EiNSTeIN VI: General and Integrated Stein Variational Inference in NumPyro

Anonymous authors
Paper under double-blind review

ABSTRACT

Stein Variational Inference is a technique for approximate Bayesian inference that is gaining popularity since it combines the scalability of traditional Variational Inference (VI) with the flexibility of non-parametric particle based inference methods. While there has been considerable progress in development of algorithms, integration in existing probabilistic programming languages (PPLs) with an easy-to-use interface is currently lacking. EinStein VI is a lightweight composable library that integrates the latest Stein Variational Inference methods with the NumPyro PPL. Inference with EinStein VI relies on ELBO-within-Stein to support use of custom inference programs (guides), non-linear scaling of repulsion force, second-order gradient updates using matrix-valued kernels and parameter transforms. We demonstrate the achieved synergy of the different Stein techniques and the versatility of the EinStein VI library by applying it on examples. Compared to traditional Stochastic VI, EinStein VI is better at capturing uncertainty and representing richer posteriors. We use several applications to show how one can use Neural Transforms (NeuTra) and second-order optimization to provide better inference using EinStein VI. We show how EinStein VI can be used to infer novel Stein Mixture versions of realistic models. We infer the parameters of a Stein Mixture Latent Dirichlet Allocation model (SM-LDA) with a neural guide. The results indicate that Einstein VI can be combined with NumPyro’s support for automatic marginalization to do inference over models with discrete latent variables. Finally, we introduce an example with a novel Stein Mixture extension to Deep Markov Models, called the Stein Mixture Deep Markov Model (SM-DMM), which shows that EinStein VI can be scaled to reasonably large models with over 500,000 parameters.

1 I N T R O D U C T I O N

Interest in Bayesian deep learning has surged due to the need for quantifying the uncertainty of predictions provided by machine learning algorithms. The idea behind Bayesian learning is to describe observed data $x$ using a model with latent variable $z$ (representing model parameters and nuisance variables, see e.g., Fig. 4a). The goal is then to infer a posterior distribution $p(z|x)$ over latent variables given a model describing the joint distribution $p(z, x) = p(x|z)p(z)$ following the rules of Bayesian inference:

$$p(z|x) = Z^{-1}p(x|z)p(z)$$

where the normalization constant $Z = \int p(x|z)p(z)dz$ is intractable for most practical models including deep neural networks: an analytic solution is lacking or may require an infeasible number of calculations.

Variational Inference (VI) techniques [Blei et al. 2017, Hoffman et al. 2013, Ranganath et al. 2014] provide a way to find an approximation of the posterior distribution. VI poses a family of distributions over latent variables $q(z) \in Q$ (e.g., Fig. 4b) and chooses the one that minimizes a chosen divergence $D(q(z) \| p(z|x))$ (e.g., Kullback-Leibler) to the true posterior distribution. VI often provides good approximations that can capture uncertainty, scaling to millions of data points using mini-batch training.

1 Asymmetric distance
def model(X, y=None):
    Wl = np.zeros((4, 3))
    Ws = np.ones((4, 3))
    bl = np.zeros(3)
    bs = np.ones(3)
    W = sample('W1', Normal(Wl, Ws))
    b = sample('b1', Normal(bl, bs))
    probs = softmax(X @ W + b)
    with plate('data', X.shape[0]):
        sample('y', Categorical(probs), obs=y)

(a) Probabilistic linear regression model

data = load_iris()
X_train, X_test, y_train, y_test = train_test_split(data.data, data.target)
stein = Stein(model, WrappedGuide(guide), Adam(0.1), ELBO(),
               RBFKernel(), num_particles=100)
state, loss = stein.train(rng_key, 15000, X_train, y_train,
                          callbacks=[Progbar()])

(c) Running EinStein VI to learn parameters using Stein particles {Wₖ,i, Wₛ,i, bₖ,i, bₛ,i}₁₀₀ᵢ=₁
y_pred = stein.predict(state, X_test)['y']
print(f"Accuracy: {np.mean(y_pred == y_test)*100:.2f} %")

(d) Using learned parameters from EinStein VI to predict values of test data

Stein Variational Inference (Liu and Wang, 2016) is a recent non-parametric approach to VI which
uses a set of particles \{zᵢ\}_{i=1}^{N} as the approximating distribution \(q(z)\) to provide better flexibility in
capturing correlations between latent variables. The technique preserves the scalability of traditional
VI approaches while offering the flexibility and modelling scope of techniques such as Markov Chain
Monte Carlo (MCMC). Stein VI has been shown to be good at capturing multi-modality (Liu and
Wang, 2016; Wang and Liu, 2019a), and has useful theoretical interpretations as particles following a
gradient flow (Liu, 2017) and as a moment matching optimization system (Liu and Wang, 2018).

Many advanced inference methods based on Stein VI have been recently developed, including
Stein mixtures (Nalisnick, 2019), non-linear Stein (Wang and Liu, 2019b), factorized graphical
models (Zhuo et al., 2018; Wang et al., 2018b), matrix-valued kernels (Wang et al., 2019) and support
for higher-order gradient-based optimization (Detommaso et al., 2018). These techniques have
been shown to significantly extend the power of Stein VI, allowing more flexible and effective
approximations of the true posterior distribution. While algorithmic power is growing, there remains
a distinct lack of integration of these techniques into a general probabilistic programming language
(PPL) framework. Such an integration would solve one of the most prominent limitations of traditional
VI, which lacks flexibility to capture rich correlations in the approximated posterior.

This paper presents the EinStein VI library that extends the NumPyro PPL (Bingham et al., 2019;
Phan et al., 2019) with support for the recent developments in Stein Variational Inference in an
integrated and compositional fashion (see Fig. 1c and Fig. 1d). The library takes advantage of the
capabilities of NumPyro—including universal probabilistic programming (van de Meent et al., 2018),
integration with deep learning using JAX (Frostig et al., 2018), and automatic optimization and
marginalization of discrete latent variables (Obermeyer et al., 2019)—to provide capabilities that
work synergetically with the Stein algorithms. Concretely, our contributions are:
Under review as a conference paper at ICLR 2020

• Einstein VI, a general library that extends NumPyro and allows Stein Variational Inference to work with custom guide programs based on ELBO-within-Stein optimization (Nalisnick 2019). The library is compositional with NumPyro features, including supported deep learning and automatic marginalization, loss functions (ELBO, Rényi ELBO, custom losses) and optimization methods, allowing it to grow organically with NumPyro development.

• Integration of recent developments in Stein variational inference within the EinStein library. This includes support for non-linear optimization (Wang and Liu 2019b), a wealth of kernels (Liu and Wang 2016, 2018; Gorham and Mackey 2017), matrix-valued kernels (Wang et al., 2019) supporting higher-order optimization, and factorization based on conditional independence between elements in the model (graphical kernels).

• We support the application of transforms on the parameter space, including triangular (Parno and Marzouk 2018) and neural transforms (Hoffman et al., 2018) to improve the gradient geometry of the inference problem.

• A series of examples demonstrate the power of an integrated library such as EinStein VI and the synergy between different Stein VI techniques. The examples include a novel Stein Mixture version of Deep Markov Models (SM-DMM), Stein Mixture Latent Dirichlet Allocation (SM-LDA), and several examples using neural transforms and higher-order optimization.

The paper proceeds as follows. We first present a primer on the theory of Stein VI in Section 2 relating it to our integrated implementation in EinStein VI. We discuss the general details of the implementation of EinStein VI in NumPyro in Section 3. We present the various examples using EinStein VI in Section 4 and finally summarize our results and future work in Section 5.

1.1 RELATED WORK

There has been a proliferation of deep probabilistic programming languages based on tensor frameworks with automatic differentiation, supporting various inference techniques.

Pyro (Bingham et al., 2019) is a universal PPL based on PyTorch (Paszke et al., 2019). The main mode of inference in Pyro is black-box Stochastic Variational Inference (Ranganath et al., 2014) with guides, which are flexible programs that approximate the posterior distribution by repeating the (non-observed) sample statements as the probabilistic model and can contain deep neural networks allowing amortization of inference (Kingma and Welling 2014; Gershman and Goodman 2014). Pyro also supports various sampling algorithms like Hamiltonian Monte Carlo (Neal 2011; Hoffman and Gelman 2014) and sample-adaptive MCMC (Zhu 2019) which provide more accurate posterior approximations but lack the scalability of VI techniques. NumPyro is a version of Pyro that runs on the JAX framework (Frostig et al., 2018), which allows it to exploit the powerful program optimization and parallelizability of the JAX compiler. Our library, EinStein VI, extends the Stochastic VI mode of inference by adding Stein Variational Inference on top of NumPyro, allowing the optimizable parameters in the guide to be approximated by a set of particles instead of a point estimate. Other languages with similar feature set include PyMC3 (Salvatier et al., 2016), Edward (Tran et al., 2016, 2019) and HackPPL (Ai et al., 2019).

2 PRIMER ON STEIN VI

The core idea of Stein VI (Liu and Wang, 2016) is to perform inference by approximating the target posterior distribution \( p(z|x) \) by an approximate distribution \( q_Z(z) = N^{-1} \sum_{i=1}^{N} \delta_{z_i}(z) \) based on a set of particles \( Z = \{z_i\}_{i=1}^{N} \). Here, \( \delta_{z}(y) \) represents the Dirac delta measure, which is equal to 1 if \( x = y \) and 0 otherwise. One could thus see the approximating distribution \( q_Z(z) \) as a mixture of point estimates, each represented by a particle \( z_i \in Z \). The idea is then that we minimize the Kullback-Leibler divergence \( D_{KL}(q(z) \parallel p(z|x)) \) between the approximation and the true posterior by iteratively updating the particles using the Stein forces:

\[
z_i \leftarrow z_i + \epsilon S_Z(z_i)
\]

where \( \epsilon \) represents the learning rate and \( S_Z \) the Stein forces.
The Two Forces of Stein VI  Stein VI consists of two forces which work additively under the form
\[ S_Z = S_Z^+ + S_Z^-, \]
where the attractive force
\[ S_Z^+(z_i) = \mathbb{E}_{z_j \sim q_Z(z)} [k(z_j, z_i) \nabla_{z_i} \log p(z_i | x)] \]
and the repulsive force
\[ S_Z^-(z_i) = \mathbb{E}_{z_j \sim q_Z(z)} [\nabla_{z_i} k(z_j, z_i)]. \]
Here \( k : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R} \) is a kernel. The attractive force can be seen as pushing the particles towards the direction that maximizes the true posterior distribution, smoothed by some kernel. For an example of a kernel, consider the standard RBF kernel \( k(z_i, z_j) = \exp \left( -\frac{1}{\theta} \| z_i - z_j \|^2 \right) \) with bandwidth parameter \( \theta \), usually chosen as \( \frac{1}{\log n} \text{med}(z) \). The normalization constant becomes additive in the log-posterior \( \log p(z_i | x) = -\log Z + \log p(x | z) + \log p(z) \) and so does not need to be computed for the gradient.

The repulsive force can be seen as moving particles away from each other, ensuring that they do not all collapse to the same mode. For the RBF kernel, the repulsive force becomes \( \mathbb{E}_{z_j \sim q_Z(z)} [k(z_i, z_j) \sum_k (z_i^k - z_j^k)] \) and so particles that are close together (thus having a high kernel value) will be pushed away from each other.

Non-linear Stein  In non-linear Stein (Wang and Liu 2019a), the repulsive force can be scaled by a factor \( \lambda \), so \( S_Z = S_Z^+ + \lambda S_Z^- \) which is often useful when dealing with multi-modal distributions. It is also useful in our framework since the repulsive force often vanishes compared to the likelihood for large datasets \( X \) and therefore scaling the repulsive force by a constant \( \lambda = \lambda(|X|) \) proportional to the size of the dataset \( |X| \) appears logical, e.g. 0.1 or 0.01.

Matrix-valued kernels  The choice of kernels can be extended to matrix-valued ones (Wang et al. 2019) \( K : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}^{d \times d} \) in which case the Stein forces become
\[ S_Z^+(z_i) = \mathbb{E}_{z_j \sim q_Z(z)} [K(z_j, z_i) \nabla_{z_i} \log p(z_i | x)] \]
and
\[ S_Z^-(z_i) = \mathbb{E}_{z_j \sim q_Z(z)} [K(z_j, z_i) \nabla_{z_i}] \]
where the standalone del \( \nabla_{z_i} \) in the repulsive force represents the vector \( \left( \frac{\partial}{\partial z_{i,1}}, \ldots, \frac{\partial}{\partial z_{i,d}} \right) \) and so \( (K(z_j, z_i) \nabla_{z_i})_\ell = \sum_k \nabla K_{\ell,k}(z_j, z_i) \). The advantage of matrix-valued kernels is that they allow preconditioning using the Hessian or Fisher Information matrix (second-order derivatives), which can capture local curvature and thus achieve better optima than standard Stein VI. Furthermore, it is easy to capture graphical kernels (Wang et al. 2018b) \( K = \text{diag}(\{K^{(i)}\}_i) \) where the set of variables are partitioned with each their own local kernel \( K^{(i)} \).

ELBO-within-Stein  The standard form of Stein VI guides the particles toward maximizing the log-posterior \( \log p(z | x) \) but in principle we could replace this with another objective (negative loss) \( \mathcal{L}_\theta \) which we want to maximize. Assume our guide \( q_\theta(z) \) comes from a parametric distribution with parameters \( \theta \), as in traditional VI. We can then for example maximize the ELBO (Kingma and Welling 2014) \( \mathcal{L}_\theta = \mathbb{E}_{z \sim q_\theta(z)} [\log p(z, x) - \log q_\theta(z)] \) as in traditional VI but using a set of particles for the parameters \( \{ \theta_j \} \) guided by the Stein forces. One can see this as a generalization of traditional VI using a Stein-based mixture-model (Nalisnick, 2019), giving the parameters the flexibility to better capture correlations between distributions and to cover multiple modes (Wang and Liu 2019a). Another perspective is that ELBO-within-Stein adds uncertainty to the approximating distribution \( q_\theta(z) \), allowing it to work with distributions over latent variables that are richer than simply point masses.

The objective to be maximized in our framework is compositional, which means that ELBO can be replaced by other objectives, e.g. Rényi ELBO (Li and Turner 2016), Tail-adaptive f-divergence (Wang et al. 2018a) or Wasserstein pseudo-divergence (Ambrogioni et al. 2018) when these are implemented in NumPyro. Einstein VI makes it possible to integrate all these developments within Stein VI in a coherent fashion.
Parameter Transforms  Deterministic transport maps have been suggested to be integrated in MCMC methods by [Parno and Marzouk, 2018] and [Hoffman et al., 2018], to allow constructing a better geometry for the posterior approximation and thus improve inference. We provide a novel adaptation of the idea to work with Stein-based methods, in the EinStein VI framework.

Consider the general objective we optimize in EinStein VI, which has the form:

\[ S_\Theta(\theta_i) = \mathbb{E}_{\theta_j \sim q(\theta_j)}[k(\theta_j, \theta_i) \nabla_{\theta_i} \mathcal{L}_\theta + \nabla_{\theta_i} k(\theta_j, \theta_i)] \]

where \( \mathcal{L}_\theta \) is our negative loss objective (e.g., ELBO) parameterised by \( \theta \) which we update using a set of Stein particles \( \Theta = \{ \theta_i \} \).

The core idea is then that we would like to reparametrize the calculation so each Stein particle \( \theta_i \) is expressed as an (invertible) deterministic transformation of another particle \( \phi_i \), so \( \theta_i = T(\phi_i) \) (thus \( \Theta = \{ T(\phi) | \phi \in \Phi \} \)).

We do this reparametrization by first declaring the set of particles to be optimized \( \Phi = \{ \phi_i \} \) and the transformation to be used \( T \), and then define the reparametrized Stein force as follows:

\[ S_\Phi(\phi_i) = S_\Theta(T(\phi_i)) (\nabla_{\phi_i} T(\phi_i))^\top \]

where \( \nabla_{\phi_i} T(\phi_i) : \mathbb{R}^{d \times e} \) is the reparametrization Jacobian, that adjusts for the change of variables (from \( \theta_i : \mathbb{R}^e \) to \( \phi_i : \mathbb{R}^d \)).

The power of parameter transforms is amplified in our EinStein VI framework, in that we allow for its parameters to be learnable. We can thus use e.g., sparse triangular maps, invertible neural networks or normalizing flows, which have been shown to be particularly effective for Bayesian inference.

3  COMPOSITIONAL IMPLEMENTATION USING NUMPYRO

The system integrates well with the existing NumPyro API and programs using the standard Stochastic Variational Inference (SVI) interface can easily be converted to use EinStein VI instead. Fig. 1c shows the core usage of the EinStein VI interface; like the standard SVI interface it accepts a model, guide, optimizer and loss function. Additionally, EinStein VI accepts a kernel interface (e.g., RBFKernel) which is used to guide the Stein particles and to specify the number of Stein particles to use for inference.

We will discuss the major features of the implementation of Einstein VI, including allowing for re-initializable guides, the core algorithm that integrates Stein VI into NumPyro, and the compositional kernel interface.

3.1 RE-INITIALIZABLE GUIDES

The Stein VI interface requires that inference programs can be re-initialized with different values for each parameter. The reason is that we would like different Stein particles to be initialized differently in order for optimization to work correctly and avoid all particles collapsing into the posterior mode.

To support re-initializable guides we provide the ReinitGuide interface which requires implementing a function find_params that accepts a list of random number generator (RNG) keys in addition to the arguments for the guide and returns a set of freshly initialized parameters for each RNG key.

Most guides in NumPyro are written as functions as in Fig. 4b and to support those, we provide the WrappedGuide class which makes a callable guide re-initializable. works by running the provided guide multiple times, and reinitializing the parameters using NumPyro’s interface as follows:

- WrappedGuide runs the guide transforming each parameter to unconstrained space.
- It replaces the values of the parameters with values provided by a NumPyro initialization strategy, e.g., init_to_uniform\( (r) \) which initializes each parameter with a uniform random value in the range \([-r; r]\).
- It saves the parameter values for each particle and the required inverse transformations to constrained space, in order to allow running the model correctly.
We also allow parameters without reinitialization, e.g., for \texttt{stax} neural network\footnote{https://jax.readthedocs.io/en/latest/jax.experimental.stax.html} which have their own initializers.

### 3.2 EinStein VI in NumPyro

The integration of Stein VI into the deep PPL NumPyro requires handling transformations between the parameter representation of NumPyro—which is a dictionary mapping parameters to their values, which can be arbitrary Python types—to vectorized Stein particles that can be used in EinStein VI. For this, we rely on NumPyro’s integration with JAX PyTrees\footnote{https://jax.readthedocs.io/en/latest/notebooks/JAX_pytrees.html} which can convert back and forth between Python collections and a flattened vectorized representation. For example, the parameters \{\(W_{i,i}, W_{a,i}, b_{i,i}, b_{a,i}\)\}, from Fig. 1 will be encoded into particles \{\(a_i\)\}, where each particle is a 30-dimensional vector \(a_i \in \mathbb{R}^{30}\).

Alg. 1 shows the core algorithm of EinStein VI. Classical variational parameters \(\phi\) and \(\phi'\) are updated by averaging the loss over Stein particles. For Stein parameters, the process is more elaborate. First, we convert the set of individual parameters to set of vector-encoded particles using JAX PyTrees. Then we compute a kernel based on the vector-encoded Stein particles; the computation is kernel-dependent but an example is the RBF kernel which requires a bandwidth parameter \(h\) that is dependent on the particle values\cite{Liu2016}. We apply JAX’s \texttt{vmap} operator\cite{Frostig2018,Phan2019} to parallelly compute the Stein forces for each particle in a vectorized manner. Alg. 2 presents how EinStein VI works with regular-valued kernels but the actual implementation is more general, since it takes into account the different features presented in Section 2: non-linear repulsion force, potential use of matrix-valued kernels, and parameter transforms. Finally, we convert the monolithic Stein particle back to their non-vectorized dictionary-based form and return the expected changes for both classical and Stein parameters.

**Algorithm 1** EinStein VI

**Input:** Classical parameters \(\phi\) and \(\phi'\), Stein parameters \{\(\theta_i\)\}, model \(p_{\phi}(z, \mathbf{x})\), guide \(q_{\theta, \phi'}(z)\), loss \(\mathcal{L}\), kernel interface \(K\).

**Output:** Parameter changes based on classical VI \((\Delta \phi, \Delta \phi')\) and Stein VI forces \((\{\Delta \theta_i\})\).

\[
\Delta \phi &\leftarrow \mathbb{E}_\theta[\nabla_\phi \mathcal{L}(p_{\phi}, q_{\theta, \phi'})] \\
\Delta \phi' &\leftarrow \mathbb{E}_\theta[\nabla_{\phi'} \mathcal{L}(p_{\phi}, q_{\theta, \phi'})] \\
\{a_i\}_i &\leftarrow \text{PYTREEFLATTEN} \{(\theta_i)_i\} \\
k &\leftarrow K(\{a_i\}_i) \\
\{\Delta a_i\}_i &\leftarrow \text{VMAP} \{a_i\}_i, a_i \mapsto \sum_j k(a_j, a_i) \nabla_a \mathcal{L}(p_{\phi}, q_{\text{PYTREESTORE}(a_i), \phi'}) + \nabla_a k(a_j, a_i) \\
\{\Delta \theta_i\}_i &\leftarrow \text{PYTREESTORE} \{\Delta a_i\}_i \\
\text{return} & \Delta \phi, \Delta \phi', \{\Delta \theta_i\}_i
\]

### 3.3 Kernel Interface

The Kernel interface \texttt{SteinKernel} is reasonably simple. Users must implement the \texttt{compute} function, which accepts as input the current set of particles, the mapping between model parameters and particles, and the loss function \(\mathcal{L}\) and returns a differentiable kernel \(k\). Currently, all kernels are disallowed from changing state, but one could imagine a stateful interface in the future which relies on the history of particles, e.g., for computing conjugate gradients or quasi-Newton optimization. The list of implemented kernels in EinStein VI are described further in App. \footnote{https://github.com/aleatory-science/numpyro}

### 4 Examples

We demonstrate the flexibility of EinStein VI by its application to two standard pathological examples (the Double Moon and Neal’s Funnel), Stein Mixture Latent Dirichlet Allocation (SM-LDA), and Stein Mixture Deep Markov Model (SM-DMM). All experimental code, and the EinStein package are available at \url{https://github.com/aleatory-science/numpyro}. We plan to integrate our work upstream to the main NumPyro repository.
**Neal’s funnel**  We first consider a toy model where MCMC methods often struggle to sample the true posterior. The model was first described in [Neal (2003)](https://www.cs.toronto.edu/~riordan/papers/Neal03.pdf), and is commonly known as Neal’s funnel. The joint posterior is

\[
y \sim \mathcal{N}(0, 3) \quad x \sim \mathcal{N}(0, e^{y^2})
\]

which becomes difficult to sample as \(y\) grows increasingly negative. Figure 2 shows that EinStein VI captures the funnel well, and the particles are concentrated around the funnel area as expected.

**Dual Moon**  The Dual Moon example is a classical model with multiple modes, which requires a rich family of variational distributions (guides) to provide a good approximation of the posterior. It thus serves as a good example of how integration of parameter transforms in EinStein VI.

In EinStein VI, each particle represents a variational parameter. Transformation of variables of the approximating distribution causes all particles to collapse at modes of the distribution. This is because the repulsive force is applied in the preimage of the transformation (e.g. Inverse Autoregressive flow, Planar flow, or Radial flow). To remedy this we apply Stein in the transformed space instead of using it to transform our approximation distribution. Figure 4 shows how to use neural flows for inference.

**Stein Mixture Latent Dirichlet Allocation**  Latent Dirichlet Allocation ([Blei et al., 2003](https://people.cs.umass.edu/~blei/papers/papers.html)) is a model that seeks to infer a distribution over topics \(\theta\), that represents a set of size \(N\) of related words, from a collection of documents, each represented as a bag of words \(w\). Each document is generated according to the following model:

\[
\begin{align*}
\theta & \sim \text{Dir}(\alpha) \\
z_n & \overset{iid}{\sim} \text{Cat}(\theta) \\
w_n & \overset{iid}{\sim} \text{Cat}(\varphi_{z_n}) \\
n & \in \{1..N\}
\end{align*}
\]

where the number of topics \(T = |\theta|\) is fixed, \(\alpha\) is a scalar that represents prior knowledge on the symmetric Dirichlet distribution, typically \(\alpha = 1/T\) and \(\{\varphi_t\}_{t=1}^T\) is a distribution of words for each topic. Especially interesting is that the model contains a set of discrete intermediate latent variables \(\{z_n\}_{n=1}^N\), which makes it non-differentiable per default. However, NumPyro supports automatic enumeration, which transforms the above generative model to the following equivalent model:

\[
\begin{align*}
\theta & \sim \text{Dir}(\alpha) \\
w_n & \overset{iid}{\sim} \sum_{t=1}^T \text{Cat}(\varphi_t)\text{Cat}(t|\theta) \\
n & \in \{1..N\}
\end{align*}
\]

![Figure 2: 100 particles fitted to Neal’s Funnel using Stein in EinStein VI [left]. EinStein VI with neural transformation applied to the two moons model.](https://github.com/pyro-ppl/numpyro/blob/master/examples/neutra.py)
Under review as a conference paper at ICLR 2020

```python
def flows(n_flows, in_dim, h_dim):
    fs = []
    for i in range(n_flows-1):
        arn = AutoRegNN(in_dim, h_dim,
                        out_fun=Elu)
        fs.append(IAutoRegTrans(arn))
        fs.append(PermTrans(
            arange(in_dim)[::-1]))
    arn = AutoregNN(in_dim, h_dim,
                    out_fun=Elu)
    fs.append(IAutoRegTrans(arn))
    return fs
```

```python
def guide(n_flows=3, h_dim=[2,2]):
    flows = flows(n_flows, 2, h_dim)
    particle = param('p', array([0.,0.]),
                     ComposeTrans(flows))
    numpyro.sample('x', Delta(particle))
```

(a) Inverse AutoRegressive Transformation Flow.  
(b) Guide with neural transformation

Figure 4: Neural transformed guide using Inverse Autoregressive Flows in Numpyro for EinStein VI (function names are shortened for formatting purposes).

Here, all the potential topic choices \( t \in \{1..T\} \) are considered explicitly and marginalized (summed) out in the intermediate step. This way, we no longer have a discrete sampling step and we can differentiate the model again.

EinStein VI integrates with this automatic enumeration support in NumPyro, and thus provides more modelling power than traditional Stein methods that are not integrated into a corresponding powerful framework. In our example, the guide uses amortized inference [Gershman and Goodman, 2014] with a multi-layer perceptron (MLP) with 100 nodes and 20 potential topics, mapping each document to the distribution of topics, thus approximating the desired posterior \( p(\theta | \mathbf{w}) \). For each Stein particle, we get a separate MLP, and thus the resulting distribution would be a Stein mixture of neural networks.

Stein Mixture Deep Markov Model Music generation requires a model to learn complex temporal dependencies to achieve local consistency between notes. The Stein Mixture Deep Markov Model (SM-DMM) is a Deep Markov Model that uses a mixture of Stein particles to estimate distributions over model parameters. We consider a vectorized version of DMM [Jankowiak and Karaletsos, 2019] for the generation of polyphonic music using the JSB chorales dataset.

The SM-DMM model consists of two feed-forward neural networks. The Transition network transitions between latent states in the Markov chain and the Emitter produces an observation at each time step from the current latent state. The variational distribution is of the form \( \prod_{n=1}^{N} q(z_{1:T_n} | f(X_{1:T_n})) \) where the parametrized feature function \( f_{1:T_n} \) is a GRU [Chung et al., 2014], with hidden dimension 400. In the JSB chorales dataset, each time step is an eighth of a note.

We run the SM-DMM using the Adam optimizer [Kingma and Ba, 2014] with a learning rate of \( 10^{-5} \), using an RBF kernel and five Stein particles for one thousand epochs. The trained SM-DMM achieves a negative log likelihood of 6.67 on the test set, comparable to the 6.85 originally reported in Krishnan et al. [2016] and 6.82 reported in the improved version in Jankowiak and Karaletsos [2019].

5 Summary

We presented EinStein VI, a novel library for Stein Variational Inference, built on top of NumPyro. It supports all recent techniques associated with Stein VI, including ELBO-within-Stein optimization, non-linear scaling of repulsive force and matrix-valued kernels, as well as adding novel features like parameter transforms. We have provided several examples that illustrate how EinStein VI has an easy-to-use high-level interface and performs well on pathological examples as well as Stein Mixture versions of realistic models such as Latent Dirichlet Allocation and Deep Markov Model.

In the future, we plan on supporting updates inspired by Stein Points [Chen et al., 2018, 2019], to handle cases where initialization of particles is sub-optimal, e.g. random restart of particles and MCMC updates to get better mode hopping properties.
REFERENCES


ACKNOWLEDGEMENTS

We thank the Pyro development team for answering our questions regarding the Pyro implementation. We especially thank Martin Jankowiak for providing us guidance with and access to the Deep Markov Model example.

A Kernels in EinSTEIN VI

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Definition</th>
<th>Detail</th>
<th>Type</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radial Basis Function (RBF)</td>
<td>(\exp\left(\frac{1}{\ell^2} | x - y |_2^2\right))</td>
<td>scalar</td>
<td>Liu and Wang (2016)</td>
<td></td>
</tr>
<tr>
<td>Inverse Multi-Quadratic (IMQ)</td>
<td>(\exp\left(\frac{1}{\ell^2} (c^2 + | x - y |_2^2)^{\beta}\right))</td>
<td>(\beta \in (-1, 0)) and (c &gt; 0)</td>
<td>scalar</td>
<td>Gorham and Mackey (2017)</td>
</tr>
<tr>
<td>Random Feature Expansion</td>
<td>(E_w[\phi(x, w)\phi(y, w)])</td>
<td>scalar</td>
<td>Liu and Wang (2018)</td>
<td></td>
</tr>
<tr>
<td>Linear</td>
<td>(x^\top y + 1)</td>
<td>scalar</td>
<td>Liu and Wang (2018)</td>
<td></td>
</tr>
<tr>
<td>Mixture</td>
<td>(\sum_i w_i k_i(x, y))</td>
<td>({k_i}) individual kernels and (w_i) are weights</td>
<td>scalar, vector, matrix</td>
<td>Liu and Wang (2018)</td>
</tr>
<tr>
<td>Scalar-based Matrix</td>
<td>(k(x, y)I)</td>
<td>(k) scalar-valued kernel</td>
<td>matrix</td>
<td>Wang et al. (2019)</td>
</tr>
<tr>
<td>Vector-based Matrix</td>
<td>(\text{diag}(k(x, y)))</td>
<td>(k) vector-valued kernel</td>
<td>matrix</td>
<td>Wang et al. (2019)</td>
</tr>
<tr>
<td>Graphical</td>
<td>(\text{diag}{{K^{(\ell)}(x, y)}_{\ell}})</td>
<td>({K^{(\ell)}(x, y)}_{\ell}) matrix-valued kernels each for a unique partition of latent variables</td>
<td>matrix</td>
<td>Wang et al. (2019)</td>
</tr>
<tr>
<td>Constant Preconditioned</td>
<td>(Q^{-\frac{1}{2}} K(Q^{\frac{1}{2}} x, Q^{\frac{1}{2}} y)Q^{-\frac{1}{2}})</td>
<td>(K) is inner matrix-valued kernel and (Q) is a preconditioning matrix like the Hessian (-\nabla^2_{\bar{z}} \log p(\bar{z}</td>
<td>x)) or Fischer information (-E_{\bar{z} \sim q(\bar{z})} [\nabla^2_{\bar{z}} \log p(z</td>
<td>x)]) matrices</td>
</tr>
<tr>
<td>Anchor Point Preconditioned</td>
<td>(\sum_{\ell=1}^{m} K_{Q_{\ell}}(x, y) w_{\ell}(x) w_{\ell}(y))</td>
<td>({a_{\ell}}<em>{\ell=1}^{m}) is a set of anchor points, (Q</em>{\ell} = Q(a_{\ell})) is a preconditioning matrix for each anchor point, (K_{Q_{\ell}}) is an inner kernel conditioned using (Q_{\ell}), and (w_{\ell}(x) = \text{softmax}_{\ell}{N(x</td>
<td>a_{\ell'}, Q_{\ell'}^{-1})}_{\ell'})</td>
<td>matrix</td>
</tr>
</tbody>
</table>