

---

## Distrax



Distrax is a lightweight library of probability distributions and bijectors. It acts as a JAX-native reimplementation of a subset of TensorFlow Probability (TFP), with some new features and emphasis on extensibility.

### Installation

You can install the latest released version of Distrax from PyPI via:

```
1 pip install distrax
```

or you can install the latest development version from GitHub:

```
1 pip install git+https://github.com/deepmind/distrax.git
```

To run the tests or examples you will need to install additional requirements.

### Design Principles

The general design principles for the DeepMind JAX Ecosystem are addressed in this blog. Additionally, Distrax places emphasis on the following:

1. **Readability.** Distrax implementations are intended to be self-contained and read as close to the underlying math as possible.
2. **Extensibility.** We have made it as simple as possible for users to define their own distribution or bijector. This is useful for example in reinforcement learning, where users may wish to define custom behavior for probabilistic agent policies.
3. **Compatibility.** Distrax is not intended as a replacement for TFP, and TFP contains many advanced features that we do not intend to replicate. To this end, we have made the APIs for distributions and bijectors as cross-compatible as possible, and provide utilities for transforming between equivalent Distrax and TFP classes.

### Features

#### Distributions

Distributions in Distrax are simple to define and use, particularly if you're used to TFP. Let's compare the two side-by-side:

---

```
1 import distrax
2 import jax
3 import jax.numpy as jnp
4
5 from tensorflow_probability.substrates import jax as tfp
6 tfd = tfp.distributions
7
8 key = jax.random.PRNGKey(1234)
9 mu = jnp.array([-1., 0., 1.])
10 sigma = jnp.array([0.1, 0.2, 0.3])
11
12 dist_distrax = distrax.MultivariateNormalDiag(mu, sigma)
13 dist_tfp = tfd.MultivariateNormalDiag(mu, sigma)
14
15 samples = dist_distrax.sample(seed=key)
16
17 # Both print 1.775
18 print(dist_distrax.log_prob(samples))
19 print(dist_tfp.log_prob(samples))
```

In addition to behaving consistently, Distrax distributions and TFP distributions are cross-compatible. For example:

```
1 mu_0 = jnp.array([-1., 0., 1.])
2 sigma_0 = jnp.array([0.1, 0.2, 0.3])
3 dist_distrax = distrax.MultivariateNormalDiag(mu_0, sigma_0)
4
5 mu_1 = jnp.array([1., 2., 3.])
6 sigma_1 = jnp.array([0.2, 0.3, 0.4])
7 dist_tfp = tfd.MultivariateNormalDiag(mu_1, sigma_1)
8
9 # Both print 85.237
10 print(dist_distrax.kl_divergence(dist_tfp))
11 print(tfd.kl_divergence(dist_distrax, dist_tfp))
```

Distrax distributions implement the method `sample_and_log_prob`, which provides samples and their log-probability in one line. For some distributions, this is more efficient than calling separately `sample` and `log_prob`:

```
1 mu = jnp.array([-1., 0., 1.])
2 sigma = jnp.array([0.1, 0.2, 0.3])
3 dist_distrax = distrax.MultivariateNormalDiag(mu, sigma)
4
5 samples = dist_distrax.sample(seed=key, sample_shape=())
6 log_prob = dist_distrax.log_prob(samples)
7
8 # A one-line equivalent of the above is:
9 samples, log_prob = dist_distrax.sample_and_log_prob(seed=key,
10               sample_shape=())
```

---

TFP distributions can be passed to Distrax meta-distributions as inputs. For example:

```
1 key = jax.random.PRNGKey(1234)
2
3 mu = jnp.array([-1., 0., 1.])
4 sigma = jnp.array([0.2, 0.3, 0.4])
5 dist_tfp = tfd.Normal(mu, sigma)
6
7 metadist_distrax = distrax.Independent(dist_tfp,
    reinterpreted_batch_ndims=1)
8 samples = metadist_distrax.sample(seed=key)
9 print(metadist_distrax.log_prob(samples)) # Prints 0.38871175
```

To use Distrax distributions in TFP meta-distributions, Distrax provides the wrapper `to_tfp`. A wrapped Distrax distribution can be directly used in TFP:

```
1 key = jax.random.PRNGKey(1234)
2
3 distrax_dist = distrax.Normal(0., 1.)
4 wrapped_dist = distrax.to_tfp(distrax_dist)
5 metadist_tfp = tfd.Sample(wrapped_dist, sample_shape=[3])
6
7 samples = metadist_tfp.sample(seed=key)
8 print(metadist_tfp.log_prob(samples)) # Prints -3.3409896
```

## Bijectors

A “bijector” in Distrax is an invertible function that knows how to compute its Jacobian determinant. Bijectors can be used to create complex distributions by transforming simpler ones. Distrax bijectors are functionally similar to TFP bijectors, with a few API differences. Here is an example comparing the two:

```
1 import distrax
2 import jax.numpy as jnp
3
4 from tensorflow_probability.substrates import jax as tfp
5 tfb = tfp.bijectors
6 tfd = tfp.distributions
7
8 # Same distribution.
9 distrax.Transformed(distrax.Normal(loc=0., scale=1.), distrax.Tanh())
10 tfd.TransformedDistribution(tfd.Normal(loc=0., scale=1.), tfb.Tanh())
```

Additionally, Distrax bijectors can be composed and inverted:

```
1 bij_distrax = distrax.Tanh()
2 bij_tfp = tfb.Tanh()
3
```

---

```

4 # Same bijector.
5 inv_bij_distrax = distrax.Inverse(bij_distrax)
6 inv_bij_tfp = tfb.Invert(bij_tfp)
7
8 # These are both the identity bijector.
9 distrax.Chain([bij_distrax, inv_bij_distrax])
10 tfb.Chain([bij_tfp, inv_bij_tfp])

```

All TFP bijectors can be passed to Distrax, and can be freely composed with Distrax bijectors. For example, all of the following will work:

```

1 distrax.Inverse(tfb.Tanh())
2
3 distrax.Chain([tfb.Tanh(), distrax.Tanh()])
4
5 distrax.Transformed(tfd.Normal(loc=0., scale=1.), tfb.Tanh())

```

Distrax bijectors can also be passed to TFP, but first they must be transformed with `to_tfp`:

```

1 bij_distrax = distrax.to_tfp(distrax.Tanh())
2
3 tfb.Invert(bij_distrax)
4
5 tfb.Chain([tfb.Tanh(), bij_distrax])
6
7 tfd.TransformedDistribution(tfd.Normal(loc=0., scale=1.), bij_distrax)

```

Distrax also comes with `Lambda`, a convenient wrapper for turning simple JAX functions into bijectors. Here are a few `Lambda` examples with their TFP equivalents:

```

1 distrax.Lambda(lambda x: x)
2 # tfb.Identity()
3
4 distrax.Lambda(lambda x: 2*x + 3)
5 # tfb.Chain([tfb.Shift(3), tfb.Scale(2)])
6
7 distrax.Lambda(jnp.sinh)
8 # tfb.Sinh()
9
10 distrax.Lambda(lambda x: jnp.sinh(2*x + 3))
11 # tfb.Chain([tfb.Sinh(), tfb.Shift(3), tfb.Scale(2)])

```

Unlike TFP, bijectors in Distrax do not take `event_ndims` as an argument when they compute the Jacobian determinant. Instead, Distrax assumes that the number of event dimensions is statically known to every bijector, and uses `Block` to lift bijectors to a different number of dimensions. For example:

```

1 x = jnp.zeros([2, 3, 4])
2

```

---

```

3 # In TFP, `event_ndims` can be passed to the bijector.
4 bij_tfp = tfb.Tanh()
5 ld_1 = bij_tfp.forward_log_det_jacobian(x, event_ndims=0) # Shape =
    [2, 3, 4]
6
7 # Distrax assumes `Tanh` is a scalar bijector by default.
8 bij_distrax = distrax.Tanh()
9 ld_2 = bij_distrax.forward_log_det_jacobian(x) # ld_1 == ld_2
10
11 # With `event_ndims=2`, TFP sums the last 2 dimensions of the log det.
12 ld_3 = bij_tfp.forward_log_det_jacobian(x, event_ndims=2) # Shape =
    [2]
13
14 # Distrax treats the number of dimensions statically.
15 bij_distrax = distrax.Block(bij_distrax, ndims=2)
16 ld_4 = bij_distrax.forward_log_det_jacobian(x) # ld_3 == ld_4

```

Distrax bijectors implement the method `forward_and_log_det` (some bijectors additionally implement `inverse_and_log_det`), which allows to obtain the `forward` mapping and its log Jacobian determinant in one line. For some bijectors, this is more efficient than calling separately `forward` and `forward_log_det_jacobian`. (Analogously, when available, `inverse_and_log_det` can be more efficient than `inverse` and `inverse_log_det_jacobian`.)

```

1 x = jnp.zeros([2, 3, 4])
2 bij_distrax = distrax.Tanh()
3
4 y = bij_distrax.forward(x)
5 ld = bij_distrax.forward_log_det_jacobian(x)
6
7 # A one-line equivalent of the above is:
8 y, ld = bij_distrax.forward_and_log_det(x)

```

## Jitting Distrax

Distrax distributions and bijectors can be passed as arguments to jitted functions. User-defined distributions and bijectors get this property for free by subclassing `distrax.Distribution` and `distrax.Bijector` respectively. For example:

```

1 mu_0 = jnp.array([-1., 0., 1.])
2 sigma_0 = jnp.array([0.1, 0.2, 0.3])
3 dist_0 = distrax.MultivariateNormalDiag(mu_0, sigma_0)
4
5 mu_1 = jnp.array([1., 2., 3.])
6 sigma_1 = jnp.array([0.2, 0.3, 0.4])
7 dist_1 = distrax.MultivariateNormalDiag(mu_1, sigma_1)
8

```

---

```
9 jitted_kl = jax.jit(lambda d_0, d_1: d_0.kl_divergence(d_1))
10
11 # Both print 85.237
12 print(jitted_kl(dist_0, dist_1))
13 print(dist_0.kl_divergence(dist_1))
```

**A note about `vmap` and `pmap`** The serialization logic that enables Distrax objects to be passed as arguments to jitted functions also enables functions to map over them as data using `jax.vmap` and `jax.pmap`.

However, ***support for this behavior is experimental and incomplete. Use caution when applying `jax.vmap` or `jax.pmap` to functions that take Distrax objects as arguments, or return Distrax objects.***

Simple objects such as `distrax.Categorical` may behave correctly under these transformations, but more complex objects such as `distrax.MultivariateNormalDiag` may generate exceptions when used as inputs or outputs of a `vmap`-ed or `pmap`-ed function.

## Subclassing Distributions and Bijectors

User-defined distributions can be created by subclassing `distrax.Distribution`. This can be achieved by implementing only a few methods:

```
1 class MyDistribution(distrax.Distribution):
2
3     def __init__(self, ...):
4         ...
5
6     def _sample_n(self, key, n):
7         samples = ...
8         return samples
9
10    def log_prob(self, value):
11        log_prob = ...
12        return log_prob
13
14    def event_shape(self):
15        event_shape = ...
16        return event_shape
17
18    def _sample_n_and_log_prob(self, key, n):
19        # Optional. Only when more efficient implementation is possible.
20        samples, log_prob = ...
21        return samples, log_prob
```

---

Similarly, more complicated bijectors can be created by subclassing `distrax.Bijector`. This can be achieved by implementing only one or two class methods:

```
1 class MyBijector(distrax.Bijector):
2
3     def __init__(self, ...):
4         super().__init__(...)
5
6     def forward_and_log_det(self, x):
7         y = ...
8         logdet = ...
9         return y, logdet
10
11    def inverse_and_log_det(self, y):
12        # Optional. Can be omitted if inverse methods are not needed.
13        x = ...
14        logdet = ...
15        return x, logdet
```

## Examples

The `examples` directory contains some representative examples of full programs that use Distrax.

`hmm.py` demonstrates how to use `distrax.HMM` to combine distributions that model the initial states, transitions, and observation distributions of a Hidden Markov Model, and infer the latent rates and state transitions in a changing noisy signal.

`vae.py` contains an example implementation of a variational auto-encoder that is trained to model the binarized MNIST dataset as a joint `distrax.Bernoulli` distribution over the pixels.

`flow.py` illustrates a simple example of modelling MNIST data using `distrax.MaskedCoupling` layers to implement a normalizing flow, and training the model with gradient descent.

## Acknowledgements

We greatly appreciate the ongoing support of the TensorFlow Probability authors in assisting with the design and cross-compatibility of Distrax.

Special thanks to Aleyna Kara and Kevin Murphy for contributing the code upon which the Hidden Markov Model and associated example are based.

## Citing Distrax

This repository is part of the DeepMind JAX Ecosystem, to cite Distrax please use the citation:

---

```
1 @software{deepmind2020jax,  
2   title = {The {D}eep{M}ind {JAX} {E}cosystem},  
3   author = {DeepMind and Babuschkin, Igor and Baumli, Kate and Bell,  
            Alison and Bhupatiraju, Surya and Bruce, Jake and Buchlovsky,  
            Peter and Budden, David and Cai, Trevor and Clark, Aidan and  
            Danihelka, Ivo and Dedieu, Antoine and Fantacci, Claudio and  
            Godwin, Jonathan and Jones, Chris and Hemsley, Ross and Hennigan,  
            Tom and Hessel, Matteo and Hou, Shaobo and Kapturowski, Steven and  
            Keck, Thomas and Kemaev, Iurii and King, Michael and Kunesch,  
            Markus and Martens, Lena and Merzic, Hamza and Mikulik, Vladimir  
            and Norman, Tamara and Papamakarios, George and Quan, John and  
            Ring, Roman and Ruiz, Francisco and Sanchez, Alvaro and Sartran,  
            Laurent and Schneider, Rosalia and Sezener, Eren and Spencer,  
            Stephen and Srinivasan, Srivatsan and Stanojevi\{'c}, Milo\v{s}  
            and Stokowiec, Wojciech and Wang, Luyu and Zhou, Guangyao and  
            Viola, Fabio},  
4   url = {http://github.com/deepmind},  
5   year = {2020},  
6 }
```