Prediger, Lukas; Loppi, Niki; Kaski, Samuel; Honkela, Antti

d3p - A Python Package for Differentially-Private Probabilistic Programming

Published in:
Proceedings of Privacy Enhancing Technologies

DOI:
10.2478/popets-2022-0052

Published: 01/04/2022

Document Version
Publisher's PDF, also known as Version of record

Please cite the original version:
Prediger, L., Loppi, N., Kaski, S., & Honkela, A. (2022). d3p - A Python Package for Differentially-Private Probabilistic Programming. *Proceedings of Privacy Enhancing Technologies*, *2022*(2), 407–425. https://doi.org/10.2478/popets-2022-0052

Lukas Prediger*, Niki Loppi, Samuel Kaski, and Antti Honkela

d3p - A Python Package for Differentially-Private Probabilistic Programming

Abstract: We present d3p, a software package designed to help fielding runtime efficient widely-applicable Bayesian inference under differential privacy guarantees. d3p achieves general applicability to a wide range of probabilistic modelling problems by implementing the differentially private variational inference algorithm, allowing users to fit any parametric probabilistic model with a differentiable density function. d3p adopts the probabilistic programming paradigm as a powerful way for the user to flexibly define such models. We demonstrate the use of our software on a hierarchical logistic regression example, showing the expressiveness of the modelling approach as well as the ease of running the parameter inference. We also perform an empirical evaluation of the runtime of the private inference on a complex model and find a ∼10 fold speed-up compared to an implementation using TensorFlow Privacy.

Keywords: differential privacy, JAX, NumPyro, probabilistic programming, variational inference

DOI 10.2478/popets-2022-0052
Received 2021-08-31; revised 2021-12-15; accepted 2021-12-16.

1 Introduction

Probabilistic modelling presents a natural way to model data by describing their (assumed) generative process. The model is then fit to observations by probabilistic inference algorithms. Probabilistic programming aims to make the process easy by allowing the user to only specify the model while the system manages the inference process. Probabilistic programming frameworks such as Stan [6], Pyro [3] and PyMC3 [14] have become popular, but they currently offer no support for privacy-preserving algorithms, which are needed for learning from sensitive data.

Differential privacy (DP) [10] provides a rigorous mathematical framework for addressing privacy concerns and has become the de-facto standard notion for privacy in machine learning. It essentially assures that an algorithm's outputs will not differ significantly whether a specific individual's data record is included in the data set or not. Unfortunately, differentially-private algorithms are usually more complex than their non-private counterparts. Software support for easily performing fast differentially-private inference is therefore a crucial tool to achieve privacy-preserving probabilistic programming. This will greatly simplify applications such as differentially-private data anonymisation using a generative probabilistic model to publish a privacy-preserving synthetic twin of a sensitive data set [19].

Using existing probabilistic programming frameworks with privacy-preserving inference is a highly non-trivial task. Practitioners are forced to come up with their own implementation, either from scratch or by adapting existing privacy-enabling libraries, which can be an onerous process and leads to many users having to implement the same (or quite similar) wrapper code. There is therefore a clear need for software solutions that enable privacy-preserving probabilistic programming in a convenient and runtime efficient way to allow for fast prototyping and development involving probabilistic programming under privacy constraints.

We address this gap and extend the tool set for growing adoption of DP by introducing an open-source Python software package called d3p.[1] d3p focuses on providing a reliable high-performance implementation of differentially-private doubly stochastic variational inference (DP-VI) [18] for tabular data, where each record corresponds to a single individual. d3p extends the NumPyro probabilistic programming framework [3, 32], allowing modellers to express and fit a large class of probabilistic models under strict privacy guarantees. Alongside the fundamental DP-VI algorithm, d3p uses

***Corresponding Author: Lukas Prediger:** Aalto University, Finland. E-mail: lukas.m.prediger@aalto.fi
Niki Loppi: NVIDIA AI Technology Center, Finland
Samuel Kaski: Aalto University, Finland & University of Manchester, UK
Antti Honkela: University of Helsinki, Finland

[1] Available at: https://github.com/DPBayes/d3p

a state-of-the-art privacy accounting technique [24] to compute tight bounds on the privacy parameters, allowing it to achieve higher levels of utility than with other commonly employed accountants.

Behind the scenes, d3p relies on the JAX framework [4] to perform computation on GPUs and implements a GPU-optimised minibatch sampling algorithm to further optimise performance. Using d3p we achieve a $\sim$10 fold speedup for fitting a variational auto-encoder model [23] compared to a similar implementation using TensorFlow Privacy [33] on modern GPUs.

d3p addresses a research audience of probabilistic modelling practitioners working with sensitive data. We aim to provide a helpful tool for experimental modelling under privacy constraints. Our main focus in its design therefore is on usability and runtime performance to enable fast modelling iterations. Due to this, d3p currently does not address technical issues arising from implementing idealised differentially private algorithms on machines with imperfect sources of randomness and finite precision, discussed further in Sec. 2.5; these could theoretically be exploited by an adversary if deployed in a production setting.

In summary, we contribute a versatile and performant off-the-shelf implementation of a privacy-preserving probabilistic programming framework as a solid basis for further research. Additionally, we introduce a highly performant subsampling approach based on a slight modification of the CUDA-Shuffle [29], a recently introduced GPU-optimised shuffling approach, and provide a (probabilistic) runtime analysis for it as a minor contribution in methods.

The remainder of the paper is organised as follows: Section 2 reviews probabilistic programming and differentially private variational inference. Based on that discussion we identify software requirements for our software package to clearly outline our design considerations in the same section. We then demonstrate use of our software on a non-trivial hierarchical logistic regression example, illustrating the expressiveness of the probabilistic programming approach and probabilistic modelling (Section 3). Section 4 highlights some implementation details that are orthogonal to the private inference algorithm that forms the core of our framework but that we consider interesting for the user. This includes the discussion of and establishing of (probabilistic) runtime bounds for a special case of the CUDA-Shuffle algorithm which enables GPU-optimised minibatch sampling in our software. Finally, Section 5 presents an evaluation of the d3p framework, including a runtime comparison to a TensorFlow-based implementation, a demonstration of the model introduced in Section 3 and a replication of an experiment for the DP-VI algorithm in [18].

2 Differentially Private Probabilistic Programming

In this section, we review the background and techniques for differentially private probabilistic programming that inform the implementation choices of our framework. We start with a broad general introduction of the probabilistic programming paradigm and variational inference (Sec. 2.1) and the definition of differential privacy as our main privacy formalism (Sec. 2.2). Following that we give an outline of the powerful DP-VI private inference algorithm (Sec. 2.3) and review privacy accounting tools (Sec. 2.4). Finally we briefly point out technical difficulties resulting from implementing idealised differentially private algorithms on machines with imperfect sources of randomness and finite precision (Sec. 2.5). Each subsection provides an overview of the topic and allows us to identify major requirements for a software implementation, which are summarised in Table 1. The requirements we identify correspond directly to our overarching goals of providing software that is convenient to use and highly performant. Accordingly, we categorise requirements in Table 1 with the labels *usability* or *performance*. We hope that explicitly stating our design goals here will allow the reader to evaluate whether our design goals are suitable for their use case and make an informed decision on whether to use d3p. We also do so to emphasise that the implementation of a software package for general use must consider other factors than a (prototypical) implementation of a newly devised method for a research paper.

2.1 Probabilistic Programming

Probabilistic programming is a programming paradigm in which a user programmatically defines a statistical model of data which often depends on a set of parameters θ. In mathematical terms, such a model determines a probabilistic density function $p(\cdot|\theta)$. A probabilistic inference algorithm is then used to determine the posterior distribution $p(\theta|X)$ of the parameter values given a training data set X. The posterior is given by Bayes' formula as $p(\theta|X) \propto p(X|\theta)p(\theta)$, where, $p(X|\theta)$ is the likelihood of the data under the probabilistic model with

No.	Requirement	Category	Section
I	Integrate with an existing popular probabilistic programming framework.	*usability*	2.1
II	Provide assistance for the user in finding adequate privacy bounds	*usability*	2.2
III	Perform efficient per-instance gradient computation and clipping.	*performance*	2.3
IV	Determine DP inference algorithm parameters (C and σ) automatically.	*usability*	2.3
V	Perform efficient independent minibatch sampling.	*performance*	2.3
VI	Provide state-of-the-art privacy accounting.	*usability*	2.4

Table 1. Requirements for the differentially private probabilistic programming framework with their corresponding category and the subsection they were discussed in (in order of appearance in the text).

given parameter values. $p(\theta)$ is a prior distribution encapsulating existing knowledge about plausible parameter values.

There exist a number of different probabilistic programming languages that use different ways of specifying the model. To enable easy user adoption, a software package providing differential privacy for probabilistic programming should not aim to re-define and re-implement yet another solution but rather integrate with existing solutions by extending them with support for differentially private inference (Table 1, Requirement I).

Defining an Example Model

To illustrate the concept of probabilistic programming, we present as an example the implementation of logistic regression for binary classification in NumPyro.

The simple logistic regression model we consider first is for a data set X of records $\boldsymbol{x}_i \in \mathbb{R}^D$ with corresponding labels $y_i \in \{0, 1\}$. We assume that each such record and label corresponds to a single individual.

Mathematically the logistic regression model is formulated as

$$\begin{aligned} p(y_i|\boldsymbol{x}_i, \boldsymbol{w}) &= \text{Bernoulli}\left(y_i; \theta_i\right), \\ \theta_i &= \sigma(\boldsymbol{w}^T \boldsymbol{x}_i), \\ p(\boldsymbol{w}) &= \mathcal{N}(\boldsymbol{w}; \boldsymbol{w}_0, \boldsymbol{S}_0), \end{aligned}$$

where $\sigma(x) = 1/(1+\exp(-x))$ denotes the sigmoid function and $\text{Bernoulli}(\cdot;\theta)$ denotes the Bernoulli distribution with success probability θ. The model uses a weight vector $\boldsymbol{w} \in \mathbb{R}^D$ to express the relationship between the records and labels. Bayesian treatment allows us to formulate a prior on $\boldsymbol{w}$ to express any prior knowledge about plausible parameter values. Here we use a weakly informative, zero-centered Gaussian prior with $\boldsymbol{w}_0 = 0$ and $\boldsymbol{S}_0 = 4\boldsymbol{I}$. The mathematical description of the model equations translates naturally into Python code for NumPyro model definition in Listing 1.

```python
# specifies the model p(ys, w | xs)
def model(xs, ys, N):
    # obtain data dimensions
    batch_size, d = xs.shape

    # the prior for w
    w = sample('w', Normal(0, 4),
               sample_shape=(d,))

    # distribution of label y for each record x
    with plate('batch', N, batch_size):
        theta = sigmoid(xs.dot(w))
        sample('ys', Bernoulli(theta), obs=ys)
```

Listing 1. Definition of a simple logistic regression model in NumPyro for d3p.

Doubly Stochastic Variational Inference

At the heart of probabilistic programming lies the inference algorithm that is used to determine the posterior distribution of parameters $p(\theta|X)$. For complex models computing this posterior exactly is typically intractable. Variational inference [20, 40], a class of approximate inference algorithms, therefore approximates it with a simpler, tractable distribution $q(\theta|\psi)$. The parameters ψ of q are found by solving the optimisation problem $\min_\psi D(q(\theta|\psi)||p(\theta|X))$ where $D(\cdot||\cdot)$ is a divergence measure for probability distributions.

Doubly stochastic variational inference (DSVI) [38] is a gradient ascent algorithm for non-conjugate models with differentiable (joint) probability densities $p(X, \theta)$ and a $q(\theta|\psi)$ from which values can be easily sampled algorithmically. While these conditions limit applicability somewhat, they still allow for a large class of models to be fitted. DSVI minimises the KL-divergence by maximising the so-called *evidence lower bound (ELBO)*, defined as

$$\mathcal{L}(\psi|X) = \mathbb{E}_{\theta\sim q(\theta|\psi)}\left[\log p(X|\theta) + \log p(\theta) - \log q(\theta|\psi)\right]. \tag{1}$$

The expectation is approximated stochastically by sampling θ from $q(\theta|\psi)$ and using a minibatch of the training data for each gradient step. For details we refer to [38].

2.2 Differential Privacy

We rely on (approximate) differential privacy [9, 10] as the primary privacy notion for a privacy-preserving variant of the DSVI algorithm. Following [8, Def. 2.4], it is defined as:

Definition 1 (Approximate Differential Privacy)**.** A randomised algorithm $\mathcal{M}$ satisfies (ε, δ)-differential privacy with $\varepsilon > 0$ and $0 \leq \delta \leq 1$ if, for all neighbouring data sets $X \sim X'$, and for all $S \subset \operatorname{im}(\mathcal{M})$, we have

$$\Pr(\mathcal{M}(X) \in S) \leq e^{\varepsilon} \Pr(\mathcal{M}(X') \in S) + \delta. \tag{2}$$

The two data sets X and X' are considered to be neighbours, denoted $X \sim X'$, when we can obtain one from the other by adding (resp. removing) a single element. ε and δ are *privacy bounds* (or *privacy parameters*) restricting the effect that the presence of any particular record in the input data set has on the output of the algorithm $\mathcal{M}$.

Smaller values for these privacy bounds correspond to stricter privacy, however there is a trade-off between privacy and utility of the algorithm's outputs. Larger values for privacy bounds typically result in higher utility of the outputs, as they allow more information to pass through the algorithm. Choosing the privacy bounds therefore requires careful consideration of this trade-off. This is difficult because for many users, especially those inexperienced with DP, it is not clear how to interpret the privacy bounds in a concrete setting. The software should therefore assist the user in choosing appropriate privacy bounds which we reflect in Requirement II.

2.3 Differentially Private Doubly Stochastic Variational Inference

Jälkö et al. introduced a (ε, δ)-DP version of the doubly stochastic variational inference algorithm in [18]. This DP-VI algorithm is derived from the influential DP-SGD [2, 34] and the relevant steps of a single iteration (out of T many) can be summarised as

1. Sample a random minibatch of size B from the training data set.
2. Sample a set of parameters θ from $q(\cdot|\psi)$.
3. For each instance in the minibatch:
    1. Compute the gradient of the ELBO.
    2. Clip the norm of the gradient to a bound C.
4. Aggregate per-instance gradients.
5. Perturb by adding zero-mean Gaussian noise with variance $C^2\sigma^2$.
6. Update the model parameters ψ with the perturbed gradient.

The main mechanism by which differential privacy is achieved is the perturbation of the minibatch gradient in step 5 via the Gaussian mechanism [8, Thm. 3.22]. The level of noise, characterised by its variance σ^2, must be carefully calibrated to provide the desired level of privacy. However, the gradient of any data instance could in theory be arbitrarily large, rendering any fixed noise level ineffective. To remedy this, the DP-VI algorithm enforces an upper bound C on the gradient of each data instance (in step 3.2). The important implication of this is that an implementation of DP-VI needs an efficient way of computing and manipulating the per-instance (often also known as per-example) gradients in a minibatch instead of a single gradient over the entire minibatch (Requirement III).

Another important observation is that the DP-VI algorithm has additional hyperparameters C and σ which govern the privacy vs. accuracy trade-off. Especially σ depends non-trivially on the clipping bound C, desired privacy bounds ε and δ, batch size B and the number of iterations T. The next requirement for the software package is therefore the ability to (automatically) derive appropriate values for the DP-VI hyperparameters from these other hyperparameters (Requirement IV).

A final crucial point is that the algorithm is shown to provide differential privacy only under the assumption that minibatches are independently sampled from the training set. As this needs to occur in every iteration of the algorithm, this routine must be especially fast to not slow down the inference as a whole, making another performance requirement for our software (Requirement V).

2.4 Privacy Accounting

As we have seen, the DP-VI algorithm consists of an iterative application of the (subsampled) Gaussian mechanism on gradients wrt. random minibatches of the training data. The overall privacy bounds ε and δ of the DP-VI algorithm then result from the DP composition. In order to achieve good utility it is crucial to compute these overall privacy bounds to be as tight as possible: Looser bounds mean that larger perturbations are required for a desired level of privacy, reducing the information extracted from the data and decreasing utility of the inferred model (cf. [11]).

While loose bounds can be computed using general DP composition theorems in a simple way (cf. [8]), obtaining tight bounds typically requires more complex computation using methods called *privacy accountants*. Abadi et al.'s Moments accountant [2] was the first of these and significantly improved over traditional DP composition theorems. The tightest privacy bounds are currently achieved by the Fourier accountant [24].

Privacy accountants are typically of the form $f_{PA}(C, \sigma, B, T, \delta) = \varepsilon$, i.e., they take in the algorithm's parameters as well as a target value for δ and compute the corresponding upper bound for ε. They are therefore the primary tool to translate between privacy bounds and inference hyperparameters and instrumental for addressing Requirements II and IV.

With these considerations, providing an implementation of a state-of-the-art privacy accountant is an important aspect for a differentially private probabilistic programming framework and becomes Requirement VI.

2.5 Remaining Technical Concerns for the Practical Implementation

The definition of approximate differential privacy given in Section 2.2 provides information-theoretic guarantees: It is impossible for the output probabilities of an algorithm to vary too much no matter what the input is. These can typically not be achieved by a computer system which does not have access to perfect sources of randomness for sampling noise and relys on finite-precision approximation of real numbers. Both of these issues have the potential to completely void the privacy guarantees of DP algorithms in practical implementations: Predictable randomness can allow an attacker to remove the perturbations [16] and finite-precision floating point numbers can leak information due to approximation errors [28].

We recognise these issues as generally important for production systems. However, as we have already pointed out, we undergo no effort to address these for the current version of d3p, which is primarily intended as a research tool. We consider solving these issues for the DP-VI algorithm as important future work.

2.6 Summary

We have seen in this section that there is a large number of desiderata for an implementation of a differentially private probabilistic programming framework. These are summarised in Table 1 and fall into different categories that make some mostly technical considerations for the implementation (e.g., Requirements III, V), while others are important aspects in the design of the user interface (e.g., Requirements II, IV).

In the following sections we first explore how our software package addresses these requirements from a user perspective by implementing an example model. However, some of the technical requirements are not experienced by the user in the programming interface directly and cannot be demonstrated in the example: Whether the implementation is performant (Requirements III, V) has almost no effect on the interface seen by the user but is nevertheless a crucial part of their experience. We therefore briefly discuss some implementation details following the examples to illustrate how the identified requirements were addressed and, following that, provide an empirical evaluation of the runtime performance.

3 d3p Usage Example

We will now demonstrate our d3p software package on a practical example to show how the previously identified requirements are addressed from a user's perspective. d3p admits non-conjugate models with differentiable probability densities and is designed to provide differential privacy guarantees for tabular data, where each individual contributed a single sensitive record and records are assumed conditionally independent. d3p uses NumPyro [3, 32] as a modelling language and JAX [4, 15] as the underlying computation framework, which offers an API similar to NumPy [17]. We start by fully implementing the simple logistic regression model shown in Section 2.1 to demonstrate the basics of probabilistic programming and show how d3p's DP-VI algorithm is invoked to infer the model's parameters. We then highlight the expressiveness of the probabilistic programming approach by adapting the code to a more complex model that achieves a better fit to the data.

3.1 Defining a Model

We recall the simple logistic regression model introduced in Sec. 2.1. The model is for a data set X of records $\boldsymbol{x}_i \in \mathbb{R}^D$ with corresponding binary labels $y_i \in \{0, 1\}$. We assume that each such record and label corresponds to a single individual.

As before, the model can be formulated mathematically as

$$\begin{aligned} p(y_i|\boldsymbol{x}_i, \boldsymbol{w}) &= \text{Bernoulli}\,(y_i; \theta_i)\,, \qquad (3) \\ \theta_i &= \sigma(\boldsymbol{w}^T \boldsymbol{x}_i), \\ p(\boldsymbol{w}) &= \mathcal{N}(\boldsymbol{w}; \boldsymbol{w}_0, \boldsymbol{S}_0), \end{aligned}$$

where $\sigma(x) = 1/(1 + \exp(-x))$ denotes the sigmoid function and $\text{Bernoulli}(\cdot; \theta)$ denotes the Bernoulli distribution with success probability θ. The weight vector $\boldsymbol{w} \in \mathbb{R}^D$ links the records and labels. For a Bayesian treatment we place a prior on $\boldsymbol{w}$ to encode existing knowledge. In this first example we assume that we do not have strong prior knowledge but want to enforce some regularisation, and therefore use a weakly informative, zero-centered Gaussian prior with $\boldsymbol{w}_0 = 0$ and $\boldsymbol{S}_0 = 4\boldsymbol{I}$. The model is visualised using plate diagram notation in Figure 1.

Implementation of the Model

The implementation of this model using the NumPyro probabilistic programming framework is reproduced in the `model` function in the top part of Listing 2. The model is defined as a function taking (a minibatch of) the data as an input and specifies a sampling process in an imperative programming style: Values for $\boldsymbol{w}$ are sampled from the specified Gaussian prior with zero mean and a standard deviation of 4. The values for labels are sampled from the Bernoulli distribution (cf. Eq. 3). However, the `sample` call for y is conditioned to return the values `ys` passed into the `model` function using the `obs` keyword. This is the mechanism by which the labels are passed into the inference algorithm despite the model being specified from a generative perspective.

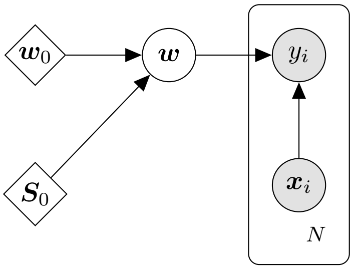

Fig. 1. Plate diagram of the simple logistic regression model $p(y, \boldsymbol{w}|\boldsymbol{x})$.

```
import jax.numpy as jnp
# specifies the model p(ys, w | xs)
def model(xs, ys, N):
    # obtain data dimensions
    batch_size, d = xs.shape

    # the prior for w
    w = sample(’w’, Normal(0, 4),
               sample_shape=(d,))

    # distribution of label y for each record x
    with plate(’batch’, N, batch_size):
        theta = sigmoid(xs.dot(w))
        sample(’ys’, Bernoulli(theta), obs=ys)

# specifies the variational posterior q(w)
def guide(xs, ys, N):
    d = jnp.shape(xs)[1]

    # variational parameters
    w_loc = param(’w_loc’, jnp.zeros((d,)))
    w_scale = jnp.exp(param(’w_scale_log’,
                            jnp.zeros((d,))))

    # variational distribution for w
    sample(’w’, Normal(w_loc, w_scale))
```

Listing 2. Implementation of a simple logistic regression model in NumPyro for `d3p`.

NumPyro’s `plate` context manager is used to express the independence assumption for the individual data records. Note that this is an important assumption in the `d3p` package and must be reflected in the model in this way.[2] This requires the additional argument `N` to the model, which specifies the total amount of data records in the training data.

Implementation of the Variational Posterior

For inference of the model’s parameters in our example, we use independent Gaussian distributions for every data dimension j with parameters $\mu_{\boldsymbol{w},j}$ and $\sigma_{\boldsymbol{w},j}$ as the

[2] Apart from clearly stating the assumptions made for the data, this ensures that using minibatches instead of the whole data set does not affect the amount by which an individual sample contributes to the ELBO.

variational approximation to the posterior distribution, i.e.,

$$q(w_j|\mu_{\boldsymbol{w},j}, \sigma_{\boldsymbol{w},j}) = \mathcal{N}(\mu_{\boldsymbol{w},j}, \sigma^2_{\boldsymbol{w},j}). \qquad (4)$$

The variational parameters $\boldsymbol{\mu_w}$ and $\boldsymbol{\sigma_w}$ will be optimised according to the discussion in Section 2.1. The corresponding NumPyro implementation is shown in the `guide` function, following naming conventions of NumPyro, in the lower portion of Listing 1. We register $\boldsymbol{\mu_w}$ and $\boldsymbol{\sigma_w}$ as parameters `w_loc` and `w_scale` for the inference algorithm and sample w according to Equation 4 in a vectorised fashion. Note that each $\sigma_{\boldsymbol{w},j}$, the standard deviation of the variational Gaussian, must be a positive number, so we actually register a parameter site named `w_scale_log` that we pass through the exponential function to obtain `w_scale`. This allows us to perform the optimisation in an unconstrained space but enforces the positivity constraint for `w_scale`.

3.2 Running the Inference

In the previous section we programmatically specified the simple logistic regression model using NumPyro. We now turn to the actual private inference of parameter values using d3p. d3p provides an implementation of the DP-VI algorithm via the `DPSVI` class, which offers the same interface as NumPyro's implementation of the DSVI algorithm in the `SVI` class.[3] As discussed in Section 2.3, the DP-VI algorithm must be configured for the desired privacy bounds given the batch size and number of training iterations for the inference (Req. IV) and use independent random minibatches (Req. V). The entire code for running the inference is shown in Listing 3.

We first use d3p's `subsample_batchify_data` on the data set which returns a function that efficiently samples and returns independent random minibatches. This function is assigned to `get_batch` in our example code. The additional call to `batchify_init` initialises the internal state of the minibatch sampler.

To instantiate a `DPSVI` object, the main driver of the inference, we need to supply a value for the privacy noise scale σ. We can obtain a σ appropriate for our desired privacy bounds and training hyperparameters using the `approximate_sigma` function. This function returns an approximate value for σ that is guaranteed to achieve the privacy bounds as measured by the Fourier accountant [24], a state-of-the-art privacy accountant method. We store the result in `dp_scale`. Note that the separate computation of σ is a deliberate choice in d3p. While it would be possible to let the instantiation code of the `DPSVI` class handle this internally, the current approach allows the user easily provide values for σ different from the ones computed by the `approximate_sigma` function, e.g., for research purposes.

```python
def infer(data, labels, batch_size, num_iter,
    epsilon, delta, rng_key):
    # set up minibatch sampling
    batchifier_init, get_batch = \
        subsample_batchify_data((data, labels),
                                batch_size)
    _, batchifier_state = \
        batchifier_init(rng_key)

    # set up DP-VI algorithm
    q = batch_size / len(data)
    dp_scale, _, _ = approximate_sigma(
        epsilon, delta, q, num_iter)
    loss = Trace_ELBO()
    optimiser = Adam(1e-3)
    clipping_threshold = 1.
    dpsvi = DPSVI(model, guide, optimiser,
                  loss, clipping_threshold,
                  dp_scale, N=len(data))
    svi_state = dpsvi.init(
        rng_key,
        *get_batch(0, batchifier_state))

    # run inference
    for i in range(num_iter):
        data_batch, label_batch = \
            get_batch(i, batchifier_state)
        svi_state, loss = dpsvi.update(
            svi_state, data_batch, label_batch)
    return dpsvi.get_params(svi_state)
```

Listing 3. Running the inference for a NumPyro model using d3p's `DPSVI` class.

After also instantiating implementations of the ELBO (`loss`) and an optimiser of our choice (`optimiser`) using classes provided by NumPyro, we are ready to create the `DPSVI` object (`dpsvi`). Similar to the minibatch sampler, the DP-VI algorithm provides an initialisation function that is called to produce a state object (`svi_state`). The state object contains randomness state as well as the current values of the parameters and the state of the optimiser. We can now finally run the inference by repeatedly sampling a batch using

[3] We chose `DPSVI` as the name for implementation of the DP-VI algorithm in d3p to stay close to NumPyro's naming convention.

the `get_batch` function we obtained before and then calling `dpsvi.update`. The `update` method completely encapsulates a single iteration of the DP-VI algorithm, including the performant per-instance gradient computation (Req. III), clipping, perturbation and the update of current parameter estimates by the optimiser.

Obtaining parameter estimates from the inference algorithm (and therefore the approximate posterior distribution $q(w)$ in our example model) completes our example at this point. The user can now use standard NumPyro code to interact with the model and the inferred parameters without additional privacy leakage due to DP's invariance to post-processing.

3.3 Switching to a More Complex Model

One of the main benefits of the probabilistic programming approach is the ability to easily tailor the model complexity to the information needs and the available prior knowledge and clearly specifying how the respective components of the model relate to each other. So far we have looked at a very simple example where our model makes the implicit assumption that the data are homogeneous and a single parameter vector $\boldsymbol{w}$ describes their relation to the labels equally well for all records. In reality, however, we often face tasks where data comes from different sources that have different local distributions for records. For example, data records containing information about wealth and income of individual persons from different countries are likely to be heavily influenced by the average level of wealth in the respective country. To address this case, we can use a hierarchical logistic regression model like the one considered in [42].

Extending the Model Specification

We now extend the previous model notation by vectors $\boldsymbol{g}_l \in \mathbb{R}^K$ of group characteristics for L groups and, for each record $\boldsymbol{x}_i$, an indicator l_i assigning it to one of the groups. We assume that the records $\boldsymbol{x}_i$ and labels y_i are sensitive but the group vectors $\boldsymbol{g}_l$ are not. Following [42] we use a separate weight vector $\boldsymbol{w}_l$ to model the relationship between data record $\boldsymbol{x}_i$ and label y_i within each group in the same way we did in the simple logistic regression model. However, we now use a hierarchical Gaussian prior centered at $\boldsymbol{M}\boldsymbol{g}_l$ for each weight vector $\boldsymbol{w}_l$. The matrix $\boldsymbol{M} \in \mathbb{R}^{D\times K}$ is a new parameter capturing the relation between group characteristics and their corresponding weight vector $\boldsymbol{w}_l$. The full model is visualised in Figure 2 and defined by

$$\begin{aligned} p(y_i^{(l)}|\boldsymbol{x}_i^{(l)}, \boldsymbol{w}_l) &= \mathrm{Bernoulli}(y_i^{(l)}; \theta_i^{(l)}), \\ \theta_i^{(l)} &= \sigma(\boldsymbol{w}_l^T \boldsymbol{x}_i^{(l)}), \\ p(\boldsymbol{w}_l|\boldsymbol{g}_l, \boldsymbol{M}, \boldsymbol{\Sigma}_l) &= \mathcal{N}(\boldsymbol{w}_l; \boldsymbol{\eta}_l, \boldsymbol{\Sigma}_l), \\ \boldsymbol{\eta}_l &= \boldsymbol{M}\boldsymbol{g}_l, \\ p(M_{k,d}) &= \mathcal{N}(M_{k,d}; \mu_0, \sigma_0^2). \end{aligned} \tag{5}$$

For simplicity, we consider the covariance matrix $\boldsymbol{\Sigma}_l$ for the distribution of the $\boldsymbol{w}_l$ a fixed model parameter of value $\boldsymbol{\Sigma}_l = \boldsymbol{I}$. We assume an independent weakly informative Gaussian prior for each element of matrix $\boldsymbol{M}$ with $\mu_0 = 0$ and $\sigma_0 = 4$ as before.

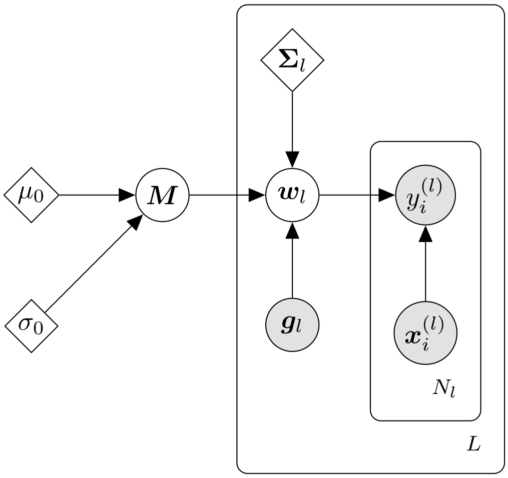

Fig. 2. Plate diagram of the hierarchical logistic regression model $p(y, \boldsymbol{M}|\boldsymbol{x}, \boldsymbol{g}_l)$.

Model Implementation

The required changes to adapt our existing model implementation in function `guide` are straightforward and essentially follow one to one from the textual description above, as shown in Listing 4. First we sample a value for $\boldsymbol{M}$ according to our prior (`M`), compute the values for $\boldsymbol{\eta}_l$ (`etas`) and sample the group-specific weight vectors $\boldsymbol{w}_l$ (`ws`) in a vectorised manner. The remaining code implementing the logistic regression for individual records is almost identical to the implementation of the simple model, except that we use the group indicators l_i provided in `ls` to select the entry in `ws` that corresponds to the $\boldsymbol{w}_{l_i}$ of the group each record belongs to.

```
def model(xs, ys, ls, gs, N):
    batch_size, D = xs.shape
    L, K = gs.shape

    M = sample('M', Normal(0, 4),
               sample_shape=(D, K))

    with plate('group', L, L):
        etas = gs @ M.T
        ws = sample(
            'ws', Normal(etas, 1).to_event(1))

    with plate('batch', N, batch_size):
        thetas = sigmoid(jnp.einsum(
            "nd,nd->n", xs, ws[ls]))
        sample('ys', Bernoulli(thetas), obs=ys)

def guide(xs, ys, ls, gs, N):
    _, D = xs.shape
    _, K = gs.shape

    M_loc = param('M_loc', jnp.zeros((D, K)))
    M_scale = jnp.exp(param('M_scale_log',
                            jnp.zeros((D, K))))
    sample('M', Normal(M_loc, M_scale))
```

Listing 4. Implementation of a hierarchical logistic regression model in NumPyro for d3p.

Variational Posterior

For the hierarchical logistic regression model, we are interested in a variational approximation to the posterior of the matrix $\boldsymbol{M}$. Again we can use independent Gaussian distributions for each dimension of $\boldsymbol{M}$:

$$q(M_{k,d}|\mu_{kd}, \sigma_{kd}) = \mathcal{N}(M_{k,d}; \mu_{kd}, \sigma^2_{kd}), \tag{6}$$

where μ_{kd} and σ_{kd} for all $1 \leq k \leq K$ and $1 \leq d \leq D$ are the variational parameters.

The implementation of this is shown in the `guide` function in Listing 4 and is almost identical to the one for the simple model. Invocation of the inference algorithm for the new model also does not change compared to the simple logistic regression model we have looked at before, except for passing the additional data to the `model` and `guide` functions, making it especially easy and fast to refine and tweak models.

Note that we do not specify posterior parameters for the group weight vectors $\boldsymbol{w}_l$. This is because in our model these are determined by $\boldsymbol{M}$, the group characteristics $\boldsymbol{g}_l$ and the known covariance matrix $\boldsymbol{\Sigma}_l$. If we are interested in obtaining values for the $\boldsymbol{w}_l$, we can use NumPyro routines to sample them from the posterior predictive distribution

$$q(\boldsymbol{w}_l|\boldsymbol{g}_l) = \int p(\boldsymbol{w}_l|\boldsymbol{M}\boldsymbol{g}_l, \boldsymbol{\Sigma}_l) q(\boldsymbol{M}|\boldsymbol{\mu}, \boldsymbol{\sigma}) d\boldsymbol{M} \tag{7}$$

after inference of the variational parameters for $\boldsymbol{M}$. In the above we slightly abuse notation to denote by $q(\boldsymbol{M}|\boldsymbol{\mu}, \boldsymbol{\sigma}) = \prod_{k=1}^{K} \prod_{d=1}^{D} q(M_{k,d}|\mu_{kd}, \sigma_{kd})$ the variational posterior of the full matrix $\boldsymbol{M}$ for all variational parameters.

Discussion

As we have seen we were able to expand our model to the additional structure within our data with a few simple changes in the implementation. The changes correspond directly to the textual description of the new hierarchical model, making it easy for the user to translate theory into implementation. While the initial simple logistic regression model is quite a common choice and specialised implementations for this exist in many software packages, extending the model in a similar fashion to what we did in this section is often more complicated, if at all possible, in those.

Note that we still have made a couple of simplifying assumptions here, one of which is the assumption that the covariance $\boldsymbol{\Sigma}_l$ of the distribution of group weight vectors is a known constant. This was merely for convenience and not because of restrictions of model expressiveness. We can easily formulate a prior and a variational posterior for $\boldsymbol{\Sigma}_l$ to learn it from the data, if this assumption does not hold. Another simplifying modelling assumption is the choice of the prior for $\boldsymbol{M}$, however, we consider selecting more informative priors outside the scope of this example. NumPyro offers a wide range of distributions from which the user can choose adequate priors according to their modelling needs. These two brief notes serve to show that the probabilistic programming paradigm allows the user to make fine-grained decisions on the model expressiveness they need by flexibly either excluding details they are not interested in or incorporating detailed prior knowledge in the model.

4 d3p Implementation Outline

We now discuss some technical details of our d3p package and how it meets the requirements identified in Section 2. In line with our discussion of these requirements, we hope that this helps the reader decide whether the choices made in implementing d3p make it suitable for their use. We also focus on the GPU-optimised minibatch sampling algorithm in particular, which we base on a recent GPU-optimised shuffling method but slightly modify to better fit our use.

We organise this section by first briefly motivating the use of NumPyro as the basis for d3p (Section 4.1). We then turn to the implementation of the DP-VI algorithm for NumPyro (Section 4.2) with a focus on how it realises its high performance for per-instance gradient computation (Req. III). As the second major implementation detail we then discuss our GPU-optimised i.i.d. minibatch sampling routine (Req. V) which has a major impact on the overall performance of the inference and for which we also contribute a runtime analysis (Section 4.3). Finally we finish the discussion on implementation by a brief overview of the remaining usability goals in the implementation (Section 4.4).

4.1 Underlying Framework

To meet Requirement I (*Integrate with an existing popular probabilistic programming framework*), we chose NumPyro as the underlying probabilistic programming framework for d3p. It is a spin-off of the popular Pyro [3] framework, providing a very similar API, but relies on Google's JAX [4, 15] for the underlying computational optimisation functionality. The JAX framework uses tracing mechanisms to compile pure (side-effect free) functions directly from Python code to efficient XLA kernels that can be run on either CPU or GPU. Implementations of the DP-SGD algorithm in JAX where found to be consistently faster than competing implementations [36], making it a promising backend for for d3p.

The choice of NumPyro as basis for d3p is therefore a compromise between fulfilling Requirement I and being able to provide a high-performance implementation (Req. III, V). While targeting Pyro instead of NumPyro would arguably have enabled access to a larger established user base, implementing per-instance gradients in the underlying PyTorch would have decreased performance of d3p.

4.2 Implementation of DP-VI

As we have seen in the examples, d3p centers around the DPSVI class implementing the DP-VI algorithm. DPSVI offers the same interface as NumPyro's non-private implementation in the SVI class and therefore works as a drop-in replacement for it. This allows for especially easy adoption of privacy-preserving methods with minimal required changes to the code base.

We have identified in Section 2.3 that the DP-VI algorithm requires computation of per-instance gradients. Unfortunately, these are typically not readily available in established machine learning frameworks. A naive but inefficient solution is to set the batch size B to one, which gives per-instance gradients at the cost of losing the performance gains resulting from parallel computation on minibatches that are crucial for efficient machine learning applications.

As the JAX framework is based on the manipulation of side-effect free functions, it also provides a range of composable higher order transformations for these. Crucially, the computation of gradients as well as vectorising a function for parallel execution are examples of these higher order transformations. This enables d3p's implementation in the DPSVI's class to efficiently parallelise computation of gradients and the subsequent clipping over a minibatch using vectorisation, i.e., single-instruction-multiple-data (SIMD) style computing. Leveraging the massively parallel processing capabilities of modern GPUs in this way allows us to negate most of the additional overhead introduced by the per-instance gradient computation.

This vectorisation approach effectively turns computation on a batch of size B into B parallel computations on batches of size 1. In order to prevent this from affecting the relative contributions of prior and variational posterior of global parameters, models must use NumPyro's plate environment to scale likelihood contributions from the batch appropriately to the perceived batch size. This typically means that each data record's contribution to the loss is scaled up. The DPSVI class therefore performs some crucial bookkeeping to ensure that the privacy perturbation in each iteration is also scaled appropriately.

As a convenience feature, the DPSVI class offers methods to obtain tight privacy bounds for its current hyperparameter values which are computed using the Fourier accountant [24].[4]

[4] We rely on the fourier-accountant package, available at https://pypi.org/project/fourier-accountant/, for the implementation.

4.3 Performant GPU Batch Subsampling

We have seen in Section 2.3 that the DP-VI algorithm's privacy guarantees rely crucially on minibatches sampled from the data set in a truly i.i.d. fashion (Requirement V). This is an important difference to non-private algorithms that can usually get away with a permute-and-iterate approach to sample minibatches: Often, the data set is permuted once in its entirety and minibatches are then consumed by iterating over the permuted set. Due to being invoked comparatively rarely, the performance of the permutation algorithm does not make a noticeable difference on the overall runtime. For the same reason, it can also run on a different device than the learning algorithm as slow bus transfers are infrequent.

The i.i.d. requirement for DP-VI demands that the minibatch sampling routine is invoked once for each iteration, making the cost of a slow sampling routine prohibitive (or at least, much more noticeable). d3p therefore ships with a parallel and GPU-optimised minibatch sampling routine based on a novel shuffling methodology (CUDA-Shuffle) proposed in [29, 35].

Conventional shuffling algorithms, such as the Fisher-Yates shuffle, are ill-suited for GPU-acceleration as they are sequential. In contrast, the main idea in the parallel shuffle algorithm is to use a bijective function f_k that for a given key k defines a unique pseudo-random mapping on sets of indices $I_B \to I_X$. Assuming $I_B = \{0, \dots B-1\}$ to describe indices in a minibatch and $I_X = \{0, \dots, n-1\}$ indices in the data set, f_k allows sampling a minibatch of elements from the input set in parallel without collisions. Previously, it has been proven that a Feistel network [13] with more than two rounds is a pseudo-random bijective function, provided that it uses a round function that is pseudo-random and the set size is a power of two, i.e., $n = 2^b$ [27].

In [29] this is generalised for arbitrary n by taking the smallest bit-length b such that $2^b > n$, applying the bijection on the index set of length 2^b and then removing all values larger than n by an efficient GPU compaction algorithm. The overall runtime of this is in $\Theta(n)$.

However, for our application of sampling a minibatch, shuffling the entire index set is inefficient. Instead, we apply the Feistel network repeatedly on values from I_B until all outputs are a value in I_X. Our generalised Feistel permutation generator can be given as

$$x_i^{(l)} = \begin{cases} i & \text{, if } l = 0 \\ f_k(x_i^{(l-1)}) & \text{, if } x_i^{(l-1)} \geq n \text{ and } l > 0 \\ x_i^{(l-1)} & \text{, otherwise,} \end{cases} \tag{8}$$

where l is the iteration count and $i \in I_B$. Sampling a minibatch then requires lB evaluations of the Feistel network for some factor l. These evaluations consist only of independent, parallel bitwise operations, making this approach very well-suited for GPU acceleration.

The factor l is the number of iterations that are required to converge Equation 8, for which we will now establish a probabilistic upper bound. The probability with which the Feistel network f_k returns an output $f_k(x) < n$, given a bit-length b such that $2^{b-1} < n < 2^b$ is

$$p = \Pr[f_k(x) < n] = \frac{2^{b-1} + r}{2^b}, \tag{9}$$

where we let $r = n - 2^{b-1}$, i.e., the non-power-of-two residue of n.

Now, let L_i be the random variable of the iteration count required to converge $x_i < n, \forall i \in I_B$ and $F_i = L_i - 1$ be the penultimate iteration count where some $x_i \geq n$. F_i follows a negative binomial distribution, $Pr[F_i = f] = (1-p)^f p$. Its expected value is

$$\mathbb{E}\left[F_i\right] = \frac{1-p}{p} = \frac{2^{b-1} - r}{2^{b-1} + r}, \tag{10}$$

and the cumulative distribution is

$$\Pr[F_i \leq f] = 1 - (1-p)^{f+1} = 1 - \left(\frac{2^{b-1} - r}{2^b}\right)^{f+1}. \tag{11}$$

In the worst case with $r = 1$, resulting in the largest gap to the next power of two, the expected values are

$$\mathbb{E}[F_i] = \frac{2^{b-1} - 1}{2^{b-1} + 1}, \ \mathbb{E}[L_i] = \frac{2^b}{2^{b-1} + 1}. \tag{12}$$

Hence with non-trivial data set sizes with $b \gg 1$, the expected number of iterations required to converge the permutation is $\mathbb{E}[L_i] \approx 2$, in the worst case. To estimate the maximum number of iterations for a given percentage θ of cases, we can calculate

$$\Pr[F_i \leq f_\theta] \leq \theta \Leftrightarrow f_\theta \leq \frac{\log(1-\theta)}{\log(2^{b-1} - r) - \log(2^b)} - 1. \tag{13}$$

Setting $\theta = 0.99$, $b \gg 1$ and the worst case $r = 1$, we obtain

$$f_\theta \geq \frac{\log(1-\theta)}{\log(1/2))} - 1 \approx 5.65 \tag{14}$$

Therefore, in 99% of cases we will see no more than six failures, i.e., seven total iterations (and no more than six in 95% of cases). In practice, as long as the batch size is sufficiently small, precisely $q = B/n < 1/7$, our approach will require less evaluations of the bijection than the one of [29].

4.4 Privacy Bound and Hyperparameter Selection

d3p offers an API to compute the perturbation hyperparameter σ for the DP-VI algorithm via the approximate_sigma function to satisfy Requirement IV (*Determine DP inference algorithm parameters automatically.*) This is realised by employing standard blackbox optimisation techniques to find a suitable input such that the Fourier accountant arrives at the desired value for ε, given all other hyperparameters.

Finally, Requirement II (*Provide assistance for the user in finding adequate privacy bounds*) is an important piece of guidance for users inexperienced with differential privacy. Optimal choice of ε and δ depends on a delicate balance between the desired utility and the level of risk of privacy violation that is considered adequate by the user — a choice, therefore, that can only be made by the user but requires knowledge of how ε, δ relate to concrete privacy risks. Unfortunately, this relationship is still an open research question, which makes it difficult to give detailed guidance to the user. Common practice is to require that $\delta < \frac{1}{N}$, where N is the number of individual records in the data, and $\varepsilon \leq 1$, however d3p does not enforce this currently to allow for free experimentation.

5 Evaluation

To demonstrate the performance and flexibility of our framework, we explore a few examples in this section. We first compare d3p's runtime performance to TensorFlow on the implementation of a variational autoencoder (Sec. 5.1). Afterwards, we show some results for the hierarchical logistic regression model discussed in detail in Sec. 3.3 and use that opportunity to highlight some privacy trade-offs in the regime of small data sets (Sec. 5.3). Finally, we briefly compare results of d3p to the original implementation of the DP-VI algorithm in [18] (Sec. 5.4) on a Gaussian mixture model.[5]

5.1 Comparison with TensorFlow Privacy

We first compare the performance of our d3p package to an implementation of an identical model using the TensorFlow Probability framework [7] with a manual implementation of the variational inference algorithm and privacy enabled by the TensorFlow Privacy package [33]. In our comparison we focus on TensorFlow Privacy only, but refer the reader to [36] for a more extensive performance comparison of a JAX-based optimised implementation of the closely related DP-SGD algorithm with a range of current DP-enabling packages for popular machine learning frameworks.

We choose a variational auto-encoder (VAE) model [23] for this purpose. VAEs are generative models that consist of an encoder function, mapping data to parameters of a distribution on latent representations, and a decoder function, converting samples from the latent space to data samples. Generating data consists of drawing a sample from the distribution in the latent space and passing it through the decoder function. These mapping functions are represented by neural networks and therefore typically have a large number of parameters.

VAEs therefore provide an excellent test case for performance and have been previously used for the same purpose [3]. We use slight variations of the same model on different image classification data sets, namely MNIST [26], Fashion-MNIST [43] and CIFAR-10 [25]. For MNIST and Fashion-MNIST datasets, we use feedforward networks with a single hidden layer encoder and decoder, consisting of 688 884 trainable parameters in total. For CIFAR-10 dataset, we employ networks of 3 convolutional layers followed by a single dense layer, consisting of 640 423 parameters in total.

Of particular interest are the respective runtime of the inference as well as a comparison of the inferred model to verify that d3p is fast and accurate. Table 2 shows the runtime per epoch as well as the loss on the held-out test set after 20 *epochs*, i.e., passes over the training data set of 60 000 images (50 000 for CIFAR-10). We use a minibatch size of 128. The reported numbers are averages over 20 training processes that were run using a single NVIDIA Tesla V100 32G GPU. All runs used the Adam optimiser [22] for parameter updates after computing gradients. For the DP variants we used $\sigma = 1.5$, resulting in $\varepsilon \approx 0.5$ for $\delta = 1/60000$ ($\delta = 1/50000$ for CIFAR).

From the comparison, we can see that the end losses with MNIST and Fashion-MNIST data sets are comparable across frameworks, and all cases were observed to converge well. However, we found the CIFAR-10 data set to be more challenging to train due to large variation in samples, relatively modest number of samples per class and three colour channels. Neither framework was able to learn good representations of CIFAR-10 with DP for our choice of hyperparameters. The aforementioned challenges may also explain the higher discrepancy be-

[5] All our experiments used the d3p revision of commit f57e6935.

Data Set	Framework	DP-VI		Non-private VI	
		Wall Time [s]	Final Loss	Wall Time [s]	Final Loss
MNIST	d3p	0.56 ± 0.00	174.06 ± 0.59	0.34 ± 0.00	99.63 ± 0.30
	TF	6.43 ± 0.17	171.26 ± 3.67	1.58 ± 0.05	105.50 ± 1.84
Fashion	d3p	0.55 ± 0.01	304.74 ± 0.68	0.33 ± 0.00	243.99 ± 0.29
	TF	7.19 ± 0.14	303.14 ± 8.79	1.70 ± 0.06	244.40 ± 7.56
CIFAR	d3p	4.22 ± 0.01	2123.05 ± 0.25	1.64 ± 0.01	1903.29 ± 2.14
	TF	49.34 ± 0.07	2129.20 ± 0.26	2.23 ± 0.05	2038.92 ± 11.06

Table 2. Performance comparison of d3p against TensorFlow probability with TensorFlow Privacy. d3p achieves significant speed-up over TensorFlow Privacy with similar loss. Values shown in the left half of the table are runtime per epoch as well as the final loss (negative ELBO, lower is better) value on test set after 20 epochs for the differentially-private VI in both frameworks, using the Adam optimiser for parameter updates in all cases. The right part of the table shows the same results for non-private inference, where NumPyro takes the place of d3p.

Data Set	Sampler	Wall Time [s]	Final Loss
MNIST	Feistel	0.56 ± 0.00	174.06 ± 0.59
	Built-in	0.66 ± 0.01	174.04 ± 0.61
Fashion	Feistel	0.55 ± 0.01	304.74 ± 0.68
	Built-in	0.65 ± 0.01	304.75 ± 0.61
CIFAR	Feistel	4.22 ± 0.01	2123.05 ± 0.25
	Built-in	4.31 ± 0.01	2123.05 ± 0.018

Table 3. Performance comparison of the d3p Feistel-based minibatch sampler compared to sampling based on JAX default routines.

tween the non-DP losses. In terms of performance, d3p consistently outperforms the TensorFlow (TF) implementation by a factor of ∼10 for all data sets. Additionally, the relative performance loss of DP-VI compared to non-private inference in the same framework is lower in d3p (up to ∼ 2.5-fold for CIFAR-10) compared to the TF implementation (up to ∼ 22-fold).

For d3p, we also compare using the Feistel-based GPU-optimised minibatch sampler against using JAX's built-in `jax.random.choice` method and summarise the results in Table 3. We observe that our optimised sampler consistently yields a ∼ 100 ms speed-up (15% on MNIST) per epoch on all data sets. This similarity is due to the similar sizes of the data sets resulting in similar amounts of total iterations and thus invocations of the subsampling.

5.2 Effect of Batch Size

While the vectorised implementation of the per-instance gradient computation and manipulation eliminates much of the time overhead required in the DP-VI algorithm, it does not remove it completely. This is due to the additional steps in DP-VI, such as gradient clipping, summing and perturbing, but may also be an effect of a larger memory footprint due to holding gradient values for each data instance intermittently. This necessitates a larger amount of memory accesses, which can slow down computation. We therefore investigate the effect of minibatch size on the runtime of the DP-VI implementation in d3p for MNIST. In Figure 3 we plot the average runtime per iteration (over 100 iterations) over the size of minibatches for DP-VI in d3p and non-private inference in NumPyro. We see that, as expected, runtime per iteration increases more steeply for DP-VI than the non-private case. However, we also observe that DP-VI runtime increases only linearly with minibatch size, which is in line with our expectations.

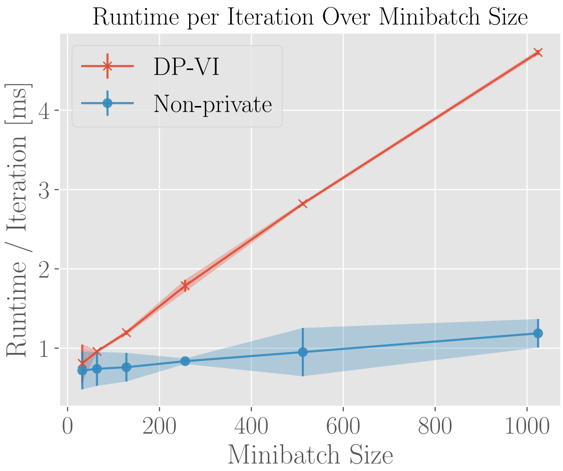

Fig. 3. Influence of minibatch size on runtime performance with and without differential privacy. Results are averages over 100 iterations. Error bars and shaded area indicate standard error.

5.3 Hierarchical Logistic Regression and Small Data

We next evaluate the implementation of the hierarchical logistic regression model presented in Section 3.3 with small data sets. These are generally problematic in privacy-preserving inference because every single data point has a comparatively larger effect on the outcome. Achieving good model performance therefore requires looser privacy bounds than in the case with larger data sets. d3p makes it easy for the user explore the trade-off between privacy and utility via its fast inference and the ease of adjusting the algorithm by simply specifying the privacy parameters from which the perturbation noise is automatically derived.

In the following experiments we use synthetic training data sets of varying size N that follow the hierarchical structure described in Section 3.3. Each data point is five-dimensional and points are split into $L = 3$ groups that are in turn described by $K = 3$ variables each. To evaluate the goodness-of-fit of the trained model we evaluate the area under the ROC curve (AUC) on a held-out test set of the same size N as the training set. The ROC curve plots the true positive rate over the false positive rate and the AUC is thus a summary measure of the inherent trade-off between those, independent of the choice of the decision boundary. An AUC of 1 corresponds to a perfect classifier and an AUC of 0.5 to random guessing. A higher AUC therefore indicates a more robust and powerful classifier, which makes it a suitable metric for our experiments.

Figure 4 shows the AUC after training on a data set with $N = 500$ data points for 100 000 iterations. Results are shown for different levels of privacy as well as non-private variational inference for ten runs with different random seeds. Privacy bound δ was kept fixed to $\frac{1}{N}$. Each run took less than 15 seconds on a commodity laptop without GPU acceleration. The red line is the AUC for non-privately fitting a simple (non-hierarchical) logistic regression model using scikit-learn [31]. The trade-off between privacy and utility is clearly visible: Smaller values of ε corresponding to stricter privacy constraints lead to lower AUC on average and a larger spread of results over different runs. Runs for $\varepsilon = 4$ or larger are close to the AUC of the non-private model on average but exhibit larger spread. Runs for $\varepsilon = 2$ fall short of this but still outperform the simpler baseline. This highlights a strength of the probabilistic modelling approach particularly relevant for privacy-preserving machine learning: By encoding prior knowledge of the generative process underlying the data in a principled way, privacy budget does not need to be spent to learn this a-priori known structure, allowing more capacity to learn the remaining parameters of the model. Curiously, for $\varepsilon = 8$ we observe on average a higher AUC than for the non-private model. This appears to not be a random artifact of the limited number of repetitions in our experiment as the results are similar for an increased number of 200 repetitions (not shown). We are uncertain why this is the case, but suspect it could be a regularising effect of clipping and the small random perturbations of the gradients. Note however that this is just an average result and any individual run with $\varepsilon = 8$ can still turn out to be worse than any non-private run, as indicated by the spread of the corresponding boxplots, which is in line with expectations.

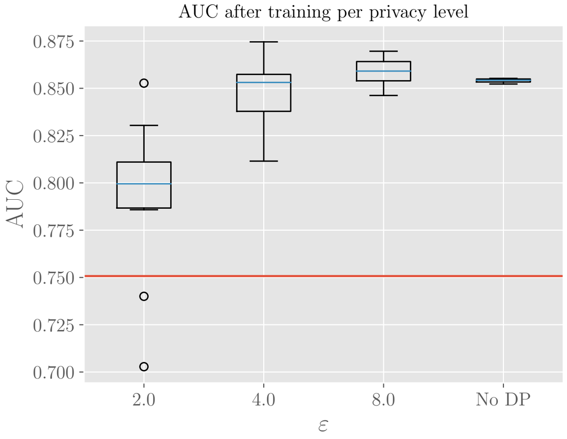

Fig. 4. AUC of the hierarchical logistic regression model after training on 500 data points. Ten runs of 100 000 iterations each were performed for each level of privacy and a non-private run, the resulting distribution of AUC values are shown as box plots. The red line indicates the AUC of a non-private non-hierarchical logistic regression model as a simple baseline.

Figure 5 shows the effect of data set size for the same privacy levels after training for 100 000 iterations. The graphs show the spread of one standard deviation above and below the mean over ten runs for each data set size and privacy level. Smaller data sets result in lower average AUC and larger spread. For data sets of less than 500 records utility deteriorates rapidly.

We finally explore the effects of the number of training iterations. Since the amount of noise added to perturb gradients in the DP-VI algorithm increases with the number of iterations, one could expect that choosing too large a number of iterations will negatively effect the learning. Figure 6 shows the evolution of AUC for $N = 500$ for different numbers of total training iterations. Note that these are results of separate runs each

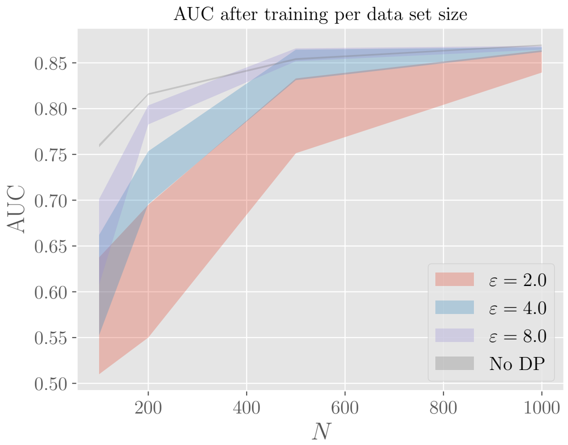

Fig. 5. AUC of the hierarchical logistic regression model after training on data sets of different sizes (100, 200, 500, 1000) for different privacy levels and non-privately. The model was trained 10 times for 100 000 iterations for each data set size and privacy bound. The graphs show the area within one standard deviation above and below the mean of the resulting distribution of AUC values.

with a different number of iterations and thus different amounts of perturbation per iteration, *not* the evolution of results over a single long training run.

We observe clearly that due to the privacy perturbations of gradients during training, the DP-VI algorithm takes longer to converge than non-private variational inference. Stricter privacy bounds move convergence to higher iteration counts. Contrary to expectation, despite the larger perturbations required for larger iteration counts, we see a general trend in improved utility for longer training (for $N = 100$, this trend continues up to 500 000 iterations). In our experiments we observe no negative impact of increasing the iteration count on the final AUC even if the training converges earlier, indicating that the DP-VI algorithm is very robust to the privacy perturbations.

5.4 Gaussian Mixture Model

We further demonstrate the ease of specifying expressive models in d3p by replicating an experiment on a Gaussian mixture model from the original DP-VI paper [18]. They used two-dimensional data generated from 5 clusters of spherical Gaussians and trained the model for 1 000 iterations for different levels of privacy. The evaluation is in terms of log-likelihood of a held-out test set on the learned predictive model.

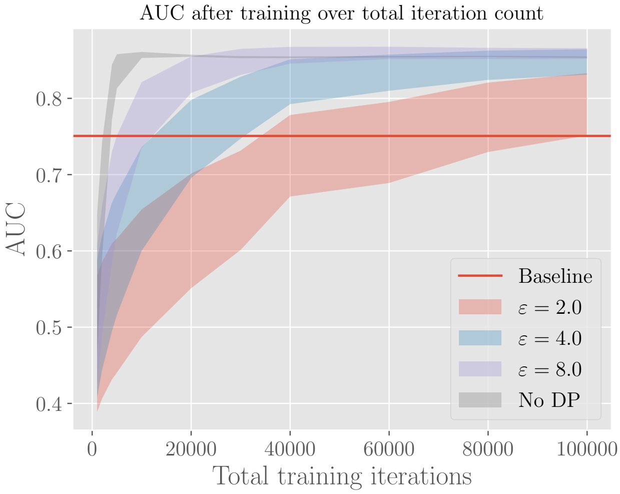

Fig. 6. AUC of the hierarchical logistic regression model after training on $N = 500$ data points for different amounts of training iterations. Plotted as in Figure 5. The additional horizontal line shows the AUC of the simple non-private non-hierarchical logistic regression baseline.

The original code[6] required model specific implementation of the DP-VI algorithm due to the absence of a generic framework for privacy-preserving probabilistic modelling. Using d3p it suffices to simply write out the model and an implementation of log-probability calculations and sampling routines for a Gaussian mixture distribution as an implementation of NumPyro's `Distribution` class. These are shown in Appendix A.

Figure 7 shows the resulting log-likelihood (higher is better) of the test data after training the model using the d3p implementation and the original code of [18]. Shown are the average with standard error over five runs in both cases. To keep results comparable we use the levels of privacy perturbation (parameter σ) from the original code for d3p. The results from d3p are very consistent with to slightly better than those from the original implementation but exhibit much less variability due to randomness in the inference algorithm, which is particularly pronounced for $\varepsilon = 0.1$ in the original paper code.

6 Related Work

Major popular probabilistic programming frameworks for the Python programming language are Edward2 [39], TensorFlow Probability [7], (Num)Pyro [3, 32] , PyMC

[6] Available at https://github.com/DPBayes/DPVI-code/

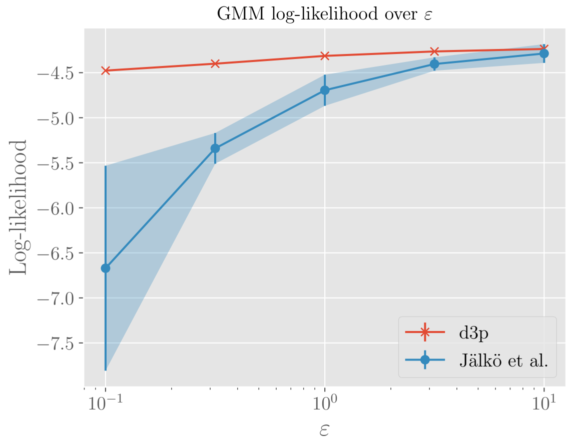

Fig. 7. Log-likelihood for test data of the Gaussian mixture model implemented with d3p and the DP-VI implementation of Jälkö et al.[18]. Graphs show average over five runs with error bars and shaded area indicating standard error (negligible for d3p).

[14], and Stan [6]. Edward2, TensorFlow Probability, Pyro and PyMC allow the user to declare models and run the inference from the Python programming language and differ mainly in the computation framework they rely on to run the inference (Edward2 and TensorFlow Probability use TensorFlow [1], Pyro uses PyTorch [30] and PyMC uses Theano [37]; NumPyro is a direct port of Pyro to JAX). The Stan framework follows a different approach and requires models to be specified in a dedicated domain-specific language, which is then evaluated using the Stan runtime, which can be invoked from Python or other major programming languages. None of these frameworks currently offers support for privacy-preserving inference.

A number of general implementations for differentially private machine learning exist for the popular frameworks. Notable are TensorFlow Privacy [33] for TensorFlow, Opacus [12] and PyVacy [41] for PyTorch. These generally provide implementations of the DP-SGD algorithm as alternative optimisers for the computational framework. In principle they could be combined with the dominant probabilistic programming framework for the respective backend, however this integration is usually not as seamless as one would desire. These implementations of DP also often suffer from poor performance in the implementation that can usually be traced back to inefficient computation of per-instance gradients [36]. With d3p we aim to provide better integration with high performance by directly targeting the NumPyro probabilistic programming framework. We are not aware of any general library of DP-SGD for the JAX framework that could be used with NumPyro to achieve the same goal. As far as we are aware, none of these libraries takes active measures to address the previously discussed technical issues of implementing differential privacy.

7 Conclusion

We have presented our d3p package which extends the NumPyro probabilistic programming framework with runtime efficient differentially private inference. We demonstrated the use of our framework and the expressiveness of the probabilistic programming approach on an extensive example and highlighted the requirements and corresponding implementation choices for our software. Our goal is to provide a helpful tool that encourages use of probabilistic programming as a viable approach to modelling data for privacy practitioners, as well as lowers the threshold for adoption of privacy-preserving methods for probabilistic modelling experts.

For future work our main focus is addressing the remaining technical implementation issues of differential privacy in real computer systems, namely predictable random number generation and finite-precision number representation (cf. Sec. 2.5), as the main obstacle for deployment in production settings. Promising solutions for this are (1) use of a cryptographically secure random number generator (CSPRNG) for DP perturbation and minibatch sampling and (2) adoption of the discrete Gaussian mechanism [5, 21].

Acknowledgements

This work was supported by the Academy of Finland (Flagship programme: Finnish Center for Artificial Intelligence FCAI, and grants 292334, 319264, 325572 and 325573) and by the Strategic Research Council at the Academy of Finland (grant 336032). We also acknowledge the computational resources provided by the Aalto Science-IT Project.

References

[1] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

[2] Martin Abadi et al. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 308–318, 2016.

[3] Eli Bingham et al. Pyro: Deep Universal Probabilistic Programming. *arXiv preprint arXiv:1810.09538*, 2018.

[4] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. JAX: composable transformations of Python+NumPy programs. https://github.com/google/jax, 2018.

[5] Clément L Canonne, Gautam Kamath, and Thomas Steinke. The discrete gaussian for differential privacy. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 15676–15688. Curran Associates, Inc., 2020.

[6] Bob Carpenter et al. Stan: a probabilistic programming language. *Journal of Statistical Software*, 76(1), 2017.

[7] Joshua V. Dillon et al. Tensorflow distributions. *arXiv preprint arXiv:1711.10604*, 2017.

[8] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

[9] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. Our data, ourselves: Privacy via distributed noise generation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 486–503. Springer, 2006.

[10] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference*, pages 265–284. Springer, 2006.

[11] Úlfar Erlingsson, Ilya Mironov, Ananth Raghunathan, and Shuang Song. That which we call private. *arXiv preprint arXiv:1908.03566*, 2019.

[12] Facebook. Opacus. https://opacus.ai/, 2020.

[13] Horst Feistel. Cryptography and computer privacy. *Scientific american*, 228(5):15–23, 1973.

[14] Chris Fonnesbeck, Anand Patil, David Huard, and John Salvatier. PyMC: Bayesian stochastic modelling in python. *Astrophysics Source Code Library*, 2015.

[15] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 2018.

[16] Simson L. Garfinkel and Philip Leclerc. Randomness concerns when deploying differential privacy. In *Proceedings of the 19th Workshop on Privacy in the Electronic Society*, WPES'20, page 73–86, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380867. 10.1145/3411497.3420211.

[17] Charles R. Harris et al. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. 10.1038/s41586-020-2649-2.

[18] Joonas Jälkö, Onur Dikmen, and Antti Honkela. Differentially private variational inference for non-conjugate models. In *Uncertainty in Artificial Intelligence 2017 Proceedings of the 33rd Conference, UAI 2017*. The Association for Uncertainty in Artificial Intelligence, 2017.

[19] Joonas Jälkö, Eemil Lagerspetz, Jari Haukka, Sasu Tarkoma, Antti Honkela, and Samuel Kaski. Privacy-preserving data sharing via probabilistic modeling. *Patterns*, 2(7):100271, 2021. ISSN 2666-3899. 10.1016/j.patter.2021.100271.

[20] Michael I. Jordan, Zoubin Ghahramani, Tommi S. Jaakkola, and Lawrence K. Saul. An introduction to variational methods for graphical models. *Machine learning*, 37(2):183–233, 1999.

[21] Peter Kairouz, Ziyu Liu, and Thomas Steinke. The distributed discrete gaussian mechanism for federated learning with secure aggregation. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 5201–5212. PMLR, 18–24 Jul 2021.

[22] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR 2015)*, 2015.

[23] Diederik P. Kingma and Max Welling. Auto-encoding variational Bayes. In *2nd International Conference on Learning Representations (ICLR 2014)*, 2014.

[24] Antti Koskela, Joonas Jälkö, and Antti Honkela. Computing tight differential privacy guarantees using FFT. In *International Conference on Artificial Intelligence and Statistics*, pages 2560–2569. PMLR, 2020.

[25] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

[26] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[27] Michael Luby and Charles Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing*, 17(2):373–386, 1988.

[28] Ilya Mironov, Omkant Pandey, Omer Reingold, and Salil Vadhan. Computational differential privacy. In *Annual International Cryptology Conference*, pages 126–142. Springer, 2009.

[29] Rory Mitchell, Daniel Stokes, Eibe Frank, and Geoffrey Holmes. Bandwidth-optimal random shuffling for GPUs. *arXiv preprint arXiv:2106.06161*, abs/2106.06161, 2021.

[30] Adam Paszke et al. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[31] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[32] Du Phan, Neeraj Pradhan, and Martin Jankowiak. Composable effects for flexible and accelerated probabilistic programming in NumPyro. *arXiv preprint arXiv:1912.11554*, 2019.

[33] Carey Radebaugh and Ulfar Erlingsson. Introducing TensorFlow privacy: Learning with differential privacy for training data. TensorFlow Blog, https://blog.tensorflow.org/2019/03/introducing-tensorflow-privacy-learning.html, 2019.

[34] Shuang Song, Kamalika Chaudhuri, and Anand D. Sarwate. Stochastic gradient descent with differentially private updates. In *2013 IEEE Global Conference on Signal and Information Processing*, pages 245–248. IEEE, 2013.

[35] Daniel Stokes and Rory Mitchell. CUDA-Shuffle: GPU shuffle using bijective functions. https://github.com/djns99/CUDA-Shuffle, 2021.

[36] Pranav Subramani, Nicholas Vadivelu, and Gautam Kamath. Enabling fast differentially private SGD via just-in-time compilation and vectorization. *arXiv preprint arXiv:2010.09063*, 2020.

[37] Theano Development Team. Theano: A python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688*, 2016.

[38] Michalis Titsias and Miguel Lázaro-Gredilla. Doubly stochastic variational Bayes for non-conjugate inference. In *International conference on machine learning*, pages 1971–1979, 2014.

[39] Dustin Tran et al. Simple, distributed, and accelerated probabilistic programming. In *Neural Information Processing Systems*, 2018.

[40] Martin J. Wainwright and Michael Irwin Jordan. *Graphical models, exponential families, and variational inference*. Now Publishers Inc, 2008.

[41] Chris Waites. PyVacy. https://github.com/ChrisWaites/pyvacy, 2019.

[42] George Y. Wong and William M. Mason. The hierarchical logistic regression model for multilevel analysis. *Journal of the American Statistical Association*, 80(391):513–524, 1985. ISSN 01621459.

[43] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

A Code for the Gaussian Mixture Model

We present the implement for the Gaussian mixture model (GMM) used in the experiment in Section 5.4 in Listings 5 and 6.

Mathematically a GMM can be specified as

$$\begin{aligned} p(\boldsymbol{x}_i|z_i) &= \mathcal{N}\left(\boldsymbol{x}_i; \boldsymbol{\mu}_{z_i}, \boldsymbol{\Sigma}_{z_i}\right), \\ p(z_i) &= \text{Categorical}\left(\pi_1, \dots, \pi_K\right) \end{aligned}$$

where $z_i \in \{1, \dots, K\}$ is a latent variable that indicates the mixture component that sample $\boldsymbol{x}_i$. Conditioned on z_i, $\boldsymbol{x}_i$ follows a regular Normal distribution. The categorical probabilities π_j and the parameters $\boldsymbol{\mu}_j$, $\boldsymbol{\Sigma}_j$ of the mixture components are the parameters of the GMM.

Sampling from the GMM can therefore be implemented by first sampling z_i from the Categorical distribution, then sampling $\boldsymbol{x}_i$ from the Normal distribution indicated by z_i. This is presented in the `sample` function of the `GaussianMixtureModel` class in Listing 5.

Subclassing `Distribution` enables us to provide a method for computing the log-probability of the Gaussian mixture where we marginalise out the latent variables to avoid these issues following [18]:

$$\log p(\boldsymbol{x}_i) = \log \sum_{j=1}^{K} \left(\pi_j \mathcal{N}\left(\boldsymbol{x}_i; \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j\right)\right).$$

This is implemented for batched data by the `log_prob` method in Listing 5 using JAX's highly performant vectorised mapping capabilities.

Having encapsulated the sampling and log-probability of the Gaussian mixture model in a NumPyro `Distribution`, we can easily make use of it in our `model` shown in Listing 6 and only need to specify the prior distributions for the parameters of the model. Following [18], we use a Dirichlet distribution for $\pi_1, \dots, \pi_K$ and zero-centered Normal priors for $\boldsymbol{\mu}_j$. We assume that each component is spherical, i.e., $\boldsymbol{\Sigma}_j = \sigma_j^2 \boldsymbol{I}$ and use the Inverse Gamma distribution as prior for σ_j. The code in Listing 6 reflects this using the same imperative sampling instructions demonstrated in the earlier examples.

We note that, in principle, it would also have been possible to implement the sampling steps of the GMM steps directly in the `model` function without the need of subclassing NumPyro's `Distribution` class. However, this would require learning the values of the latent variables z_i for each data record during inference, which presents a problem for private inference [18]. The resulting need for a specific implementation of the marginalised log-probability is what makes the GMM an interesting example for the flexibility and expressiveness of NumPyro-based models for privacy-preserving probabilistic programming in d3p. Note that, compared to the implementation of the experiment in [18], we did not have to concern our implementation with effects of reparametrisation on gradients and other details of DP-VI but focus on providing a straightforward implementation of the model.

```
class GaussianMixtureModel(Distribution):

    def __init__(self, mixture_probabilities,
    mixture_locs, mixture_scales):
        self._pis = mixture_probabilities
        self._locs = mixture_locs
        self._scales = mixture_scales

        batch_shape = ()
        event_shape = self._locs.shape[1:]
        super().__init__(
            batch_shape, event_shape)

    def sample(self, rng_key, sample_shape=()):
        zs_rng, xs_rng = \
            jax.random.split(rng_key)
        zs = CategoricalProbs(self._pis)\
            .sample(zs_rng, sample_shape)
        xs = Normal(
            self._locs[zs], self._scales[zs]
        ).sample(xs_rng)
        return xs

    def log_prob(self, value):
        per_component_log_prob = jax.vmap(
            lambda loc, scale: Normal(
                loc, scale
            ).log_prob(value),
            out_axes=-1
        )(self._locs, self._scales)

        log_pis = jnp.log(self._pis)

        # sum log-likelihood contributions
        # from event dimensions
        per_component_log_prob =\
            per_component_log_prob.sum(axis=-2)

        # aggregate over components
        loglik = logsumexp(
            per_component_log_prob + log_pis,
            axis=-1
        )
        return loglik
}
```

Listing 5. Implementation of a Gaussian mixture model distribution in NumPyro with log-likelihood marginalised over the latent component assignments.

```
def model(xs, N, k=5, d=2):
    pis = sample('pis', Dirichlet(jnp.ones(k)))

    with plate('component_priors', k, k):
        mus = sample('locs',
            MultivariateNormal(
                jnp.zeros((d,)), jnp.eye(d)
            ), sample_shape=(k,)
        )
        sigmas = sample('sigmas',
            InverseGamma(1, 1),
            sample_shape=(k,)
        )

    batch_size = xs.shape[0]
    with plate('batch', N, batch_size):
        sample(
            'xs', GaussianMixtureModel(
                pis, mus, sigmas
            ),
            obs=xs, sample_shape=(batch_size,)
        )
```

Listing 6. Definition of the model for a Gausian mixture model, using the `GaussianMixtureModel` distribution class defined in Listing 5.