample()
        sampled_decoding_dist = decoder(sampled_z)
        current_loss = loss(x, encoding_dist, sampled_decoding_dist, prior)
    grads = tape.gradient(current_loss, encoder.trainable_variables + decoder.trainable_variables)
    return current_loss, grads
```

In [33]:
```
# Compile and train the model
num_epochs = 5
opt = tf.keras.optimizers.Adam()
for i in range(num_epochs):
    for train_batch in x_train:
        current_loss, grads = get_loss_and_grads(train_batch)
        opt.apply_gradients(zip(grads, encoder.trainable_variables + decoder.trainable_variables))

    print('-ELBO after epoch {}: {:.0f}'.format(i + 1, current_loss.numpy()))
```

```
-ELBO after epoch 1: 9014
-ELBO after epoch 2: 8835
-ELBO after epoch 3: 8832
-ELBO after epoch 4: 8787
-ELBO after epoch 5: 8762
```

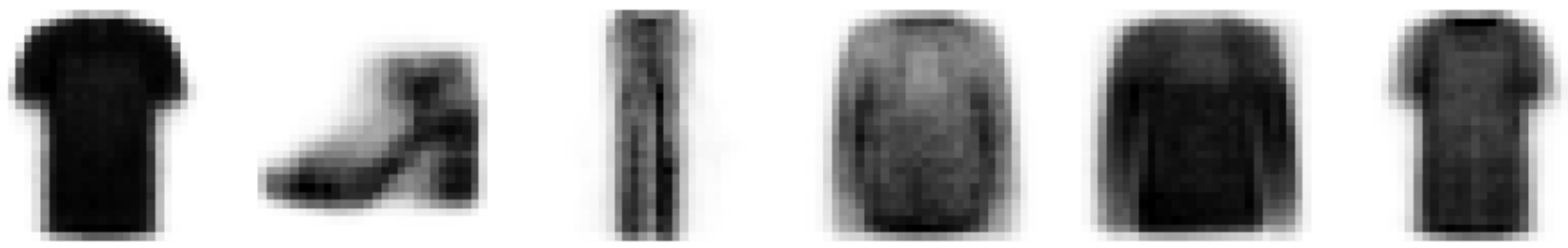
In [34]:
```
# Connect encoder and decoder, compute a reconstruction
def vae(inputs):
    approx_posterior = encoder(inputs)
    decoding_dist = decoder(approx_posterior.sample())
    return decoding_dist.mean()

example_reconstruction = vae(example_x).numpy().squeeze()
```

In [35]:
```
# Plot examples against reconstructions
f, axs = plt.subplots(2, 6, figsize = (16, 5))
for j in range(6):
    axs[0, j].imshow(example_x[j, :, :].squeeze(), cmap = 'binary')
```

```
    axs[1, j].imshow(example_reconstruction[j, :, :], cmap = 'binary')
    axs[0, j].axis('off')
    axs[1, j].axis('off')
```

In [36]:
```
# Generate an example - sample a z value, then sample a reconstruction from p(x|z)
z = prior.sample(6)
generated_x = decoder(z).mean()
```

In [37]:
```
# Display generated_x
f, axs = plt.subplots(1, 6, figsize = (16, 5))
for j in range(6):
    axs[j].imshow(generated_x[j, :, :].numpy().squeeze(), cmap = 'binary')
    axs[j].axis('off')
```



In [38]:
```
# -ELBO estimate using an estimate of the KL divergence
def loss(x, encoding_dist, sampled_decoding_dist, prior, sampled_z):
    recon_loss = -sampled_decoding_dist.log_prob(x)
    kl_approx = (encoding_dist.log_prob(sampled_z) - prior.log_prob(sampled_z))
    return tf.reduce_sum(kl_approx + recon_loss)
```

KL divergence layers

In [39]:
```
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import Dense, Flatten, Reshape
import matplotlib.pyplot as plt
import numpy as np
```

In [40]:
```
# Import Fashion MNIST
(x_train, _), (x_test, _) = tf.keras.datasets.fashion_mnist.load_data()
x_train = x_train.astype('float32') / 256. + 0.5 / 256
x_test = x_test.astype('float32') / 256. + 0.5 / 256
example_x = x_test[:16]

batch_size = 32
x_train = tf.data.Dataset.from_tensor_slices((x_train, x_train)).batch(batch_size)
x_test = tf.data.Dataset.from_tensor_slices((x_test, x_test)).batch(batch_size)
```

In [41]:
```
# Define latent_size and the prior, p(z)
latent_size = 4
prior = tfd.MultivariateNormalDiag(loc = tf.zeros(latent_size))
```

In [42]:
```
# Define the encoding distribution using a tfpl.KLDivergenceAddLoss layer
event_shape = (28, 28)
encoder = Sequential ([
    Flatten(input_shape = event_shape),
    Dense(128, activation = 'relu'),
    Dense(64, activation = 'relu'),
    Dense(32, activation = 'relu'),
    Dense(16, activation = 'relu'),
    Dense(tfpl.MultivariateNormalTriL.params_size(latent_size)),
    tfpl.MultivariateNormalTriL(latent_size),
    tfpl.KLDivergenceAddLoss(prior) # estimates KL[ q[z|x] || p(z) ]
])

# samples z_j from q(z|x_j)
# then computes log q(z_j|x_j) - log p(z_j)
```

In [43]:
```
# See how `KLDivergenceAddLoss` affects `encoder.losses`
# encoder.losses before the network has received any inputs
encoder.losses
```

Out[43]:
```
[<tf.Tensor 'kl_divergence_add_loss/kldivergence_loss/batch_total_kl_divergence:0' shape=() dtype=float32>]
```

In [44]:
```
# Pass a batch of images through the encoder
encoder(example_x)
```

Out[44]:
```
<tfp.distributions.MultivariateNormalTriL 'sequential_4_multivariate_normal_tri_l_MultivariateNormalTriL_MultivariateNormalTriL' batch_shape=[16] event_shape=[4] dtype=float32>
```

In [45]:
```
# See how encoder.losses has changed
encoder.losses
```

Out[45]:
```
[<tf.Tensor: shape=(), dtype=float32, numpy=0.35394907>]
```

In [46]:
```
# Re-specify the encoder using `weight` and `test_points_fn`
encoder = Sequential ([
    Flatten(input_shape = event_shape),
    Dense(128, activation = 'relu'),
    Dense(64, activation = 'relu'),
    Dense(32, activation = 'relu'),
    Dense(16, activation = 'relu'),
    Dense(tfpl.MultivariateNormalTriL.params_size(latent_size)),
    tfpl.MultivariateNormalTriL(latent_size),
    tfpl.KLDivergenceAddLoss (
        prior, use_exact_kl = False, weight = 1.5,
        test_points_fn = lambda q: q.sample(10),
        test_points_reduce_axis = 0
    ) # estimates KL[ q[z|x] || p(z) ]
])

# (n_samples, batch_size, dim_z)
# z_{ij} is the ith sample for x_j (is at (i,j,:) in tensor of samples)
# is mapped to log q(z_{ij}|x_j) - log p(z_{ij})
# => tensor of KL divergences has shape (n_samples, batch_size)
```

In [47]:
```
# Replacing `KLDivergenceAddLoss`  with `KLDivergenceRegularizer` in the previous (probabilistic) layer
divergence_regularizer = tfpl.KLDivergenceRegularizer (
    prior, use_exact_kl = False,
    test_points_fn = lambda q: q.sample(5),
    test_points_reduce_axis = 0
)

encoder = Sequential ([
    Flatten(input_shape = event_shape),
    Dense(128, activation = 'relu'),
    Dense(64, activation = 'relu'),
    Dense(32, activation = 'relu'),
    Dense(16, activation = 'relu'),
    Dense(tfpl.MultivariateNormalTriL.params_size(latent_size)),
    tfpl.MultivariateNormalTriL(latent_size, activity_regularizer = divergence_regularizer)
])
```

In [48]:
```
# Specify the decoder, p(x|z)
decoder = Sequential ([
    Dense(16, activation = 'sigmoid', input_shape = (latent_size,)),
    Dense(32, activation = 'sigmoid'),
    Dense(64, activation = 'sigmoid'),
    Dense(2 * event_shape[0] * event_shape[1], activation = 'exponential'),
    Reshape((event_shape[0], event_shape[1], 2)),
    tfpl.DistributionLambda \
        (lambda t: tfd.Independent(tfd.Beta(concentration1 = t[..., 0], concentration0 = t[..., 1])))
])
```

In [49]:
```
# Connect the encoder and decoder to form the VAE
vae = Model(inputs = encoder.inputs, outputs = decoder(encoder.outputs))
```

In [50]:
```
# Define a loss that only estimates the expected reconstruction error,
# -E_{Z ~ q(z|x)}[log p(x|Z)]
def log_loss(x_true, p_x_given_z):
    return -tf.reduce_sum(p_x_given_z.log_prob(x_true))
```

In [51]:
```
# Compile and fit the model
vae.compile(loss = log_loss)
vae.fit(x_train, validation_data = x_test, epochs = 10)
```

```
Epoch 1/10
1875/1875 [==============================] - 41s 22ms/step - loss: -45334.5352 - val_loss: -51362.8516
Epoch 2/10
1875/1875 [==============================] - 39s 21ms/step - loss: -53343.5117 - val_loss: -54360.8320
Epoch 3/10
1875/1875 [==============================] - 40s 21ms/step - loss: -56219.7969 - val_loss: -57634.0352
Epoch 4/10
1875/1875 [==============================] - 41s 22ms/step - loss: -57441.0781 - val_loss: -58558.2773
Epoch 5/10
1875/1875 [==============================] - 40s 21ms/step - loss: -57147.9180 - val_loss: -57236.7695
Epoch 6/10
1875/1875 [==============================] - 40s 21ms/step - loss: -59585.8672 - val_loss: -59426.6172
Epoch 7/10
1875/1875 [==============================] - 39s 21ms/step - loss: -62342.7188 - val_loss: -62573.9766
Epoch 8/10
1875/1875 [==============================] - 39s 21ms/step - loss: -64889.9609 - val_loss: -65869.4219
Epoch 9/10
1875/1875 [==============================] - 39s 21ms/step - loss: -66032.7812 - val_loss: -67289.3594
Epoch 10/10
1875/1875 [==============================] - 40s 21ms/step - loss: -65745.0938 - val_loss: -68463.7500
```

Out[51]: <tensorflow.python.keras.callbacks.History at 0x7f86dd8279d0>

In [52]:
```
# Generate an example reconstruction
example_reconstruction = vae(example_x).mean().numpy().squeeze()
```

In [53]:
```
# Plot the example reconstructions
f, axs = plt.subplots(2, 6, figsize = (16, 5))

for j in range(6):
    axs[0, j].imshow(example_x[j, :, :].squeeze(), cmap = 'binary')
    axs[1, j].imshow(example_reconstruction[j, :, :], cmap = 'binary')
    axs[0, j].axis('off')
    axs[1, j].axis('off')
```