2nd Symposium on Advances in Approximate Bayesian Inference, 2019 1–17

`Bijectors.jl`: Flexible transformations for probability distributions

Tor Erlend Fjelde TEF30@CAM.AC.UK
University of Cambridge

Kai Xu KAI.XU@ED.AC.UK
University of Edinburgh

Mohamed Tarek M.MOHAMED@STUDENT.ADFA.EDU.AU
UNSW Canberra

Sharan Yalburgi SHARANYALBURGI@GMAIL.COM
BITS Pilani

Hong Ge HG344@CAM.AC.UK
University of Cambridge

Abstract

Transforming one probability distribution to another is a powerful tool in Bayesian inference and machine learning. Some prominent examples are constrained-to-unconstrained transformations of distributions for use in Hamiltonian Monte Carlo and constructing flexible and learnable densities such as normalizing flows. We present `Bijectors.jl`, a software package in Julia for transforming distributions, available at github.com/TuringLang/Bijectors.jl. The package provides a flexible and composable way of implementing transformations of distributions without being tied to a computational framework. We demonstrate the use of `Bijectors.jl` on improving variational inference by encoding known statistical dependencies into the variational posterior using normalizing flows, providing a general approach to relaxing the mean-field assumption usually made in variational inference.

1. Introduction

When working with probability distributions in Bayesian inference and probabilistic machine learning, transforming one probability distribution to another comes up quite often. For example, when applying Hamiltonian Monte Carlo on constrained distributions, the constrained density is usually transformed to an unconstrained density for which the sampling is performed (Neal, 2012). Another example is to construct highly flexible and learnable densities often referred to as *normalizing flows* (Dinh et al., 2014; Huang et al., 2018; Durkan et al., 2019); for a review see Kobyzev et al. (2019).

When a distribution P is transformed into some other distribution Q using some measurable function b, we write $Q = b_* P$ and say Q is the *push-forward* of P. When b is a differentiable bijection with a differentiable inverse, i.e. a *diffeomorphism* or a *bijector* (Dillon et al., 2017), the induced or pushed-forward distribution Qit is obtained by a simple application of *change of variables*. Specifically, given a distribution P on some $\Omega \subseteq \mathbb{R}^d$ with density $p : \Omega \to [0, \infty)$, and a bijector $b : \Omega \to \tilde{\Omega}$ for some $\tilde{\Omega} \subseteq \mathbb{R}^d$, the induced or pushed forward distribution $Q = b_* P$ has density

$$q(y) = p\big(b^{-1}(y)\big)\, |\det \mathcal{J}_{b^{-1}}(y)| \quad \text{or} \quad q\big(b(x)\big) = \frac{p(x)}{|\det \mathcal{J}_b(x)|}$$

1.1. Coupling flow

As mentioned, one application of this idea is learnable bijectors such as normalizing flows. One particular family of normalizing flow which has received a lot of attention is *coupling flows* (Dinh et al., 2014; Rezende and Mohamed, 2015; Huang et al., 2018). The idea is to use certain parts of the input vector $\boldsymbol{x}$, say, $\boldsymbol{x}_{I_1}$ to construct parameters for a bijector f (the *coupling law*), which is then applied to a *different* part of the input vector, say, $\boldsymbol{x}_{I_2}$. In full generality, a coupling flow c_{I_1,I_2}, the transformation in a coupling flow, is defined

$$\begin{aligned} c_{I_1,I_2}(\cdot\,; f, \theta): \quad & \mathbb{R}^d \to \mathbb{R}^d & c^{-1}_{I_1,I_2}(\cdot\,; f, \theta): \quad & \mathbb{R}^d \to \mathbb{R}^d \\ & \boldsymbol{x}_{I \setminus I_2} \mapsto \boldsymbol{x}_{I \setminus I_2} & & \boldsymbol{y}_{I \setminus I_2} \mapsto \boldsymbol{y}_{I \setminus I_2} \\ & \boldsymbol{x}_{I_2} \mapsto f\big(x_{I_2}\,; \theta(x_{I_1})\big) & & \boldsymbol{y}_{I_2} \mapsto f^{-1}\big(y_{I_2}; \theta(y_{I_1})\big) \end{aligned} \tag{1}$$

where $I_1, I_2 \subset I := \{1, \dots, d\}$ are *disjoint*. As long as $f\big(\cdot\,; \theta(\boldsymbol{x}_{I_1})\big) : \mathbb{R}^{I_2} \to \mathbb{R}^{I_2}$ is a bijector, c_{I_1,I_2} is invertible since $y_{I_1} = x_{I_1}$. Note the parameter-map θ can be arbitrarily complex.

2. `Bijectors.jl`: a package for bijectors in Julia

`Bijectors.jl` is a framework for creating and using bijectors in the Julia programming language. The main idea is to treat standard constrained-to-unconstrained bijectors, e.g. $\log : \mathbb{R} \to (0, \infty)$, and more complex and possibly parameterized bijectors, e.g. coupling flows, *as the same* just as they are mathematically the same. This turns out to be quite a useful abstraction allowing seamless interaction between standard and learnable bijectors, making something like *automatic differentiation variational inference* (ADVI; Kucukelbir et al., 2016) easy to implement (see Source Code 1). Table 1 shows supported mathematical operations. Only $b(x)$ and $b^{-1}(y)$ need to be manually implemented for a new bijector b.

Another example is the introduction of *neural autoregressive flows* (NAFs; Huang et al., 2018) where the inverse autoregressive flow (IAF; Kingma et al., 2016) is extended by replacing the affine coupling law used in IAF with a monotonic deep neural network. Despite the novel introduction of neural network, in `Bijectors.jl` the only difference between IAF and NAF is the choice of `Bijector` as the coupling law.

2.1. Related work

A summarization of the related work and how it compares to `Bijectors.jl` can be seen in Table 2. More detailed comparisons can be found in Appendix A.1.

1. This refers to the `torch.distributions` submodule. *After* our submission, another transformer module based on `PyTorch` was released in `Pyro` (Bingham et al., 2018): `pyro.distributions.transforms`. At the time of writing, we have not yet done a thorough comparison with this. At a first glance its features seem similar in natu re to `tensorflow.probability` which can be found in Table 2.
2. `Bijectors.jl` is agnostic to array-types used, therefore GPU functionality is provided basically for free by using the independent package `CuArrays.jl` to construct the arrays.

Operation	Method	Automatic
$b \mapsto b^{-1}$	`inv(b)` or `b^(-1)`	✓
$(b_1, b_2) \mapsto (b_1 \circ b_2)$	`b1` $\circ$ `b2`	✓
$(b_1, b_2) \mapsto [b_1, b_2]$	`stack(b1, b2)`	✓
$(b, n) \mapsto b^n := b \circ \cdots \circ b$ (n times)	`b^n`	✓
$x \mapsto b(x)$	`b(x)`	✗
$y \mapsto b^{-1}(y)$	`inv(b)(y)` or `b^(-1)(y)`	✗
$x \mapsto \log\lvert\det \mathcal{J}_b(x)\rvert$	`logabsdetjac(b, x)`	AD
$x \mapsto \big(b(x), \log\lvert\det \mathcal{J}_b(x)\rvert\big)$	`forward(b, x)`	✓
$P \mapsto Q := b_* Q$	`Q = transformed(P, b)`	✓
$y \sim Q$	`y = rand(Q)`	✓
$y \mapsto \log q(y)$	`logpdf(Q, y)`	✓
$P \mapsto b$ s.t. $\text{support}(b_* P) = \mathbb{R}^d$	`bijector(P)`	✓
$\big(x \sim P,\ b(x),\ \log\lvert\det \mathcal{J}_b(x)\rvert,\ \log q(y)\big)$	`forward(Q)`	✓

Table 1: The table shows all the mathematical operations supported in `Bijectors.jl`. Automatic operations need not be defined for new bijectors. Automatic differentiation (AD) can be used to define `logabsdetjac` optionally.

	`Bijectors.jl`	`TensorFlow`	`PyTorch`[1]	`PyMC3/Stan`
Unified `Bijector`	✓	✓	✗	✗
Compositions: $b_1 \circ b_2$	✓	✓	✗	N/A
Decoupled from distributions	✓	✓	✗	✗
Statically sized input	soon	✓	✓	N/A
Dynamically sized input	✓	✗	✓	N/A
Interop with ecosystem	✓	limited	limited	limited
GPU compatibility	✓[2]	✓	✓	✓
Automatic caching	manual	✓	✓	N/A

Table 2: Comparison of support for bijectors in different packages, nothing else in frameworks. "Unified `Bijector`" refers to whether or not both more classical and parameterized bijectors, e.g. normalizing flows, are considered as the same. *Limited* means applicability is constrained to the underlying computational framework. *Caching* refers to caching of input/output pairs and thus zero-cost inversion in simple cases.

2.2. Examples & Interoperability

Using `Bijectors.jl` is straight-forward; a couple of examples can be seen in Source Code 1. This also demonstrates the interoperability with other packages; we use the reverse-mode AD package `Tracker.jl` to compute the gradients of the flow. Other examples are using `CuArrays.jl` to get GPU compatibility, `Distributions.jl` (Lin et al., 2019) for implementations of distributions, using normalizing flows from `Bijectors.jl` in your generative model in `Turing.jl` (Ge et al., 2018), using neural networks from `Flux.jl` to define the coupling law in a coupling flow, and so on. For more examples, see the project website.

```julia
julia> using Bijectors

julia> dist = Beta(2, 2);                # From Distributions.jl

julia> b = bijector(dist)                # (0, 1) → ℝ
Bijectors.Logit{Float64}(0.0, 1.0)

julia> td = transformed(dist, b);        # x ∼ Beta(2, 2) ⟹ b(x) ∈ ℝ

julia> b⁻¹ = inv(b)                       # ℝ → (0, 1)
Inversed{Bijectors.Logit{Float64},0}(Bijectors.Logit{Float64}(0.0, 1.0))

julia> # Works like a standard `Distribution`
       y = rand(td)                      # ∈ ℝ
-0.6044789394180846

julia> logpdf(td, y), logpdf(dist, b⁻¹(y)) + logabsdetjac(b⁻¹, y)
(-1.1608110510380623, -1.1608110510380623)

julia> (b ∘ b⁻¹)(y) == y
true
```

```julia
dists = (InverseGamma(2, 3), Beta()) # target distributions

base = MvNormal(zeros(2), ones(2));  # base distribution
ibs = inv.(bijector.(dists));        # support(dist) -> [ℝ, ℝ]
sb = stack(ibs...)                   # -> ℝ

# Taking gradients of ELBO wrt. parameters:
using Tracker: param
b = sb ∘ PlanarLayer(2, param)        # NF + add gradient tracking
td = transformed(base, b)

function elbo(x, flow, num_samples)
    # Most efficient path to obtain both `z` and `logjac`
    z, z, logq, logjac = forward(flow, num_samples)
    # Assumes existence of `logjoint` which computes `p(x, z)`
    return mean(logjoint(x, z) + logjac) - entropy(q.base)
end

# Assumes `x` is the observations
Tracker.back!(elbo(x, td, 10))       # estimate ∇ with 10 samples
```

Source Code 1: Example code using `Bijectors.jl`. **Left:** demonstrates the basics of `Bijectors.jl` in the Julia REPL. **Right:** demonstrates how to use `Bijectors.jl` to perform NF-ADVI.

3. Adding structure to mean-field VI using coupling flows

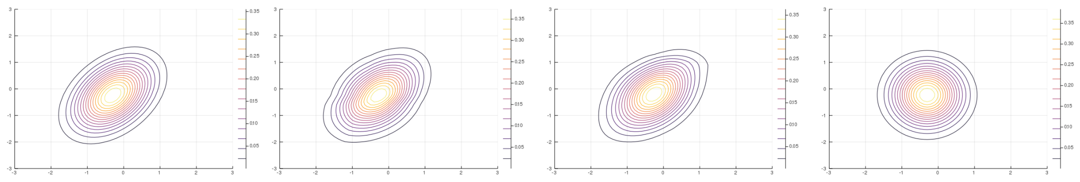
We demonstrate how to use `Bijectors.jl` by a possible approach to relaxing the mean-field assumption commonly made in variational inference through the use of normalizing flows. We consider a simple two-dimensional Gaussian with known covariance matrix with non-zero off-diagonal entries, i.e. different components are dependent, defined as follows

$$
\begin{aligned}
&\boldsymbol{m} \sim \mathcal{N}(\mathbf{0}, \mathbf{1}) \\
&\boldsymbol{x}_i \overset{i.i.d.}{\sim} \mathcal{N}(\boldsymbol{m}, LL^T), \quad i = 1, \dots, n \quad \text{where} \quad L = \begin{pmatrix} 10 & 0 \\ 10 & 10 \end{pmatrix}
\end{aligned} \tag{2}
$$

In this case we can obtain an analytical expression for the posterior $p(\boldsymbol{m} \mid \{\boldsymbol{x}_i\}_{i=1}^n)$, and can indeed observe that the covariance matrix has non-zero off-diagonals. In this case, the mean-field assumption often made in variational inference is incorrect. Recall that in variational inference the objective is to maximize the *evidence lower bound* (ELBO) of the variational posterior $q(\boldsymbol{m})$ and the true posterior $p(\boldsymbol{m} \mid \{\boldsymbol{x}_i\}_{i=1}^n)$,

$$
\mathrm{ELBO}\big(q(\boldsymbol{m})\big) = \sum_{i=1}^{n} \mathbb{E}_{\boldsymbol{m} \sim q(\boldsymbol{m})}\big[\log p(x_i, \boldsymbol{m})\big] - \mathbb{E}_{\boldsymbol{m} \sim q(\boldsymbol{m})}\big[\log q(\boldsymbol{m})\big] \tag{3}
$$

For the ELBO of a transformed distribution, see Appendix B.2. Here we propose using a mean-field multivariate normal as a starting point, and then combine this with coupling flows to encode the structure of the model we are approximating into the variational posterior at a low computational cost. The idea is to encode an *undirected* edge between the



(a) Posterior (b) NFVI (Affine) (c) NFVI (RQS) (d) MFVI

Figure 1: Contour plots of the resulting densities from the experiment described in Appendix B.1. For (b) we used an affine transform and (c) *rational-quadratic spline (RQS)* as the coupling law. See Figure 3 & 5 in appendix for more examples.

random variables m_1 and m_2 by adding *directed* mappings in both directions; we do this by composing coupling flows $c_{\{2\},\{1\}}$ and $c_{\{1\},\{2\}}$.

For the coupling flows, we experimented with two different coupling laws f, *affine* (Dinh et al., 2014) and the recently introduced *rational-quadratic splines* (Durkan et al., 2019). The parameter maps θ_1 and θ_2, respectively, were defined by a simple neural network in both cases. The resulting density, letting $Q_{\boldsymbol{\mu},\boldsymbol{\sigma}}$ be the distribution of an isotropic multivariate Gaussian with mean $\boldsymbol{\mu}$ and variance $\boldsymbol{\sigma}$, is given by

$$b = c_{\{1\},\{2\}}(\cdot\ ; f, \theta_2) \circ c_{\{2\},\{1\}}(\cdot\ ; f, \theta_1) \quad \text{and} \quad Q := b_* Q_{\boldsymbol{\mu},\boldsymbol{\sigma}} \tag{4}$$

We then optimized the ELBO w.r.t. the parameters of the neural networks θ_1, θ_2, $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ to obtain our variational posteriors. The result of the standard mean-field VI (MFVI) and this particular normalizing flow VI (NFVI) applied to the model in Equation (2) can be seen in Figure 1. Here we observe that NFVI captures the correlation structure of the true posterior in Figure 1(a) while MFVI, as expected, fails to do so. This is also reflected in the value of the ELBO for the two approaches (see Appendix B.1). This can potentially provide a flexible approach to taking advantage of structure in the joint distribution when performing variational inference without introducing a large number of parameters in addition to the mean-field parameters. See Appendix B.1 for specifics of the experiment.

4. Conclusion & future work

We presented `Bijectors.jl`, a framework for working with bijectors and thus transformations of distributions. We then demonstrated the flexibility of `Bijectors.jl` in an application of introducing correlation structure to the mean-field ADVI approach. We believe `Bijectors.jl` will be a useful tool for future research, especially in exploring normalizing flows and their place in variational inference.

An interesting note about the NF variational posterior we constructed is that it only requires a constant number of extra parameters on top of what is required by mean-field normal VI. This approach can be applied in more general settings where one has access to the directed acyclic graph (DAG) of the generative model we want to perform inference. Then this approach will scale linearly with the number of unique edges between random variables. It is also possible in cases where we have an undirected graph representing a model by simply adding a coupling in both directions. This would be very useful for tackling issues faced when using mean-field VI and would be of interest to explore further.

Acknowledgments

We would like to thank the organizers of Google Summer of Code (GSOC); two of the authors began their work on `Bijectors.jl` while funded by GSOC 2019. We also thank the authors of `tensorflow_probability` (Dillon et al. (2017)) as it was an inspiration for our approach.

References

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep Universal Probabilistic Programming. *arXiv preprint arXiv:1810.09538*, 2018.

Joshua V. Dillon, Ian Langmore, Dustin Tran, Eugene Brevdo, Srinivas Vasudevan, Dave Moore, Brian Patton, Alex Alemi, Matt Hoffman, and Rif A. Saurous. Tensorflow distributions. *CoRR*, 2017. URL `http://arxiv.org/abs/1711.10604v1`.

Laurent Dinh, David Krueger, and Yoshua Bengio. Nice: Non-linear independent components estimation. *CoRR*, 2014. URL `http://arxiv.org/abs/1410.8516v6`.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul): 2121–2159, 2011.

Conor Durkan, Artur Bekasov, Iain Murray, and George Papamakarios. Neural spline flows. *CoRR*, 2019. URL `http://arxiv.org/abs/1906.04032v1`.

Hong Ge, Kai Xu, and Zoubin Ghahramani. Turing: A language for flexible probabilistic inference. In Amos Storkey and Fernando Perez-Cruz, editors, *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*, volume 84 of *Proceedings of Machine Learning Research*, pages 1682–1690, Playa Blanca, Lanzarote, Canary Islands, 09–11 Apr 2018. PMLR. URL `http://proceedings.mlr.press/v84/ge18b.html`.

Chin-Wei Huang, David Krueger, Alexandre Lacoste, and Aaron Courville. Neural autoregressive flows. *CoRR*, 2018. URL `http://arxiv.org/abs/1804.00779v1`.

Mike Innes. Flux: Elegant machine learning with julia. *Journal of Open Source Software*, 2018. doi: 10.21105/joss.00602.

Durk P Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, and Max Welling. Improved variational inference with inverse autoregressive flow. In *Advances in neural information processing systems*, pages 4743–4751, 2016.

Ivan Kobyzev, Simon Prince, and Marcus A. Brubaker. Normalizing flows: Introduction and ideas. *CoRR*, 2019. URL `http://arxiv.org/abs/1908.09257v1`.

Alp Kucukelbir, Dustin Tran, Rajesh Ranganath, Andrew Gelman, and David M. Blei. Automatic differentiation variational inference. *CoRR*, 2016. URL `http://arxiv.org/abs/1603.00788v1`.

Dahua Lin, John Myles White, Simon Byrne, Douglas Bates, Andreas Noack, John Pearson, Mathieu Besançon, Alex Arslan, Kevin Squire, David Anthoff, Theodore Papamarkou, Jan Drugowitsch, Moritz Schauer, John Zito, Avik Sengupta, Brian J Smith, Glenn Moynihan, Giuseppe Ragusa, Alexey Stukalov, Gord Stephen, Christoph Dann, micklat, Martin O'Leary, Mike J Innes, Jiahao Chen, Iain Dunning, Gustavo Lacerda, Simon Kornblith, Richard Reeve, and Kai Xu. Juliastats/distributions.jl: v0.21.1, July 2019. URL https://doi.org/10.5281/zenodo.3356998.

Radford M. Neal. Mcmc using hamiltonian dynamics. *CoRR*, 2012. URL http://arxiv.org/abs/1206.1901v1.

Danilo Jimenez Rezende and Shakir Mohamed. Variational inference with normalizing flows. *CoRR*, 2015. URL http://arxiv.org/abs/1505.05770v6.

Appendix A. `Bijectors.jl`

A.1. Related work

For related work we have mainly compared against Tensorflow's `tensorflow_probability`, which is used by other known packages such `pymc4`, and PyTorch's `torch.distributions`, which is used by packages such as `pyro`. Other frameworks which make heavy use of such transformations using their own implementations are `stan`, `pymc3`, and so on. But in these frameworks the transformations are mainly used to transform distributions from constrained to unconstrained and vice versa with little or no integration between those transformation and the more complex ones, e.g. normalizing flows. `pymc3` for example support normalizing flows, but treat them differently from the constrained-to-unconstrained transformations. This means that composition between standard and parameterized transformations is not supported.

Of particular note is the `bijectors` framework in `tensorflow_probability` introduced in (Dillon et al., 2017). One could argue that this was indeed the first work to take such a drastic approach to the separation of the determinism and stochasticity, allowing them to implement a lot of standard distributions as a `TransformedDistribution`. This framework was also one of the main motivations that got the authors of `Bijectors.jl` interested in making a similar framework in Julia. With that being said, other than the name, we have not set out to replicate `tensorflow_probability` and most of the direct parallels were observed after-the-fact, e.g. a transformed distribution is defined by the `TransformedDistribution` type in both frameworks. Instead we believe that Julia is a language well-suited for such a framework and therefore one can innovate on the side of implementation. For example in Julia we can make use of *code-generation* or *meta-programming* to do program transformations in different parts of the framework, e.g. the composition $b \circ b^{-1}$ is transformed into the identity function at compile time.[3] In addition, the *actual code* that performs $b \circ b^{-1}$ is `b ∘ inv(b)` and `b ∘ b^(-1)` due to the unicode-support for operators and functions in Julia.

A.1.1. Higher order bijectors

Similar to `tensorflow_probability` we can use higher-order bijectors to construct new bijectors. Examples of such are `Inverse`, `Compose`, and `Stacked`. A significant difference is that in `Bijectors.jl`, the constructors are rarely called explicitly by the user but instead through a completely intuitive interface, e.g. `inv(b)` gives you the `Inverse`, `b1 ∘ b2` gives you the composition of `b1` and `b2`, `stack(b1, b2)` gives you the two bijectors "stacked" together. Moreover, if `b` actually has a "named" inverse, e.g. `b = Exp()`, then `inv(b)` will result in `Log()` rather than some thin wrapper `Inversed(Exp())`. Irregardless of whether the bijector has a named inverse or not, the dual-nature is exploited in compositions so that `b ∘ inv(b)` results in `Identity()`. For type-stable code, this is all done at compile-time.

A particularly nice one is the `Stacked(bijectors, ranges)` which allows the user to specify which parts (or ranges) of the input vector should be passed to which of the "stacked". For all methods acting on a `Stacked` the loop for iterating through the different ranges and applying the corresponding `Bijector` will be unrolled, meaning that this

3. Julia takes a *lazy* approach to compilation, i.e. a function declaration is not compiled until needed, and then any subsequent calls with a similar signature will use the already compiled native method.

abstraction has a zero-cost overhead and the only cost is the evaluation of corresponding methods on for the bijectors it wraps.

A.1.2. Program transforms

In a limited sense `Bijectors.jl` can do what is known as *program transformations*. A good example is $b \circ b^{-1}$ resulting in identity *at compile-time* for simple transformations which we have mentioned before.

In `tensorflow_probability` indeed $b \circ b^{-1}$ is reduced to the identity mapping, *not* by collapsing the computational graph but instead by the use of *caching*. This means that when $(b \circ b^{-1})(x)$ is evaluated, work will only be done for the $b^{-1}(x)$ evaluation. When $b^{-1}(x)$ is evaluated by b, the cached value x used to evaluate b^{-1} just before will be returned immediately. `torch.distributions` take a similar approach but because caching can come with its own issues, especially when used in conjunction with automatic differentiation, there are cases where it will fail, e.g. dependency reversal.

In `Bijectors.jl` there are two parts of this story. First off, $b \circ b^{-1}$ will, as noted earlier, be compiled to the identity map upon compilation, i.e. there is zero-overhead at run-time to this evaluation. *But* one nice property of the Tensorflow and PyTorch approach which uses caching is that one can write code that looks like

```
# Samples x from base and returns b(x)
y = rand(transformed_distribution)

# Do some more computation potentially involving y
# ...

# Zero cost due to caching
x = b.inverse(y)  # <= equivalent of inv(b)(y) in Bijectors.jl
return x
```

In `Bijectors.jl` this has to be done manually by the user through the `forward` method for a `TransformedDistribution`. Recall from Table 1 that `forward` returns a 4-tuple `(x, b(x), logabsdetjac(b, x), logpdf(q, b(x)))` using the most efficient computation path. Therefore to replicate the above example in `Bijectors.jl`, we can do

```
# Samples x from base and returns y = b(x)
x, y, _, _ = forward(transformed_distribution)

# do some more computation potentially involving y
# ...

return x
```

Therefore "caching" in `Bijectors.jl` cannot be done *across* function barriers at the time of writing (unless the function explicitly returns all values used). On the bright side one can explicitly do caching, making it more difficult to do something wrong in addition to the fact that the computation is transparent from the users perspective.

Appendix B. Adding structure to mean-field VI using coupling flows

B.1. Experiment: 2D multivariate normal

In the experimental setup we generate data by fixing $\boldsymbol{m} = \mathbf{0}$ and generating $n = 100$ samples from Equation (2). This resulted in a posterior multivariate normal with covariance matrix

$$\Sigma = \begin{pmatrix} 0.4 & 0.2 \\ 0.2 & 0.6 \end{pmatrix}$$

This was done by design, as we specifically chose $L = \begin{pmatrix} 10 & 0 \\ 10 & 10 \end{pmatrix}$ to get a posterior covariance matrix with non-zero off-diagonals.

Let $q_{\boldsymbol{\mu},\boldsymbol{\sigma}}$ denote the density of a multivariate Gaussian with mean $\boldsymbol{\mu}$ and diagonal covariance $\boldsymbol{\sigma}$, and b denote the coupling flow in Equation (1) with f as a rational-quadratic spline (RQS) with $K = 3$ knot points and bin $[-50, 50]$, θ as a neural network consisting of one layer with a $(3K-1) \times 1$ weight matrix and bias with identity activation, i.e. a simple affine transformation. See (Durkan et al., 2019) for more information on RQS. We use `Distributions.jl` for implementation of the Gaussian multivariate distribution (Lin et al., 2019).

We then performed variational inference on the model in Equation (2) with variational posterior q taking the following approaches:

1. $q := q_{\boldsymbol{\mu},\boldsymbol{\sigma}}$ with objective

$$\max_{\boldsymbol{\mu},\boldsymbol{\sigma}} \widehat{\mathrm{ELBO}}\big(q_{\boldsymbol{\mu},\boldsymbol{\sigma}}(\boldsymbol{m})\big)$$

   resulting in Figure 5(*d*),

2. $q := q_{\boldsymbol{\mu},\boldsymbol{\sigma}}$ with objective

$$\min_{\boldsymbol{\mu},\boldsymbol{\sigma}} \widehat{\mathrm{D}_{\mathrm{KL}}}\big(q_{\boldsymbol{\mu},\boldsymbol{\sigma}}(\boldsymbol{m}), p(\boldsymbol{m} \mid \{\boldsymbol{x}\}_{i=1}^{n})\big)$$

   resulting in Figure 5(*e*),

3. $q := (b_\theta)_* q_{\boldsymbol{\mu},\boldsymbol{\sigma}}$ with objective

$$\max_{\boldsymbol{\mu},\boldsymbol{\sigma},\theta} \widehat{\mathrm{ELBO}}\Big(\big((b_\theta)_* q_{\boldsymbol{\mu},\boldsymbol{\sigma}}\big)(\boldsymbol{m})\Big)$$

   resulting in Figure 5(*b*),

4. $q := (b_\theta)_* q_{\boldsymbol{\mu},\boldsymbol{\sigma}}$ with objective

$$\min_{\boldsymbol{\mu},\boldsymbol{\sigma},\theta} \widehat{\mathrm{D}_{\mathrm{KL}}}\Big(\big((b_\theta)_* q_{\boldsymbol{\mu},\boldsymbol{\sigma}}\big)(\boldsymbol{m}), p(\boldsymbol{m} \mid \{\boldsymbol{x}\}_{i=1}^{n})\Big)$$

   resulting in Figure 5(*c*).

The resulting densities can be observed in Figure 3. Note that here we have used a slight abuse of notation writing $\max_\theta$ to mean "maximize wrt. *parameters* of θ". The expressions for the KL-divergence and the ELBO, which under the transformation by a bijector picks up an additional term, see Equation (5) and Equation (8), respectively. In all cases we set the number of samples used in the Monte-Carlo estimate of the objective to be $m = 50$.

In all cases we used a `DecayedADAGrad` from `Turing.jl` to perform gradient updates. This is a classical `ADAGrad` (Duchi et al., 2011) but with a decay for the accumulated gradient norms. This is to circumvent the possibility of large initial gradient norms bringing all subsequent optimization steps to practically zero step-size. For `DecayedADAGrad` we used a base step-size $\eta = 0.001$, post-factor decay $\beta_{\text{post}} = 1.0$ and pre-factor decay $\beta_{\text{pre}} = 0.9$, and we performed 5 000 optimization steps before terminating.

In general we of course do not have access to the true posterior and so we cannot minimize the KL-divergence between the variational posterior and the true posterior directly, but instead have to do so implicitly by minimizing the ELBO. In theory there is no difference, but in practice one usually observe a significantly lower variance in the gradient estimates of the KL-divergence compared to the ELBO. We therefore also performed VI using the KL-divergence to verify that the NF did not lack the expressibility to capture the true posterior, but that the slight inaccuracy in the variational posterior obtained by maximizing the ELBO was indeed due to the variance in the gradient estimate. And, as expected, minimizing the KL-divergence directly in the MF-case did not provide much of a gain compared to maximizing the ELBO.

Numerical results for multiple runs where the ELBO was used as an objective can be seen in Table 3 and Figure 2; the NFVI approach consistently obtains lower KL divergence and a greater ELBO.

The main quantity of interest is the KL-divergence which quantifies the *difference* between the variational posterior and the true posterior. The ELBO is a lower bound on the evidence and thus the actual values can vary widely across experiments. Additionally, the difference between the ELBO of two distributions with respect to the *same set of observations* is equal to the difference between KL-divergence on that set of observations, and so we gain no additional information of the difference between the variational posterior and the true posterior by looking at the ELBO. Therefore we visualize the KL-divergence instead of the ELBO in Figure 2, but still provide the numerical values for both in Table 3.

	MFVI		NFVI	
	KL	ELBO	KL	ELBO
Run 1	0.092305	-748.64	0.0022869	-748.526
Run 2	0.0729075	-753.33	0.0014182	-753.31
Run 3	0.0841829	-730.58	0.0034853	-730.486
Run 4	0.0887038	-748.34	0.0030914	-748.256
Run 5	0.0897964	-754.498	0.0043046	-754.418

Table 3: (Rational-Quadratic Spline coupling law) Exponentially smoothed estimates of the last 4000 (out of 5000) optimization steps. As can be seen in Figure 2, after the 1000th step is basically when the optimas are reached. Here the ELBO has been used as the objective.

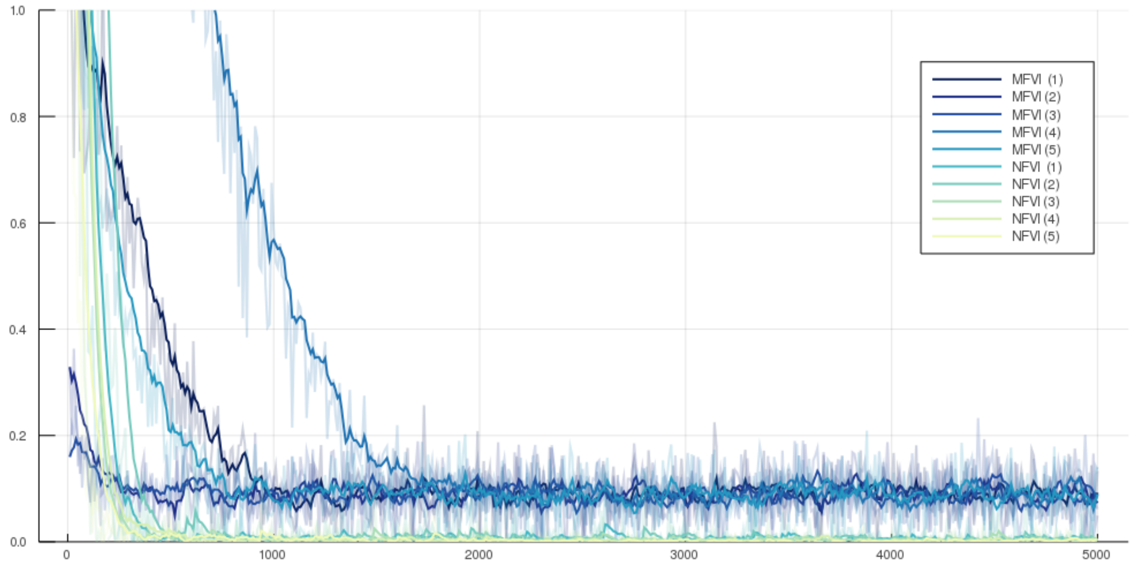


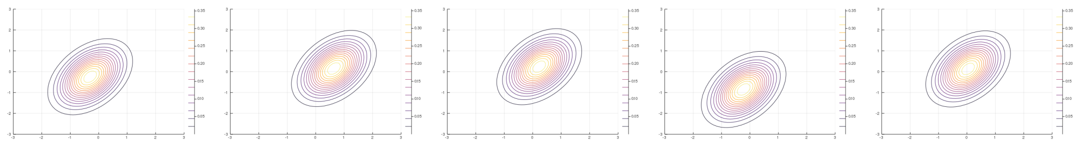
Figure 2: (Rational-Quadratic Spline coupling law) KL-divergences of different runs where the objective used is the ELBO. The opaque lines represent the true samples while the non-opaque lines correspond to smoothed estimates using exponential smoothing.

B.1.1. Affine coupling law

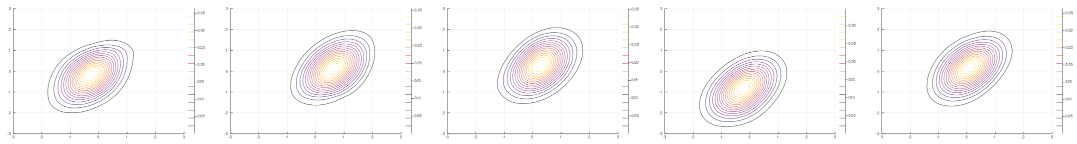
We also performed the same experiment using an *affine* transformation f as a coupling law. The setup is identical, but now θ is a neural network consisting of two layers; the first layer is a dense layer with 2×1 weight matrix and bias and ReLU activation, and the second layer is a dense layer with 2×2 weight matrix and bias and identity activation. In `Flux.jl`, which we have used for the neural network part of the bijector, this is given by `Chain(Dense(1, 2, relu), Dense(2, 2))` (Innes, 2018). As one can see in Table 4 and Figure 4, even with an affine coupling law we obtain very good approximations.

	MFVI		NFVI	
	KL	ELBO	KL	ELBO
Run 1	0.110996	-748.651	0.005025	-748.517
Run 2	0.113718	-756.825	-0.000113	-756.684
Run 3	0.091862	-735.373	-0.000506	-735.263
Run 4	0.094180	-753.609	0.001358	-753.468
Run 5	0.090356	-748.306	0.004798	-748.182

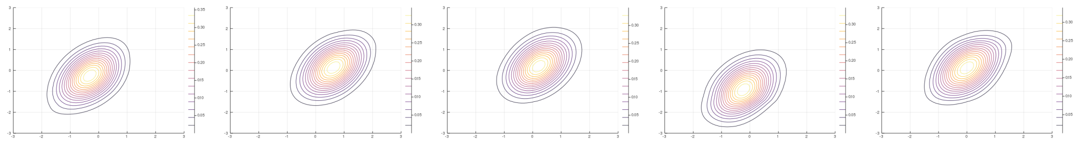
Table 4: (Affine coupling law) Exponentially smoothed estimates of the last 4000 (out of 5000) optimization steps. As can be seen in Figure 2, after the 1000th step is basically when the optimas are reached. Here the ELBO has been used as the objective.



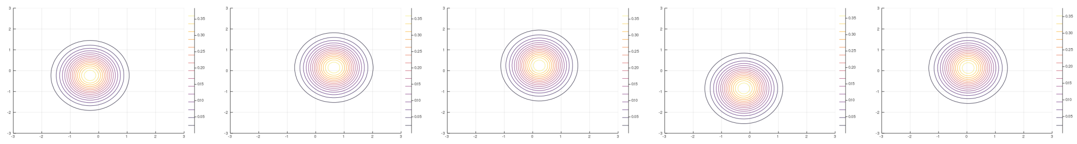
(a) Posterior



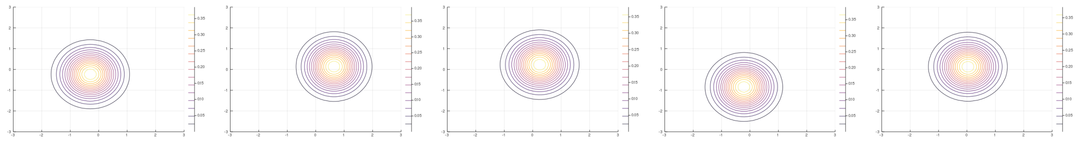
(b) NFVI (ELBO)



(c) NFVI (KL)



(d) MFVI (ELBO)



(e) MF (KL)

Figure 3: (Rational Quadratic Spline coupling law) Contour plots of the resulting densities for 5 different runs of the the experiment described in Appendix B.1. *(ELBO)* means the the density has been obtained through maximization of the ELBO, and *(KL)* means the density has been obtained through direct minimization of KL-divergence between the density and the corresponding true posterior from (a).

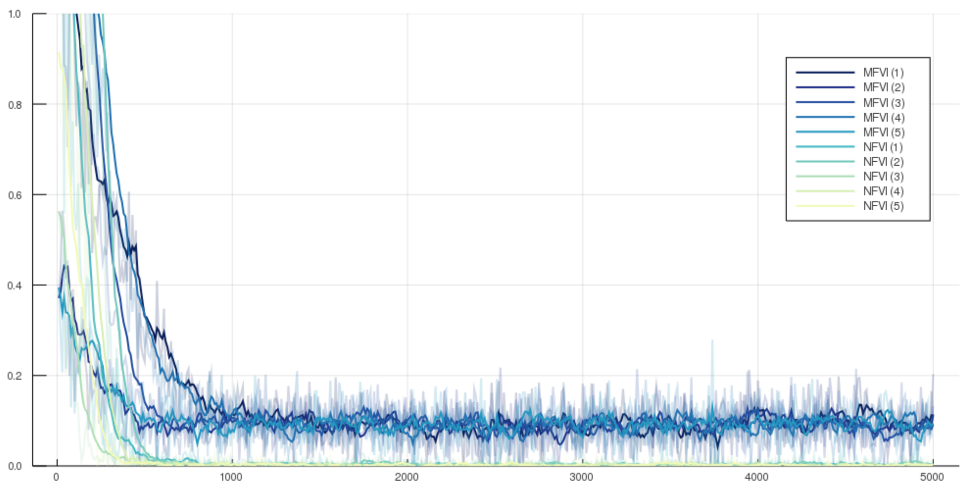
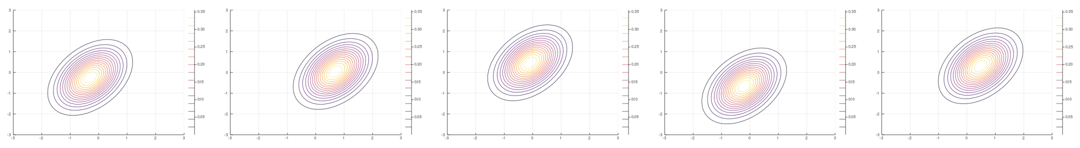
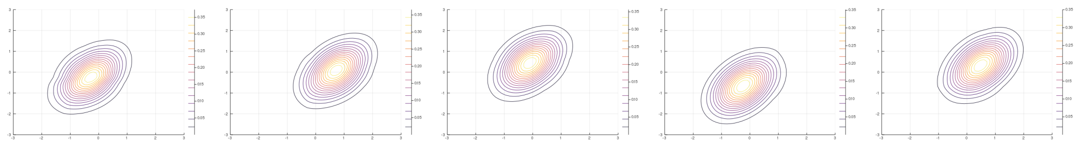
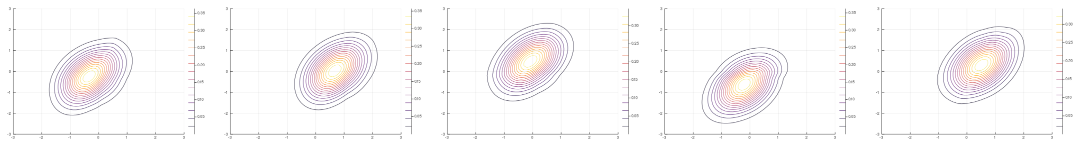
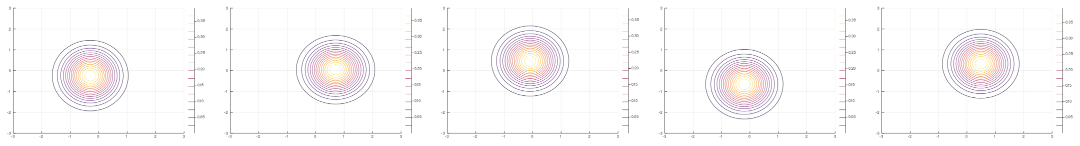


Figure 4: (Affine coupling law) KL-divergences of different runs where the objective used is the ELBO. The opaque lines represent the true samples while the non-opaque lines correspond to smoothed estimates using exponential smoothing.


(a) Posterior


(b) NFVI (ELBO)


(c) NFVI (KL)


(d) MFVI (ELBO)

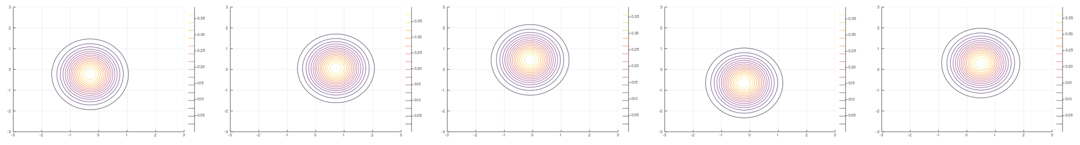

(e) MF (KL)

Figure 5: (Affine coupling law) Contour plots of the resulting densities for 5 different runs of the the experiment described in Appendix B.1. *(ELBO)* means the the density has been obtained through maximization of the ELBO, and *(KL)* means the density has been obtained through direct minimization of KL-divergence between the density and the corresponding true posterior from (a).

B.2. Kullback-Leibler (KL) divergence and the Evidence Lower Bound (ELBO)

Recall the definition of the *Kullback-Leibler (KL) divergence*, here relating the variational density $q(\boldsymbol{z})$ and posterior $p(\boldsymbol{z} \mid \{\boldsymbol{x}\}_{i=1}^{n})$,

$$\mathrm{D}_{\mathrm{KL}}\big(q(\boldsymbol{z}), p(\boldsymbol{z} \mid \{\boldsymbol{x}_i\}_{i=1}^{n})\big) = \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z})}\left[\log \frac{q(\boldsymbol{z})}{p(\boldsymbol{z} \mid \{\boldsymbol{x}_i\}_{i=1}^{n})}\right] \tag{5}$$

As per usual, we can rewrite this

$$\begin{aligned}
\mathrm{D}_{\mathrm{KL}}\big(q(\boldsymbol{z}), p(\boldsymbol{z} \mid \{\boldsymbol{x}_i\}_{i=1}^{n})\big) &= \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z})}\big[\log q(\boldsymbol{z})\big] - \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z})}\big[p(\boldsymbol{z} \mid \{\boldsymbol{x}_i\}_{i=1}^{n})\big] \\
&= \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z})}\big[\log q(\boldsymbol{z})\big] - \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z})}\big[p\big(\{\boldsymbol{x}_i\}_{i=1}^{n}, \boldsymbol{z}\big)\big] + \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z})}\big[\log p\big(\{\boldsymbol{x}_i\}_{i=1}^{n}\big)\big] \\
&= \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z})}\big[\log q(\boldsymbol{z})\big] - \sum_{i=1}^{n} \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z})}\big[p(\boldsymbol{x}_i, \boldsymbol{z})\big] + \sum_{i=1}^{n} \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z})}\big[\log p(\boldsymbol{x}_i)\big] \\
&= \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z})}\big[\log q(\boldsymbol{z})\big] - \sum_{i=1}^{n} \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z})}\big[p(\boldsymbol{x}_i, \boldsymbol{z})\big] + \sum_{i=1}^{n} \log p(\boldsymbol{x}_i)
\end{aligned}$$

where in the second-to-last equality we used the assumption that the observations are i.i.d. and in the last equality we used the fact that $\log p(\boldsymbol{x}_i)$ is independent of $\boldsymbol{z}$ for all $i = 1, \dots, n$. We can then arrange this into

$$\sum_{i=1}^{n} \log p(\boldsymbol{x}_i) = \mathrm{D}_{\mathrm{KL}}\big(q(\boldsymbol{z}), p(\boldsymbol{z} \mid \{\boldsymbol{x}_i\}_{i=1}^{n})\big) + \left(\sum_{i=1}^{n} \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z})}\big[p(\boldsymbol{x}_i, \boldsymbol{z})\big]\right) - \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z})}\big[\log q(\boldsymbol{z})\big]$$

Observe that given a set of observations, the left-hand side is *constant.* Therefore we can minimize the KL-divergence by *maximizing* the remaining terms on the right-hand side of the equation, which we call the *evidence lower bound (ELBO)*

$$\mathrm{ELBO}\big(q(z)\big) := \left(\sum_{i=1}^{n} \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z})}\big[p(\boldsymbol{x}_i, \boldsymbol{z})\big]\right) - \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z})}\big[\log q(\boldsymbol{z})\big] \tag{6}$$

Now suppose that the variational posterior $q(z)$ is in fact a *transformed* distribution, say, with base density q_0 and using transformation b, i.e.

$$q(\boldsymbol{z}) = q_0\big(b^{-1}(\boldsymbol{z})\big)|\det \mathcal{J}_{b^{-1}}(\boldsymbol{z})| \quad \text{or} \quad q\big(b(\boldsymbol{\eta})\big) = \frac{q_0(\boldsymbol{\eta})}{|\mathcal{J}_b(\boldsymbol{\eta})|}$$

By change of variables $z = b(\boldsymbol{\eta})$, the first term of the ELBO becomes

$$\begin{aligned}
\mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z})}\big[\log p(\boldsymbol{x}_i, \boldsymbol{z})\big] &= \int \log p(\boldsymbol{x}_i, \boldsymbol{z}) q(\boldsymbol{z}) \,\mathrm{d}z \\
&= \int \log p\big(\boldsymbol{x}_i, b(\boldsymbol{\eta})\big) q\big(b(\boldsymbol{\eta})\big)|\det \mathcal{J}_b(\boldsymbol{\eta})| \,\mathrm{d}\boldsymbol{\eta} \\
&= \int \log p\big(\boldsymbol{x}_i, b(\boldsymbol{\eta})\big) q_0(\boldsymbol{\eta}) \,\mathrm{d}\boldsymbol{\eta} \\
&= \mathbb{E}_{\boldsymbol{\eta}\sim q_0(\boldsymbol{\eta})}\big[\log p\big(\boldsymbol{x}_i, b(\boldsymbol{\eta})\big)\big]
\end{aligned}$$

Simililarily, the second term becomes

$$\begin{aligned}
\mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z})}\big[\log q(\boldsymbol{z})\big] &= \int q(\boldsymbol{z}) \log q(\boldsymbol{z}) \,\mathrm{d}\boldsymbol{z} \\
&= \int q\big(b(\boldsymbol{\eta})\big) |\det \mathcal{J}_b(\boldsymbol{\eta})| \log\Big(q\big(b(\boldsymbol{\eta})\big)\Big) \,\mathrm{d}\boldsymbol{\eta} \\
&= \int q_0(\boldsymbol{\eta}) \log\left(\frac{q_0(\boldsymbol{\eta})}{|\det \mathcal{J}_b(\boldsymbol{\eta})|}\right) \mathrm{d}\boldsymbol{\eta} \\
&= \int q_0(\boldsymbol{\eta}) \log q_0(\boldsymbol{\eta}) \,\mathrm{d}\boldsymbol{\eta} - \int q_0(\boldsymbol{\eta}) \log |\det \mathcal{J}_b(\boldsymbol{\eta})| \,\mathrm{d}\boldsymbol{\eta} \\
&= \mathbb{E}_{\boldsymbol{\eta}\sim q_0(\boldsymbol{\eta})}\big[\log q_0(\boldsymbol{\eta})\big] - \mathbb{E}_{\boldsymbol{\eta}\sim q_0(\boldsymbol{\eta})}\big[\log|\det \mathcal{J}_b(\boldsymbol{\eta})|\big] \\
&= -\mathbb{H}\big(q_0(\boldsymbol{\eta})\big) - \mathbb{E}_{\boldsymbol{\eta}\sim q_0(\boldsymbol{\eta})}\big[\log|\det \mathcal{J}_b(\boldsymbol{\eta})|\big]
\end{aligned}$$

Substituting these terms into the ELBO from Equation (6), we get

$$\mathrm{ELBO}\big(q(\boldsymbol{z})\big) = \mathbb{E}_{\boldsymbol{\eta}\sim q_0(\boldsymbol{\eta})}\left[\log|\det \mathcal{J}_b(\boldsymbol{\eta})| + \sum_{i=1}^{n} p\big(\boldsymbol{x}_i, b(\boldsymbol{\eta})\big)\right] + \mathbb{H}\big(q_0(\boldsymbol{\eta})\big) \tag{7}$$

This expression ise very useful when q_0 is a density which it is computationally cheap to sample from and we have an analytical expression for the entropy of q_0, e.g. if q_0 is the density of a mulitvariate Gaussian both of these conditions are satisfied. In practice we use a Monte-Carlo estimate of the ELBO

$$\widehat{\mathrm{ELBO}}\big(q(\boldsymbol{z})\big) = \frac{1}{m}\sum_{k=1}^{m}\left[\log|\det \mathcal{J}_b(\boldsymbol{\eta}_k)| + \sum_{i=1}^{n} p\big(\boldsymbol{x}_i, b(\boldsymbol{\eta}_k)\big)\right] + \mathbb{H}\big(q_0(\boldsymbol{\eta})\big) \tag{8}$$

where $\boldsymbol{\eta}_k \sim q_0(\boldsymbol{\eta})$ for $k = 1, \dots, m$. From this we can then obtain a Monte-Carlo estimate of the gradient wrt. parameters.