Supplementary Material

7.1 Further line search figures

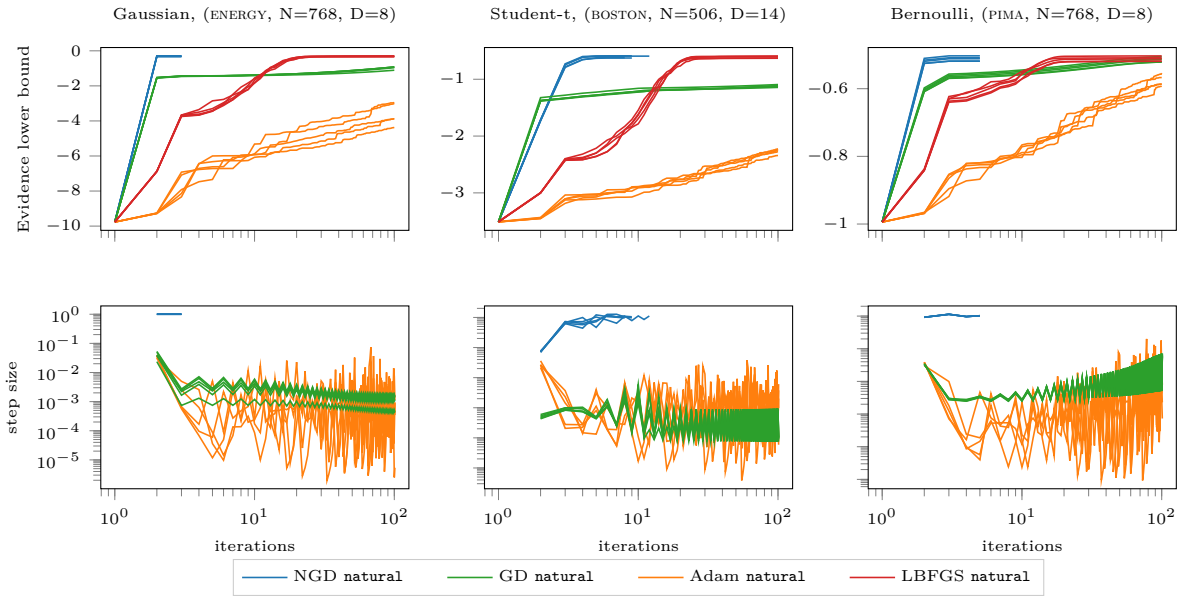

Figure 8: Line search in the `natural` parameterization across 5 splits.

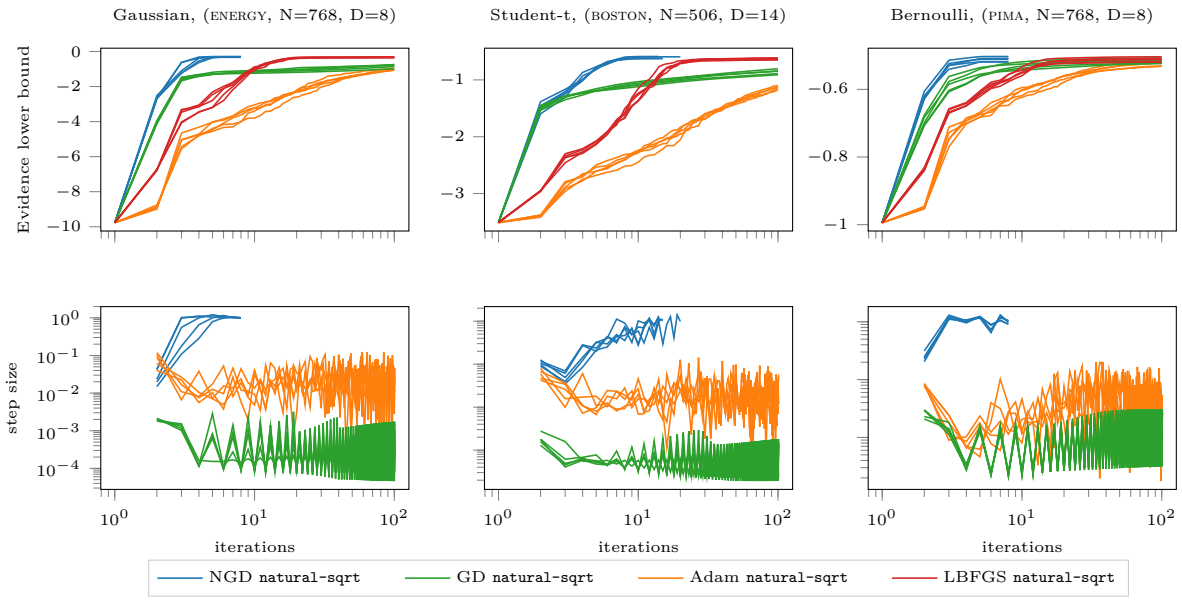

Figure 9: Line search in the `natural-sqrt` parameterization across 5 splits.

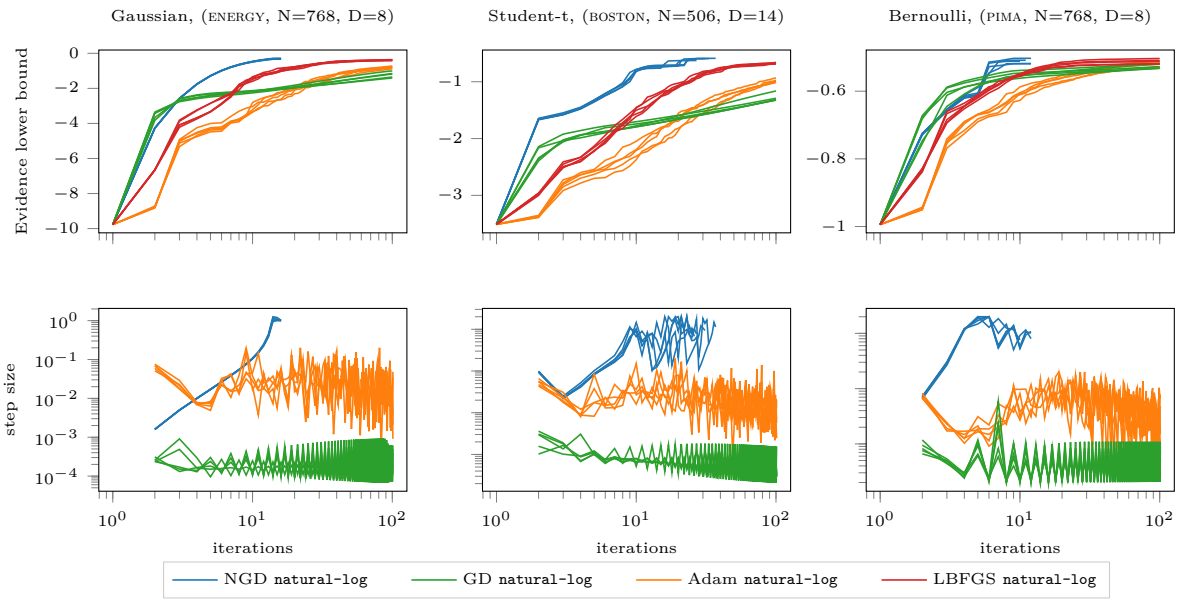

Figure 10: Line search in the `natural-log` parameterization across 5 splits.

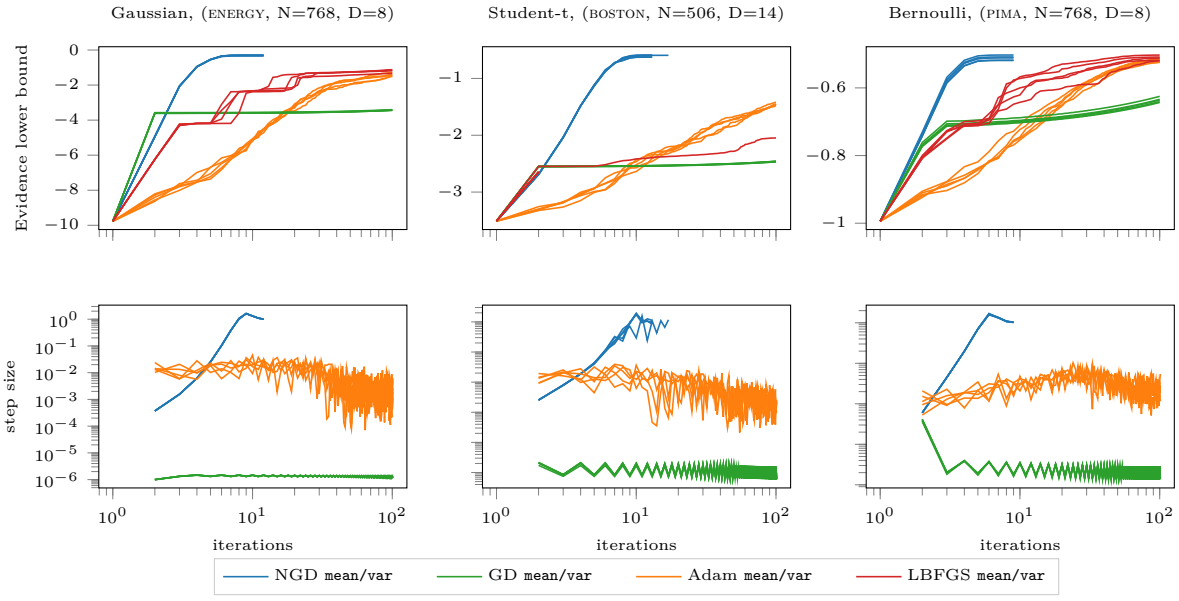

Figure 11: Line search in the `mean/var` parameterization across 5 splits.

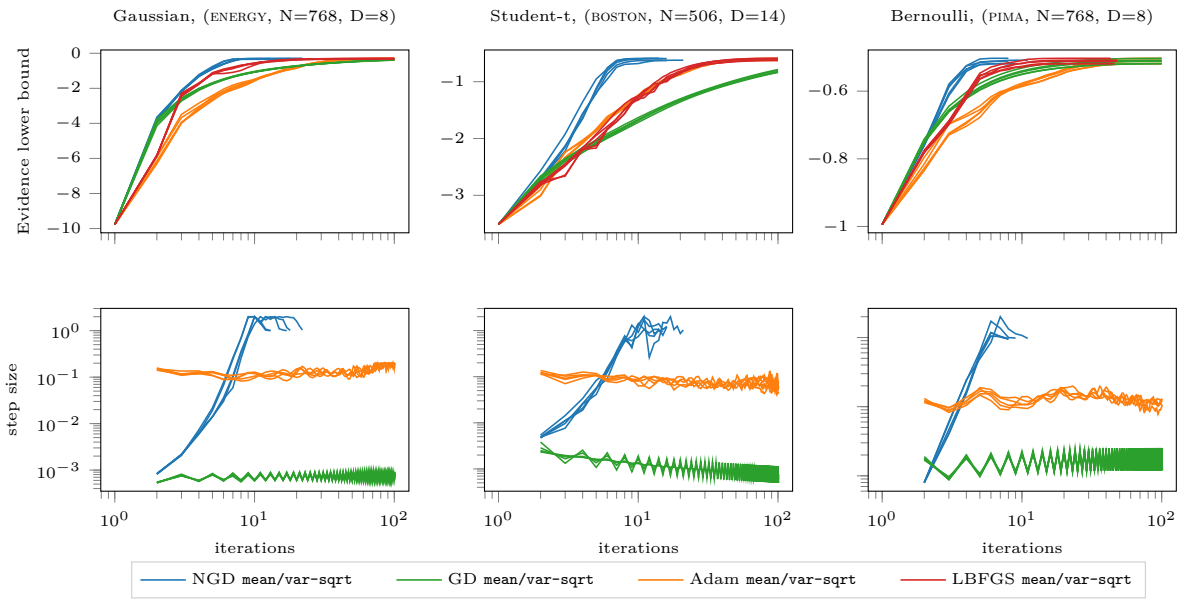

Figure 12: Line search in the `mean/var-sqrt` parameterization across 5 splits.

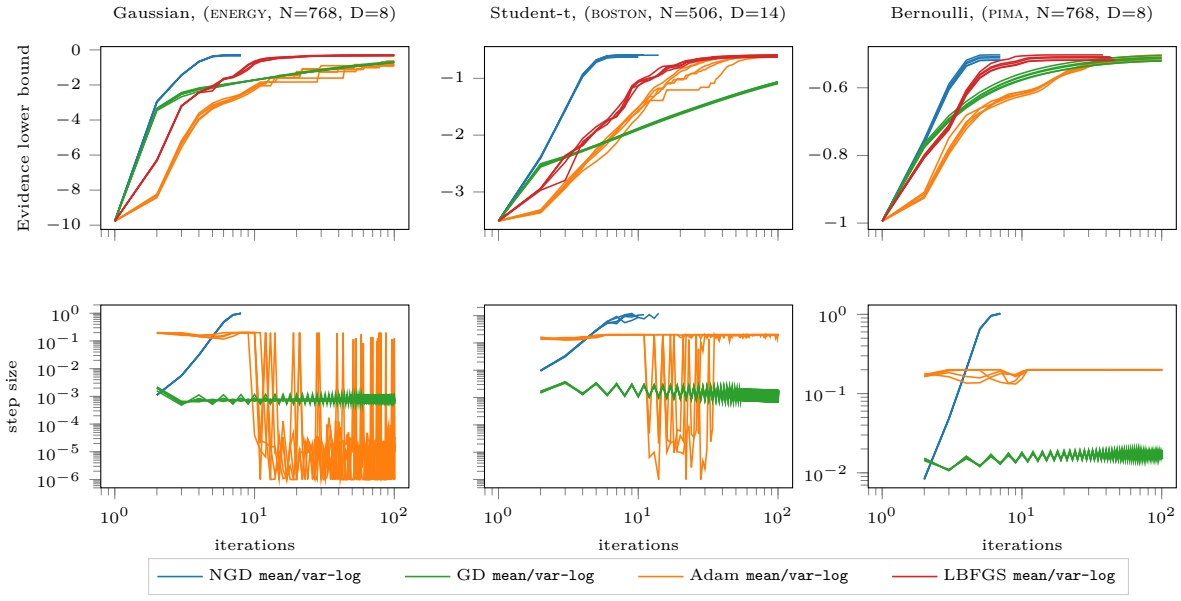

Figure 13: Line search in the `mean/var-log` parameterization across 5 splits.

7.2 Experimental details

Implementation. All experiments were run on a single desktop machine with a GTX 1070 GPU. The code was written in GPflow (Matthews et al., 2017), a GP library built on tensorflow.

Kernel. For all experiments we used the Matern $\frac{5}{2}$ kernel, with the (single) lengthscale initialized to the square root of the data dimension. The kernel variance was initalized to 2, except for MNIST, where we initialized to 10.

Inducing points. We used 100 inducing points, initialized with k-means. The variational parameters were intialized to mean zero and identity covariance.

Jitter. We used a small jitter level of 10^{-10} for all experiments.

Data normalization. For all datasets apart from MNIST we scaled the inputs to have zero mean and unit variance in the training data. We applied the same scaling to the test data. For MNIST we used the standard scaling to the unit interval.

For the Gaussian and student-t likelihoods we scaled the outputs to have zero mean and unit standard deviation in the training data. The beta and ordinal likelihood are described in the main text.

7.3 The forward-mode trick

We describe how to obtain a forward-mode derivative using a reverse-mode library. The trick is due to Townsend et al. (2017) and this explanation closely follows `https://j-towns.github.io/2017/06/12/A-new-trick.html`.

Reverse-mode differentiation is the successive application of the vector-Jacobian product (vjp) operation. The vjp operation left multiplies a vector $\mathbf{u}$ with the Jacobian of $\mathbf{f}$ with respect to its input $\mathbf{x}$:

$$\text{vjp}(\mathbf{f}, \mathbf{x}, \mathbf{u}) = \mathbf{u}^\top \frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \sum_i u_i \frac{\partial f_i}{\partial \mathbf{x}} \,.$$

The vjp operation can be used to implement the gradient of a function $L(\mathbf{f}(\mathbf{g}(\mathbf{x})))$ by using the chain rule $\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{f}} \frac{\partial \mathbf{f}}{\partial \mathbf{g}} \frac{\partial \mathbf{g}}{\partial \mathbf{x}}$ and successively applying the vjp operation from left to right, i.e.,

$$\begin{aligned} \mathbf{u} &= \text{vjp}(L, \mathbf{f}, 1) \,, \\ \mathbf{u} &\leftarrow \text{vjp}(\mathbf{f}, \mathbf{g}, \mathbf{u}) \,, \\ \mathbf{u} &\leftarrow \text{vjp}(\mathbf{g}, \mathbf{x}, \mathbf{u}) \,. \end{aligned}$$

After these operations $\mathbf{u} = \frac{\partial L}{\partial \mathbf{x}}$. Automatic reverse-mode differentiation libraries implement vjp for all basic operations they support. Compositions of basic operations can be computed as above. Note that the values of $\mathbf{f}$ and $\mathbf{g}$ need to be computed first, which requires a forward pass through the function.

Forward-mode differentiation makes use of a Jacobian-vector product operation (jvp), defined as

$$\text{jvp}(\mathbf{f}, \mathbf{x}, \mathbf{u}) = \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \mathbf{u} = \sum_i \frac{\partial \mathbf{f}}{\partial x_i} u_i \,.$$

Using the jvp operation, the chain rule can be implemented by successive application of jvp, working from right to left, i.e.,

$$\begin{aligned} \mathbf{u} &= \text{jvp}(\mathbf{g}, \mathbf{x}, \mathbf{1}) \,, \\ \mathbf{u} &\leftarrow \text{jvp}(\mathbf{f}, \mathbf{g}, \mathbf{u}) \,, \\ \mathbf{u} &\leftarrow \text{jvp}(L, \mathbf{f}, \mathbf{u}) \,, \end{aligned}$$

where $\mathbf{1}$ is a vector of ones with the same shape as $\mathbf{x}$.

To implement natural gradients in any parameterization we require the jvp operation, but common libraries such as Tensorflow implement only vjp (i.e. reverse mode). The trick to achieve jvp from vjp is to introduce a dummy

variable $\mathbf{v}$ and define $\mathbf{g}(\mathbf{v}) = \text{vjp}(\mathbf{f}, \mathbf{x}, \mathbf{v})$. We then use vjp again to find the gradient of $\mathbf{g}$ with respect to $\mathbf{v}$, passing in the vector $\mathbf{u}$ to be pushed forward: $\text{vjp}(\mathbf{g}, \mathbf{v}, \mathbf{u})$. Since $\mathbf{g}$ is linear in $\mathbf{v}$, we have

$$\text{vjp}(\mathbf{g}, \mathbf{v}, \mathbf{u}) = \mathbf{u}^\top \frac{\partial}{\partial \mathbf{v}} \left(\mathbf{v}^\top \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \right) = \mathbf{u}^\top \left(\frac{\partial \mathbf{f}}{\partial \mathbf{x}} \right)^\top .$$

This is exactly the transpose of $\text{jvp}(\mathbf{f}, \mathbf{x}, \mathbf{u})$. Therefore, any reverse-mode differentiation library can be used to compute forward-mode derivatives.