author mohd-faizy Made with markdown

The TensorFlow Probability library

a) Univariate distributions

These distributions have an `empty_event` shape, indicating that they are distributions for a **single random variable**.

Distribution objects are vital building blocks to build **Probabilistic deep learning Models** as these objects capture the essential operations on probability distributions that we're going to need to build these models.

:heavy_check_mark: Defining our first univariate distribution object

```
 import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions    # Shortcut to the distributions

'''
Standard normal distribution with 0 mean and standard
deviation equal to 1
'''
normal = tfd.Normal(loc=0., scale=1.)
print(normal)
```

```
 # output
tfp.distributions.Normal("Normal", batch_shape=[], event_shape=[], dtype=float32)
```

- `loc` and `scale` : These two **keyword arguments** are required when you instantiate a normal distribution.
- `event_shape=[]` is what captures the dimensionality of the **random variable** itself. Since this distribution is of a single random variable, the event shape is empty, just in the same way that a scalar tensor has an empty shape.

:heavy_check_mark: `Sampling()` from the distribution:

```
 import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

normal = tfd.Normal(loc=0., scale=1.)
normal.sample()
```

```
 # Output
<tf.Tensor: shape=(), dtype=float32, numpy=1.7527679>
```

- One of the key methods for any distribution object is the `sample()` method, which we can use to sample from the distribution.
- If I call the `sample()` method with no arguments, it will return a single sample from this distribution i.e It's a tensor object and has an empty or scalar shape.

```
 import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

normal = tfd.Normal(loc=0., scale=1.)
normal.sample(3) # We can also draw multiple independent samples from the distribution
```

```
# Output
<tf.Tensor: shape=(3,), dtype=float32, numpy=array([ 1.4466898 ,  0.7341992 , -0.91509706], dtype=float32)>
```

Now it returns a tensor of length three with samples from the standard normal distribution.

:heavy_check_mark: `prob()` Method

This funtion evaluates the **Probability Density Function** at the given input.

```
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

normal = tfd.Normal(loc=0., scale=1.)
normal.prob(0.5) # Evaluating a standard normal PDF at the point 0.5
```

```
# Output
<tf.Tensor: shape=(), dtype=float32, numpy=0.35206532>
```

:heavy_check_mark: `log_prob()` Method

This Method computes the **log probability** at the given input.

```
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

normal = tfd.Normal(loc=0., scale=1.)
normal.log_prob(0.5)
```

The Output obtained is natural logarithm of the previous tense value we obtained from the prob method.

```
# Output
<tf.Tensor: shape=(), dtype=float32, numpy=-1.0439385>
```

:heavy_check_mark: Discrete Univariate distribution object(Bernoulli distribution)

In the code below the Bernoulli distribution has one parameter, which is the probability that the random variable takes the value 1.Here we're setting this probability to 0.7 using the probs keyword argument

```
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

bernoulli = tfd.Bernoulli(probs=0.7)
print(bernoulli)
```

```
# Output
tfp.distributions.Bernoulli("Bernoulli", batch_shape=[], event_shape=[], dtype=int32)
```

- Since this is also a univariate distribution, the `event_shape` is empty.
- We can instead instantiate a Bernoulli distribution using the `logits` keyword argument. This might be more convenient depending on how we're using the distribution.

```
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

bernoulli = tfd.Bernoulli(logits=0.847)
print(bernoulli)
```

```
# Output
tfp.distributions.Bernoulli("Bernoulli", batch_shape=[], event_shape=[], dtype=int32)
```

- The relation to the probability value is that the probability is equal to the value of the **sigmoid function** applied to the `logits`.
- The logit's value we can see here, approximately gives the *same Probability value of 0.7* that we had before.

> **Note:** The Bernoulli constructor requires either the prompts or the logits keyword argument to be provided, but not both.

:heavy_check_mark: Sampling from the Bernoulli Distribution

```
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

bernoulli = tfd.Bernoulli(logits=0.847)
bernoulli.sample(3)
```

Here we drawing `3` independent samples from this Bernoulli distribution, which returns the tensor object shown below.

```
# Output
<tf.Tensor: shape=(3,), dtype=int32, numpy=array([0, 1, 1], dtype=int32)>
```

> **Note:** that the type of this tensor is different, it's an int 32 tensor, since a Bernoulli random variable is discrete, and can only take the values 0 or 1.

:heavy_check_mark: Computing the Probabilities from the Bernoulli Distribution

`prob()` Method

```
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

bernoulli = tfd.Bernoulli(logits=0.847)
bernoulli.prob(1) # Computing the probability of the event one
```

```
# Output
# Probability of the event one equals to 0.69993746
<tf.Tensor: shape=(), dtype=float32, numpy=0.69993746>
```

`log_prob` Method

```
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

bernoulli = tfd.Bernoulli(logits=0.847)
bernoulli.log_prob(1)
```

```
# Output
<tf.Tensor: shape=(), dtype=float32, numpy=-0.35676432>
```

:heavy_check_mark: `batch_shape` Argument

Creating another Bernoulli object, Passing in, an array of two values for the probs argument.

```
 import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

batched_bernoulli = tfd.Bernoulli(probs=[0.4, 0.5])
print(batched_bernoulli)
```

```
 # Output
# This object contains a batch of two Bernoulli distributions
tfp.distributions.Bernoulli("Bernoulli", batch_shape=[2], event_shape=[], dtype=int32)
```

> One of the powerful features of distribution objects is that a single object can represent a batch of distributions of the same type.

Since **Bernoulli distribution** is a **Univariate Distribution**, so each of these probability values is used to create a *Separate Bernoulli probability distribution*, both of which are contained within this single Bernoulli object.

- The batch shape is a property of the distribution object, which we can access through the batch shape attribute like this `batched_bernoulli.batch_shape` --> `TensorShape([2])`

```
 import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

batched_bernoulli = tfd.Bernoulli(probs=[0.4, 0.5])
batched_bernoulli.sample(3)
```

We recall the `sample(3)` method with 3 as the argument, we'll get three independent samples from both of these Bernoulli distributions in the batch, so the resulting tensor will have shape `3` by `2` . We can also compute the probability given by each distribution in the batch by passing in an array to the prob method.

```
 # Output
<tf.Tensor: shape=(3, 2), dtype=int32, numpy=
array([[0, 1],
       [0, 1],
       [1, 0]], dtype=int32)>
```

We can also compute the probability given by each distribution in the batch by passing in an array to the prob method

`prob()` Method

```
 import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

batched_bernoulli = tfd.Bernoulli(probs=[0.4, 0.5])
batched_bernoulli.prob([1, 1])
```

Here we're computing the probability of the event value 1 for each distribution, which as we expect returns 0.4 and 0.5 in a tensor of length 2.

```
 # Output
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([0.4, 0.5], dtype=float32)>
```

`log_prob` Method

```
 import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

batched_bernoulli = tfd.Bernoulli(probs=[0.4, 0.5])
batched_bernoulli.log_prob([1, 1])
```

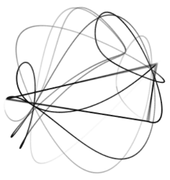
```
 <tf.Tensor: shape=(2,), dtype=float32, numpy=array([-0.9162907, -0.6931472], dtype=float32)>
```

:triangular_flag_on_post: Que :one: What is the shape of the Tensor that is returned from the following call to the sample method:question::question::question:

```
 import tensorflow_probability as tfp
tfd = tfp.distributions
batched_normal = tfd.Normal(loc=[-0.8, 0., 1.9], scale=[1.25, 0.6, 2.8])
batched_normal.sample(2)
```

Ans: `shape=(2, 3)`

```
 # Output
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[0.7245038 , 0.02775155, 2.3153021 ],
       [0.9195015 , 0.20723073, 1.4882771 ]], dtype=float32)>
```



---

b) Multivariate distributions

:heavy_check_mark: Multivariate Gaussian Distribution

Using the `MultivariateNormalDiag` class to create a two-dimensional diagonal Gaussian

```
 import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

mv_normal = tfd.MultivariateNormalDiag(loc=[-1., 0.5], scale_diag=[1., 1.5])
print(mv_normal)
```

```
 # Output
tfp.distributions.MultivariateNormalDiag("MultivariateNormalDiag", batch_shape=[], event_shape=[2], dtype=float32)
```

Note: If we didn't use the `scale_diag` argument, then the **covariance matrix** would be the **identity matrix** by default, that is the standard deviation of one for each component.

> we can access the event shape by using the following code `mv_normal.event_shape` which is `#(2,)`

:small_orange_diamond: Sampling the Multivariate Distribution

```
 import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

mv_normal = tfd.MultivariateNormalDiag(loc=[-1., 0.5], scale_diag=[1., 1.5])
mv_normal.sample(3) # This will produce 3 independent samples from the multivariate distribution.
```

> The distribution has an event shape of `2`, which means that each of those `3` samples will be two-dimensional, so the resulting tensor has a shape of `3` by `2`.

```
# Output
<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[-0.68523514,  0.97463423],
       [-1.8620067 , -1.7331753 ],
       [-1.1199658 , -1.0719669 ]], dtype=float32)>
```

:heavy_check_mark: Batch Normal Distribution

Creating a batch normal distribution by passing in an array of values for both the `loc` and the `scale` arguments. This distribution will have a `batch_shape` of two and an empty or scalar `event_shape`, since the normal is a **univariate distribution**

```
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

batched_normal = tfd.Normal(loc=[-1., 0.5], scale=[1., 1.5])
print(batched_normal)
```

```
# Output
tfp.distributions.Normal("Normal", batch_shape=[2], event_shape=[], dtype=float32)
```

:small_orange_diamond: Sampling the Batch Normal Distribution

```
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

batched_normal = tfd.Normal(loc=[-1., 0.5], scale=[1., 1.5])
batched_normal.sample(3)
```

```
# Output
<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[-0.9501647 , -1.041851  ],
       [-0.5308881 , -0.77874947],
       [-2.6619897 , -1.2877599 ]], dtype=float32)>
```

:large_orange_diamond: Comparing the Multivariate Gaussian Distribution & Batch Normal Distribution

```
# Output Multivariate Gaussian Distribution
tfp.distributions.MultivariateNormalDiag("MultivariateNormalDiag", batch_shape=[], event_shape=[2], dtype=float32)

# Output Batch Normal Distribution
tfp.distributions.Normal("Normal", batch_shape=[2], event_shape=[], dtype=float32)
```

- `MultivariateNormalDiag` distribution is a distribution of a **2D** random variable, as `event_shape=[2]`.
- The `Normal` distribution below is a batch of two distributions of a single random variable, `batch_shape=[2]` and `event_shape=[]` being empty.

Computing the `log_probs` for both Distributions

:small_blue_diamond: **For Multivariate Distribution:**

```
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

mv_normal = tfd.MultivariateNormalDiag(loc=[-1., 0.5], scale_diag=[1., 1.5])
mv_normal.log_prob([0.2, -1.8])
```

```
# Output
<tf.Tensor: shape=(), dtype=float32, numpy=-4.1388974>
```

when we pass in a length two array to the `log_prob method`, this array represents a single realization of the two-dimensional random variable. Correspondingly, the tensor that is returned contains a single `log_prob` value.

:small_blue_diamond: **For Batch Normal Distribution:**

```
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

batched_normal = tfd.Normal(loc=[-1., 0.5], scale=[1., 1.5])
batched_normal.log_prob([0.2, -1.8])
```

```
# Output
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([-1.6389385, -2.499959 ], dtype=float32)>
```

Here, the input array represents a value for each of the random variables for the two normal distributions in the batch. The `log_ probs` for each of these two realizations are evaluated and return these two values in a length two tensor, as we can see above.

:heavy_check_mark: Batch Multivariate Distribution

```
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

batched_mv_normal = tfd.MultivariateNormalDiag(
    loc=[[-1., 0.5],[2., 0], [-0.5, 1.5]],
    scale_diag=[[1., 1.5], [2., 0.5], [1., 1.]]
    )

print(batched_mv_normal)
```

```
# Output
tfp.distributions.MultivariateNormalDiag(
    "MultivariateNormalDiag",
    batch_shape=[3],
    event_shape=[2],
    dtype=float32
    )
```

- In the above distributions Each argument is taking a `3` by `2` array. The last dimension corresponds to the `event_size`, and remaining dimensions get absorbed into the `batch_shape`.
- That means that this `MultivariateNormalDiag` distribution has an `event_shape=[2]` and `batch_shape=[3]`. In other words, it contains a batch of three multivariate Gaussians, each of which is a distribution over a two-dimensional random variable.

:small_orange_diamond: Sampling above Distribution

```
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

batched_mv_normal = tfd.MultivariateNormalDiag(
    loc=[[-1., 0.5],[2., 0], [-0.5, 1.5]],
    scale_diag=[[1., 1.5], [2., 0.5], [1., 1.]]
    )

batched_mv_normal.sample(2)
```

```
 # Output
<tf.Tensor: shape=(2, 3, 2), dtype=float32, numpy=
array([[[-1.7474746 , -0.39185297],
        [ 2.605815  , -0.6507868 ],
        [-0.2742607 ,  1.7156713 ]],

       [[-0.22726142, -0.8659065 ],
        [ 1.665063  ,  0.9733336 ],
        [-0.57607734,  3.7140775 ]]], dtype=float32)>
```

In the Tensor: `shape=(2, 3, 2)` :

- The first `2` here comes from the `sample_size` of two.
- The `3` is the `batch_size` .
- The final `2` is the `event_size` of the distribution

:triangular_flag_on_post: Que :two: Suppose we define the following `MultivariateNormalDiag` object:question::question::question:

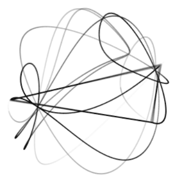
```
 import tensorflow_probability as tfp
tfd = tfp.distributions
batched_mv_normal = tfd.MultivariateNormalDiag(
    loc=[[0.3, 0.8, 1.1], [2.3, -0.3, -1.]],
    scale_diag=[[1.5, 1., 0.4], [2.5, 1.5, 0.5]])

# Que: What is the shape of the Tensor returned by the following?
batched_mv_normal.log_prob([0., -1., 1.])
```

Ans: `shape=(2,)`

```
 # Output
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([ -3.9172401, -11.917513 ], dtype=float32)>
```

> This section shows how `sample` , `batch` , and `event_shapes` are used in distribution objects. And By designing distribution objects in this way, the **TensorFlow probability library** can exploit the **Performance gains** from **Vectorizing Computations**



---

c) The Independent distribution

When we compute `log_probs` of an `input_event` , -- distribution calculates the **log probability** of *each event in the batch of distributions* and *returns a single number for each distribution in the batch.*

But sometimes we might want to reinterpret a batch of independent distributions over an `event_space` as a single joint distribution over a product of `event_spaces` .

> For **example**, our model might assume that the features of our data are independent given a class label. In this case, we could set up a separate class conditional distribution for each feature in a batch. But this batch of distributions is really a **joint distribution** over all the features, and we'd like that to be reflected in the `batch_shape` and `event_shape` properties, and the outputs of the `log_probs` method. This is precisely the setting of the **Naive Bayes classifier.**

Naive Bayes classifier :question::question::question:

In statistics, **Naive Bayes classifiers** are a family of simple `probabilistic classifiers` based on applying `Bayes theorem` with strong assumption that the value of a particular feature is **Independent** of the value of any other feature, given the class variable.They are among the simplest Bayesian network models, but coupled with **Kernel density estimation**, they can achieve higher accuracy levels.

:checkered_flag::checkered_flag::checkered_flag:

> **For Example**, A **fruit** may be considered to be an `apple` :apple: if it is red , round, and about 10 cm in diameter. A naive Bayes classifier considers each of these features to contribute **independently** to the probability that this fruit is an apple, regardless of any possible correlations between the color, roundness, and diameter features.

:anchor: **Below** :anchor: is the `MultivariateNormalDiag` distribution, which has a two-dimensional `event_space` , and an instance of the Univariate Normal dDistribution, which instead has a `batch_shape` of two.

```
 # Importing tensorflow probability library
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

# Instance of a multivariate Distributions
mv_normal = tfd.MultivariateNormalDiag(loc=[-1., 0.5], scale_diag=[1., 1.5])
print(mv_normal)

'''
tfp.distributions.MultivariateNormalDiag("MultivariateNormalDiag",
                                          batch_shape=[],
                                          event_shape=[2],
                                          dtype=float32
                                          )
'''
# Instance of a univariate Distributions
batched_normal = tfd.Normal(loc=[-1., 0.5], scale=[1., 1.5])
print(batched_normal)

'''
tfp.distributions.Normal("Normal",
                          batch_shape=[2],
                          event_shape=[],
                          dtype=float32
                          )
'''
```

Multivariate Normal Diag	Normal distribution
The **2-D** array that we're passing into the `MultivariateNormalDiag` is interpreted as a single realization of the **2-D random variable**, and the distribution returns the **log probability** of this realization.	In the case of the **Normal distribution**, the array is interpreted as values for each of the random variables in the batch. The distribution then calculates the `log probability` for each batch **separately** from each other, and returns them both in a length 2 tensor.

Independent Distribution

The **Independent Distribution** gives us a way to absorb some or all of the **batch dimensions** into the `event_shape` .

```

import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

batched_normal = tfd.Normal(loc=[-1., 0.5], scale=[1., 1.5])

independent_normal = tfd.Independent(batched_normal, reinterpreted_batch_ndims=1)
print(independent_normal)
```

> The `reinterpreted_batch_ndims` argument is an optional keyword argument, that specifies how many of the batch dimensions should be absorbed into the event space.

```
 # Output

tfp.distributions.Independent("IndependentNormal", batch_shape=[], event_shape=[2], dtype=float32)
```

Since there's only one batch dimension, which is equal to two, this will become part of the `event_space` .

```
 import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

batched_normal = tfd.Normal(loc=[-1., 0.5], scale=[1., 1.5])

independent_normal = tfd.Independent(batched_normal, reinterpreted_batch_ndims=1)
independent_normal.log_prob([-0.2, 1.8])
```

:point_up: So If we again compute the `log_prob` of an input array of **length 2** , we now see that the result is a **Scalar**, just as we had with the `MultivariateNormalDiag` distribution. Mathematically, this **independent distribution** is now equivalent to the `MultivariateNormalDiag` distribution we had before.

```
 # Output

<tf.Tensor: shape=(), dtype=float32, numpy=-2.9388976>
```

:point_right:**Another Example of independent Distribution:**

```
 import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

batched_normal = tfd.Normal(loc=[[-1., 0.5], [0., 1.], [0.3, -0.1]],
                            scale=[[1., 1.5], [0.2, 0.8], [2., 1.]])
print(batched_normal)

independent_normal = tfd.Independent(batched_normal, reinterpreted_batch_ndims=1)
print(independent_normal)
```

```
 # Output

tfp.distributions.Normal("Normal", batch_shape=[3, 2], event_shape=[], dtype=float32)

tfp.distributions.Independent("IndependentNormal", batch_shape=[3], event_shape=[2], dtype=float32)
```

The default value for `reinterpreted_batch_ndims` is none, and with this default, the independent distribution will take **all batch dimensions except the first into the** `event_space` .

when `reinterpreted_batch_ndims=2` :question::question::question:

If we wanted to transfer both **batch dimensions** into the `event_space` , then set `reinterpreted_batch_ndims=2` . the result is an independent distribution with an empty `batch_shape=[]` and a **rank2** `event_shape=[3,2]` . So we use independent distributions to manipulate the `batch_shape` and `event_shape` properties of distributions. **This is a useful tool that we can use to create distributions that have the properties that we want.**

```
 # Output

tfp.distributions.Normal("Normal", batch_shape=[3, 2], event_shape=[], dtype=float32)

tfp.distributions.Independent("IndependentNormal", batch_shape=[], event_shape=[3, 2], dtype=float32)
```

:triangular_flag_on_post: Que :three: What is the shape of the Tensor that is returned from the following call to the log_prob method:question::question::question:

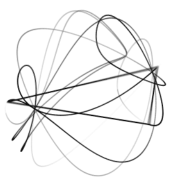
```
import numpy as np
import tensorflow_probability as tfp
tfd = tfp.distributions

probs = 0.5 * tf.ones((2, 4, 5))
dist = tfd.Independent(tfd.Bernoulli(probs=probs))
dist.log_prob(np.zeros((4, 5)))
```

Answer: `shape=(2,)`

```
# Output

<tf.Tensor: shape=(2,), dtype=float32, numpy=array([-13.862944, -13.862944], dtype=float32)>
```



d) Sampling and log probs

```
import tensorflow_probability as tfp
tfd = tfp.distributions

# Univariate Distribution
# Passing Rate as a (2x3) array
exp = tfd.Exponential(rate=[[1., 1.5, 0.8], [0.3, 0.4, 1.8]])
print(exp)
```

```
# output

tfp.distributions.Exponential("Exponential", batch_shape=[2, 3], event_shape=[], dtype=float32)
```

- Empty `event_shape=[]` indicate that it's a **Scalar Random Variable**.

Let's apply the independent distribution

```
import tensorflow_probability as tfp
tfd = tfp.distributions

exp = tfd.Exponential(rate=[[1., 1.5, 0.8], [0.3, 0.4, 1.8]])
ind_exp = tfd.Independent(exp)
print(ind_exp)
```

Note: We not passing in the `reinterpreted_batch_ndims` keyword argument, so the independent distribution by default is converting all but the first batch dimension into the `event_shape`.

So we Now have a `batch_shape=[2]` and an `event_shape=[3]`

```
# Output

tfp.distributions.Independent("IndependentExponential", batch_shape=[2], event_shape=[3], dtype=float32)
```

Sampling the Independent distribution:

```
 import tensorflow_probability as tfp
tfd = tfp.distributions

exp = tfd.Exponential(rate=[[1., 1.5, 0.8], [0.3, 0.4, 1.8]])
ind_exp = tfd.Independent(exp)
ind_exp.sample(4)
```

This will return the tensor of `shape=(4, 2, 3)` **Order**:

- `sample_shape` comes first - `4`
- then `batch_shape` - `2`
- finally `event_shape` - `3`

```
 # Output

<tf.Tensor: shape=(4, 2, 3), dtype=float32, numpy=
array([[[0.878214  , 0.10177544, 1.0854489 ],
        [0.0861064 , 0.39792454, 0.10340352]],

       [[1.7000741 , 0.3063202 , 0.5343896 ],
        [7.7648444 , 0.5894142 , 0.27669817]],

       [[0.07917001, 0.05815649, 0.03421117],
        [4.5987663 , 0.6690836 , 0.10257661]],

       [[0.34468663, 0.5017041 , 1.0447694 ],
        [1.6518481 , 3.3994725 , 0.06371849]]], dtype=float32)>
```

Creating exponential distribution of rank 4

```
 import tensorflow_probability as tfp
tfd = tfp.distributions

rates = [
        [[[1., 1.5, 0.8], [0.3, 0.4, 1.8]]],
        [[[0.2, 0.4, 1.4], [0.4, 1.1, 0.9]]]
]

exp = tfd.Exponential(rate=rates)
print(exp)

ind_exp = tfd.Independent(exp, reinterpreted_batch_ndims=2)
print(ind_exp)
```

- **Exponential** `batch_shape=[2, 1, 2, 3]`, `event_shape=[]` : Since the exponential distribution is a univariate distribution, all of these dimensions will become part of the `batch_shape` and the `event_shape` will be empty.
- **IndependentExponential** `batch_shape=[2, 1]`, `event_shape=[2, 3]` : here we have a distribution with a **rank-2** `batch_shape` and a **rank-2** `event_shape`. The independent distribution has **absorbed** the last two batch dimensions into the `event_space`.

Now Sampling the above distribution

```
import tensorflow_probability as tfp
tfd = tfp.distributions

rates = [
        [[[1., 1.5, 0.8], [0.3, 0.4, 1.8]]],
        [[[0.2, 0.4, 1.4], [0.4, 1.1, 0.9]]]
]

exp = tfd.Exponential(rate=rates)

ind_exp = tfd.Independent(exp, reinterpreted_batch_ndims=2)
ind_exp.sample([4, 2])
```

Outputs the Tensor of `shape=(4, 2, 2, 1, 2, 3)`

The resulting tensor will be rank-6

- `sample_shape` **(4 X 2)**
- `batch_shape` **(2 X 1)**
- `event_shape` **(2 X 3)**

Using the `log_prob` Method on Independent distribution

This is a simple example of using **broadcasting** when computing `log probs`.

```
import tensorflow_probability as tfp
tfd = tfp.distributions

rates = [
        [[[1., 1.5, 0.8], [0.3, 0.4, 1.8]]],
        [[[0.2, 0.4, 1.4], [0.4, 1.1, 0.9]]]
]

exp = tfd.Exponential(rate=rates)

ind_exp = tfd.Independent(exp, reinterpreted_batch_ndims=2)
ind_exp.log_prob(0.5)
```

Remember the `event_shape` of our distribution is `(2 X 3)`, so the distribution will compute the **log probability** for *each event in the batch*. This means that the `0.5` input will be **broadcast** to both the `event_shape` & `batch_shape`.

```
# Output

<tf.Tensor: shape=(2, 1), dtype=float32, numpy=
array([[-4.2501554],
       [-5.3155975]], dtype=float32)>
```

Think of this as computing `log probs` for `(2 X 3)` event, where each `entry = 0.5` & the **log probability** for this event is computed for each distribution in the `batch`.

Passing (1 x 3) shape input in `log_prob` Method

```
import tensorflow_probability as tfp
tfd = tfp.distributions

rates = [
        [[[1., 1.5, 0.8], [0.3, 0.4, 1.8]]],
        [[[0.2, 0.4, 1.4], [0.4, 1.1, 0.9]]]
]

exp = tfd.Exponential(rate=rates)

ind_exp = tfd.Independent(exp, reinterpreted_batch_ndims=2)
ind_exp.log_prob([[0.3, 0.5, 0.8]]) # Shape - (1, 3)
```

```
# Output

<tf.Tensor: shape=(2, 1), dtype=float32, numpy=
array([[-4.7701554],
       [-5.885597 ]], dtype=float32)>
```

So According to the broadcasting rules, this is broadcastable against the `batch_shape` and `event_shape` of `(2 X 1 X 2 X 3)` . So in this case, the input will be broadcast against the first dimension of the `event_shape` as well as both dimensions of the `batch_shape` . The result is again, a tensor with the **same shape** as the **batch_shape**.

Passing the Rank-5 input in `log_prob` Method

```
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

rates = [
        [[[1., 1.5, 0.8], [0.3, 0.4, 1.8]]],
        [[[0.2, 0.4, 1.4], [0.4, 1.1, 0.9]]]
]

exp = tfd.Exponential(rate=rates)

ind_exp = tfd.Independent(exp, reinterpreted_batch_ndims=2)
ind_exp.log_prob(tf.random.uniform((5, 1, 1, 2, 1)))
```

Queston Is, this input broadcastable against the `batch_shape` and `event_shape` of `[B, E] = [2 x 1 x 2 x 3]` If it is, what is the resulting tensor shape returned by the log prob method:question::question::question:

The answer is yes, it is broadcastable against the `batch_shape` & `event_shape` .

- The **last dimension** of the input will be broadcast against the **second dimension** of the `event_shape` and
- The **second dimension** of the input will be broadcast against the **first dimension** of the `batch_shape` .

we can think of this extra dimension of five which is here as being the `sample_shape` . So the **log probability** will be computed for **Each Event** of **Each Batch** and for **Each Sample**. The resulting tensor is then going to be the concatenation of sample and `batch_shapes` , which is **Output** = `Tensor: shape=(5, 2, 1)`

:triangular_flag_on_post: Que :four: Suppose we define the following MultivariateNormalDiag object:question::question::question:

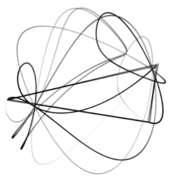
```
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

loc = tf.zeros((2, 3, 1))
scale_diag = tf.ones(4)
dist = tfd.Independent(tfd.MultivariateNormalDiag(loc=loc, scale_diag=scale_diag))

# What is the shape of the Tensor returned by the following?
dist.log_prob(tf.random.uniform((2, 1, 1, 4)))
```

Answer `Tensor: shape=(2, 2)`

> Using the above methods we can understand of how `sample_shape` , `batch_shape` and `event_shapes` behave, particularly when it comes to **Creating distribution objects** and using the `sample` or `log_prob` methods we can also apply the **broadcasting** to these methods or instantiating a distribution objects.



e) Trainable distributions

Making the parameters of distribution objects trainable

This is smae as using an **optimizer object** to apply gradients obtained from a loss function and data.Let's take a **Normal distribution** object as an example And let's say **we want to learn the mean of this distribution**.

```
 import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

normal = tfd.Normal(loc=tf.Variable(0., name='loc'), scale=1.)
normal.trainable_variables # The distribution object also has a trainable variables attribute
```

```
 # Output
(<tf.Variable 'loc:0' shape=() dtype=float32, numpy=0.0>,)
```

The `mean value` of this normal distribution is now trainable and can be updated according to some learning principle. The learning principle that we often use when training deep learning models is **Maximum likelihood**. The goal is to maximize the likelihood or probability of our data.

Finding the parameters that maximize the likelihood is the same as finding the parameters that minimize the negative log likelihood.

hence defining the **negative log likelihood** of our simple model as a function of a training set as `x_train`. `x_train` is an **array of scalar data points**. Then we can compute the **log probability** of these data points according to our model, using the `log_prob` method and this method will return a tensor that is the same shape as `x_train`.

For the above Example we don't need to run an **Optimization** procedure to find the value of the variable that **Minimizes the negative log likelihood**. We can just solve this directly. The answer will be the mean of the data points in x_train

```

import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

normal = tfd.Normal(loc=tf.Variable(0., name='loc'), scale=1.)

def nll(x_train):  # Defining the negative log likelihood(nll)
    return -tf.reduce_mean(normal.log_prob(x_train))

'''
This function get_loss_and_grads takes a batch of training examples as an input
and computes the loss and gradients for our model.
'''
@tf.function
def get_loss_and_grads(x_train):
    with tf.GradientTape() as tape:
        tape.watch(normal.trainable_variables)
        loss = nll(x_train)
    grads = tape.gradient(loss, normal.trainable_variables)
    return loss, grads

optimizer = tf.keras.optimizers.SGD(learning_rate=0.05)

for _ in range(num_steps):
    loss, grads = get_loss_and_grads(x_samples)
    optimizer.apply_gradients(zip(grads, normal.trainable_variables))
```

- To compute the gradients, we set up a `tf.GradientTape` context and start with a **call** to `tape.watch` to specify the variables that we want to track in the operations that follow. Here, that's the trainable variables of our normal distribution.
- The loss that we want to compute is the **negative log likelihood** as given by our `nll function`. Finally, we can now compute the**gradient of the loss** with respect to the **trainable variables** of our distribution with a call to `tape.gradient`.
- **The calculation of the loss and gradients is where most of the heavy lifting happens in terms of computation.** So we can help to speed this up by using the `@tf.function` decorator, and this makes a graph out of the function.

- The last thing we need to do is to define our **training loop**. We'll use an **Optimizer object** to update the **Trainable Variables**, and here I'm setting up an **SGD optimizer** with a **learning rate of 0.05**.
- There's no batching going on, and the loss and gradients are being computed on the entire training data `x_samples` in each iteration of the loop.

In this Module we learned

:black_circle: How to create different types of distribution objects using TensorFlow probability, including **Univariates, Multivariates, discrete** and **Continuous distributions**.

:black_circle: We have seen the **three core** methods of these distributions by the `sample`, `prob`, and `log prob` methods.

:black_circle: We have also seen how the `independent distribution` can be used to manipulate the `batch_shapes` and the `event_shapes`, and how `broadcasting` can be applied with the `prob` and `log_prob` methods.

:blackcircle: And _Finally*, how the parameters of these distributions can be learned in a training loop, similar to that of training loops for deep learning models in TensorFlow.