Enabling Fast Differentially Private SGD via Just-in-Time Compilation and Vectorization

Pranav Subramani* Nicholas Vadivelu† Gautam Kamath‡

October 20, 2020

Abstract

A common pain point in differentially private machine learning is the significant runtime overhead incurred when executing Differentially Private Stochastic Gradient Descent (DPSGD), which may be as large as two orders of magnitude. We thoroughly demonstrate that by exploiting powerful language primitives, including vectorization, just-in-time compilation, and static graph optimization, one can dramatically reduce these overheads, in many cases nearly matching the best non-private running times. These gains are realized in two frameworks: JAX and TensorFlow. JAX provides rich support for these primitives as core features of the language through the XLA compiler. We also rebuild core parts of TensorFlow Privacy, integrating features from TensorFlow 2 as well as XLA compilation, granting significant memory and runtime improvements over the current release version. These approaches allow us to achieve up to 50x speedups in comparison to the best alternatives. Our code is available at https://github.com/TheSalon/fast-dpsgd.

1 Introduction

Machine learning has recently experienced tremedous growth, being used to solve problems with unprecedented accuracy in a myriad of domains. However, not all domains are alike. One may wish to train a model on a dataset which is publicly available—for instance, training an digit classifier on the popular (public) MNIST database [LBBH98]. On the other hand, applications frequently involve datasets which are in some way *private*, potentially containing data which is personal information, or otherwise proprietary. To provide some examples, consider a medical application, where one wishes to classify whether or not images contain a tumor. Alternatively, one can imagine a retail company training a machine learning model based on valuable market research data. In both cases, it is of paramount importance that the trained model does not leak information about the training data, with consequences ranging from loss of revenue to legal liability. Troublingly, it has been demonstrated that disregarding these concerns can result in significant leakage of private information—for example, a language model naïvely trained on a sensitive dataset may end up regurgitating Social Security numbers [CLE+19]. Furthermore, many heuristic and best-effort privacy approaches (such as data anonymization) have been demonstrated to be non-private [SS98, BDF+18].

*Cheriton School of Computer Science, University of Waterloo. pranav.subramani@uwaterloo.ca.
†Cheriton School of Computer Science, University of Waterloo. nbvadive@uwaterloo.ca.
‡Cheriton School of Computer Science, University of Waterloo. g@csail.mit.edu. Supported by an NSERC Discovery grant, a Compute Canada RRG grant, and a University of Waterloo startup grant.

In order to assuage concerns of private information leakage, in 2006, Dwork, McSherry, Nissim, and Smith [DMNS06] introduced the celebrated notion of differential privacy (DP). This principled and rigorous measure is widely accepted as a strong standard for privacy-preserving data analysis. Informally speaking, an algorithm is said to be differentially private if its distribution over outputs is insensitive to the addition or removal of a single datapoint from the dataset. Differential privacy has enjoyed widespread adoption in practice, including deployments by Apple [Dif17], Google [EPK14, BEM+17], Microsoft [DKY17], and the US Census Bureau for the 2020 Census [DLS+17].

One of the workhorse algorithms in machine learning is stochastic gradient descent (SGD), which is effective at training machine learning models in rather general settings. A differentially private analogue, DPSGD [SCS13, BST14, ACG+16], has been introduced as a drop-in replacement for SGD. This algorithm can (informally) be described as iteration of the following simple procedure:

1. Draw a minibatch from the training dataset of appropriate size;
2. For each point (x_i, y_i) in the minibatch:
   (a) Compute the gradient g_i of the objective function at (x_i, y_i);
   (b) "Clip" the gradient: if $\|g_i\|_2$ is greater than some hyperparameter threshold C, rescale g_i so that $\|g_i\|_2 = C$;
3. Aggregate the clipped gradients in the minibatch, and add Gaussian noise of sufficient magnitude to guarantee differential privacy;
4. Update the parameters of the model by taking a step in the direction of the noised aggregated gradient.

The primary differences with respect to (non-private) SGD are the per-example clipping and noising operations. While these modifications seem relatively innocuous, they have so far led to non-trivial costs in terms of both running time as well as accuracy of the trained model. In this paper, we address and mitigate the running time overhead of DPSGD.[1]

Taking a step back: as we alluded to before, the deep learning revolution has been catalyzed by advances in graphics processing units (GPUs) and similar devices which allow simultaneous processing of several datapoints at once. More precisely, they allow one to compute gradients for each example, which are then aggregated. This specific procedure is ubiquitous in machine learning, and has thus been highly optimized in most machine learning frameworks.

In fact, one could even consider this procedure to be *too* optimized: in vanilla machine learning settings, one simply needs the gradient as averaged over a minibatch, and not for each individual point. Consequently, modern machine learning frameworks generally do not allow access to gradients for individual points, also known as *per-example gradients*. Access to these objects is critical for the clipping procedure in DPSGD, as well as other applications of independent interest beyond privacy, for instance optimization based on importance sampling [ZZ15]. If one wishes to generate per-example gradients, the immediate solution is to process the gradients one by one, thus losing all advantage bestowed by parallel processing on GPUs and creating massive overheads in terms of the running time. Lack of support for fast computation of per-example gradients has been noted and lamented numerous times in discussion forums and GitHub issues for both TensorFlow [Lip16, see16, act17] and PyTorch [Kun18, ale18]. In the context of PyTorch, co-creators

[1]This article concentrates on DPSGD, as it is the most commonly run private algorithm on large-scale datasets. However, we note that this per-example clipping operation is present in numerous differentially private algorithms [KV18, KLSU19, KSU20, BDKU20], and similar methods might be useful in performance optimization in these settings as well.

Chintala and Paszke have both commented on this issue, stating in 2018 that "this is currently impossible with [auto differentiation] techniques we use" [Pas18], due to limitations with the THNN and cuDNN backends [Chi17], and adding support for this functionality would require "chang[ing] 5k+ lines of code" [Chi18].

Numerous attempts to avoid these computational roadblocks have been proposed. Goodfellow [Goo15] proposed an algorithmic solution for computing per-example ℓ_2-norms of the gradients for fully-connected networks.[2] Other proposed solutions work by exploiting Jacobians [DKH20] or parallelizing over the batch dimension [AG19]. Several of these approaches are are restricted to specific types of architectures—for example, [Goo15] is restricted to fully-connected layers, though [RMT19] extends this to convolutional layers. BackPACK [DKH20] currently supports only fully connected and convolutional layers, and while the paper states that it can be extended to recurrent and residual layers, GitHub issues related to implementation of these features have been open since November 2019 [Dan19]. Very recently (roughly a month prior to submission of this paper), Facebook revealed Opacus, the release version of their differentially private machine learning library previously known as PyTorch-DP [Fac20a]. Their primary advertised points are speed and scalability, which we take as strong evidence for the importance of fast differentially private machine learning. We briefly mention microbatching [MAE+18], in which small "microbatches" of points are averaged before clipping, reducing the number of per-example clipping operations (and thus the running time) at the cost of requiring additional noise to achieve the same privacy guarantee. Since this generally results in significantly worse accuracy, we do not investigate it further in our work. A more thorough description of approaches is provided in Section 2.1.

As mentioned in the literature (and thoroughly explored in this paper), all existing approaches seem to incur moderate to severe running time overhead versus non-private SGD, with slowdowns as large as two orders of magnitude. For instance, Carlini et al. [CLE+19] comment, "Training a differentially private algorithm is known to be slower than standard training; our implementation of this algorithm is 10-100x slower than standard training," where their implementation is based on TensorFlow Privacy. Additionally, Thomas et al. [TAD+20] (working in an unspecified framework) document a slowdown from 12 minutes to 14 hours due to the introduction of differential privacy, a 70x slowdown. The effect of these slowdowns can range from an inconvenience when it comes to rapid prototyping of smaller models, to prohibitively expensive for a single training run of a larger model. Overcoming this obstacle is an important step in helping differentially private machine learning transition from its present nascent state to widespread adoption.

1.1 Results

We demonstrate that one can mostly eliminate the significant running time overhead of differentially private SGD by exploiting language primitives such as vectorization, just-in-time (JIT) compilation, and static graph optimization. These features are core primitives within JAX [FJL18, BFH+18] and TensorFlow 2 (TF2) [ABC+16], both tensor-processing libraries by researchers at Google. These frameworks combine JIT compilation backed by the Accelerated Linear Algebra (XLA) [Gooe] compiler with auto differentiation for high-performance machine learning. As we will see, JAX is consistently the fastest method for running DPSGD, with running times comparable to the non-private case. Our custom TensorFlow Privacy (TFP) implementation (referred to as Custom TFP), which takes advantage of new features in TensorFlow 2 as well as XLA, demonstrates similar performance to JAX and significantly outperforms the existing TFP library.

Our primary contributions are as follows:

[2]Though he was motivated by other applications [ZZ15], these are the exact objects we require for DPSGD.

1. We thoroughly benchmark several frameworks and libraries for DPSGD.
2. We extend TensorFlow Privacy to support language features present in TF2 as well as XLA compilation, significantly improving its running time in most cases.
3. We demonstrate that methods which use vectorization, JIT compilation, and static graph optimization are consistently the fastest and more memory-efficient: specifically, JAX and our modification of TFP.
4. We find that, despite similarities in the compilation pipeline, JAX is generally faster than our customized TFP. We examine and discuss compiled XLA assembly to explain the discrepancy.
5. We provide developers of DP machine learning libraries and frameworks with recommendations to optimize running time performance.
6. Finally, we publicly release code to reproduce these experiments, as well as guide researchers and engineers in producing fast code for differentially private machine learning. Code is available at `https://github.com/TheSalon/fast-dpsgd`.

Table 1 summarizes some of our experimental results, with median running time per epoch for a variety of settings.

Architecture	Private JAX/Custom TFP	Best Private Alternative	Best Non-Private
Logistic Regression	**0.23**	0.48	0.12
FCNN	**0.21**	0.77	0.20
MNIST CNN	**0.53**	6.50	0.42
CIFAR10 CNN	**7.3**	12	1.7
Embedding	**0.23**	3.8	0.11
LSTM	**8.2**	407	3.6

Table 1: Median running time (s) per epoch of training various models at batch size 128, both with and without differential privacy, comparing our suggested solutions (JAX and Custom TFP) with other frameworks. FCNN stands for Fully-Connected Neural Network, CNN stands for Convolutional Neural Network, and LSTM stands for Long-Short Term Memory network. JAX or Custom TFP are consistently the fastest private options with little overhead over the best non-private variants.

We observe dramatic improvements for embedding network and LSTMs [HS97], for the latter, potentially significant enough to bring LSTMs from impractical into the realm of feasibility. JAX is able to privately train these models 17x and 50x faster than the best alternative.[3] Examining the overhead due to privacy: JAX's running time increases by roughly 2x, compared to factors closer to 10x for alternatives. We take this as evidence that optimization and improvements to the core JAX framework (which is still relatively young) will translate to further advantages for private training.

For the fully-connected and MNIST convolutional networks, JAX or Custom TFP almost entirely remove the overhead due to privacy. In fact, the running times are significantly better than some alternatives *without* privacy. Recall that these are per-epoch times: while an improvement of

[3]Note that a confirmed bug in TF2 currently prevents us from running Custom TFP in these cases [Vad20b].

0.5 seconds might seem insignificant, this can add up when training for many epochs. We perform an ablation study (Tables 2 and 3) for some models to pinpoint the source of all improvements.

While there has been significant work towards making DPSGD run faster, they are without exception built *on top* of frameworks such as PyTorch and TensorFlow. By investigating the effect of low-level language features, we thus provide a qualitatively different (and more systems-focused) perspective on the problem, which we hope will refocus efforts by the community to speed up private machine learning. We discuss how other frameworks can take advantage of this perspective in Section 4.1. Briefly, we recommend improved support for vectorization and JIT compilation.

While our investigations show the consistent and substantial superiority of JAX for fast private machine learning, these benefits remain relatively unknown. Though a small number of experts are aware [Tal20], and the official JAX repo contains a toy demonstration [Goo19], a Google Scholar search revealed only two papers which use JAX for differential privacy [WC20, PTS+20], and neither emphasizes or even comments on the computational advantages of JAX. We hope that our investigation will document this phenomenon and encourage others to adopt it for their private machine learning needs.

Recent Developments. In the several weeks prior to posting this paper, there have been a number of recent and simultaneous works focused on fast differentially private machine learning. First is Opacus [Fac20a], Facebook's library for differentially private machine learning, built on top of PyTorch. Opacus is advertised with emphasis on its speed and scalability, so it is a good point of comparison which we include in our experimental benchmarks. Even more recently, Lee and Kifer [LK20] extended the chain-rule-based method used in [Goo15, RMT19] to architectures beyond fully-connected and convolutional layers. Additionally, an anonymous preprint uses Johnson-Lindenstrauss projections to quickly approximate per-example gradient norms [Ano21]. Note that this method is an algorithmic modification, and will not be functionally identical to DPSGD—similar to microbatching, there is a time-accuracy tradeoff (though not as severe in this case). Since these two works appeared in the last month, and neither provides code, we do not compare with them at this time.

2 Description of Approaches

2.1 Libraries Enabling DPSGD

JAX [FJL18, BFH+18]. JAX is a recently introduced framework for machine learning, defined by its automatic differentiation capabilities and JIT compilation via the XLA compiler [Gooe]. Programs written in pure Python and JAX's NumPy [HMvdW+20] API can be translated to an intermediate language (XLA-HLO) to generate custom kernels, enabling optimizations such as kernel-fusions, buffer reuse, improved memory layout, and more. Additionally, one of the core functions present in JAX is `VMAP`, a vectorized map, which enables easy-to-write and efficient batch level parallelism that is fundamental to DPSGD. As we will demonstrate, these enable the fastest approach for DPSGD that we are aware of.

Custom TFP. Vectorization and XLA-driven JIT compilation is also available in TensorFlow 2 [ABC+16], which we leverage in our implementation, Custom TFP. With these primitives, we achieve performance comparable to JAX and surpassing existing DPSGD implementations in TensorFlow. We augment TensorFlow Privacy to better utilize `tf.vectorized_map` and follow TensorFlow 2 best practices while retaining the existing functionality.

Chain-Rule-Based Per-Example Gradients [Goo15, RMT19]. This suite of techniques is implemented on top of PyTorch [PGM$^+$19]. They support efficient GPU-accelerated per-example gradients for fully-connected layers via [Goo15], as well as convolutional layers via [RMT19], which we describe in the detail in the following paragraphs.

Let C, D, T, and B refer to the number of input channels, output channels, the spatial dimension, and the batch size. The shape of the input x is (B, C, T). The conventional formula for the discrete convolution can be written as:

$$\sum_{c=0}^{C-1} \sum_{k=0}^{K-1} x[b, c, t+K] h[d, c, k].$$

The gradient of this expression can be efficiently computed via automatic differentiation [Ral81]. PyTorch's automatic differentiation cannot be parallelized across the batch dimension b [RMT19], which is required to backpropagate through the above expression. Instead, they rewrite the convolution as follows:

$$\sum_{c=0}^{C/G-1} \sum_{k=0}^{K-1} x\left[b, c, g\frac{C}{G}, t+K\right] h[d, g, c, k],$$

where G is the number of groups and the shape of x is $(1, B, C, T)$. The initial convolution is 1-dimensional, while the above expression includes an added dimension. Similarly, to allow back-propagation through a k-dimensional convolutional layer, a $(k+1)$-dimensional convolutional layer is required. This can be achieved by utilizing the `group` attribute in the convolution function in PyTorch, since splitting it into groups implies that the same convolution is applied to each individual group.

BackPACK [DKH20]. The chain rule gives the following expression for the gradient of a loss function:

$$\nabla_{\theta^{(i)}} \ell(\theta) = (J_{\theta^{(i)}} z_n^{(i)})^T \left(\prod_{j=i}^{L-1} (J_{z_n^{(j)}} z_n^{(j+1)})^T \right) (\nabla_{z_n^{(L)}} \ell_n(\theta)).$$

In order to compute this quantity, one requires the ability to multiply the Jacobian by a vector and by a matrix, which is not currently supported in PyTorch's automatic differentiation framework. In BackPACK, Dangel et al. [DKH20] extend several layers within PyTorch to support fast Jacobian-vector and Jacobian-matrix products in order to extract quantities like individual gradients, variance, ℓ_2-norm of the gradients, and second-order quantities. In particular, to extract first-order gradients, their method multiplies the transposed Jacobian with the outputs of the layer:

$$\frac{1}{N} \nabla_{\theta^{(i)}} \ell(\theta) = \frac{1}{N} (J_{\theta^{(i)}} z_n^{(i)})^T (\nabla_{z_n^{(i)}} \ell(\theta)),$$

where $i = 1, \ldots, N$ and each $\theta^{(i)}$ has a gradient which is of shape $(N, d^{(i)})$. BackPACK provides efficient computation for the transpose of the Jacobian as well as the Jacobian. This method compares favourably to extracting gradients sequentially via a for-loop. The authors mention that the cost of extracting the individual gradients does come at a minor overhead when compared to regular training.[4]

[4]By regular training we mean evaluating the gradients on the average loss for the batch.

Opacus [Fac20a]. Opacus is a library for training PyTorch models with differential privacy, recently released by Facebook. It supports per-example gradients, using PyTorch's forward and backward hooks to propagate gradients. They provide support for several PyTorch layers including LSTM layers, which are not supported in either of the previous two frameworks. Note that Opacus does not support PyTorch's `nn.LSTM` but instead implements a separate `opacus.layers.DPLSTM`, with adjustments that allow individual gradients to propagate through it.

PyVacy [Wai19]. Before the release of Opacus (and its predecessor PyTorch-DP), PyVacy was the most popular library for DP machine learning in PyTorch. PyVacy has no custom support for parallelization across the batch dimension for any layer since it processes each sample individually (by way of a for-loop). This generally leads to a large increase in runtime for models trained using PyVacy.

TensorFlow Privacy [PGC18]. TensorFlow Privacy is a library for differentially private machine learning, built on top of TensorFlow. TensorFlow Privacy has general support for a vectorized implementation of DPSGD. In particular, it has a function `vectorized_map` which allows it to parallelize across the batch dimension, used to extract per-example gradients. It should be noted that according to the documentation of the function [Good], the memory footprint of `vectorized_map` is much larger than the alternative, which is `map_fn`. This tradeoff comes with the advantage that it runs significantly faster and is another tool for removing the necessity of microbatching.

2.2 Notable Framework Features

Static versus Dynamic Graph. TensorFlow and JAX use a *static graph* to track computation in order to optimize execution and compute gradients. This means the sequence of operations is traced and a large proportion of shapes are determined during the first invocation of the function, allowing for kernel fusion, buffer reuse, and other optimizations on subsequent calls. PyTorch uses a *dynamic graph* to track computation flow in order to compute gradients, but does *not* optimize execution. This enables increased dynamism in the shapes and types of computations, at the cost of losing all the aforementioned optimizations.

Grappler versus XLA. TensorFlow has two optimization engines: Grappler [Goob] and XLA. Grappler, TensorFlow's original graph optimizer, takes as input the computation graph and is able to prune dead nodes, remove redundant computation, improve memory layouts, and more. XLA, TensorFlow's new optimizing just-in-compiler, can perform the same optimizations as Grappler, in addition to generating code for fused kernels. For this reason, XLA has the potential to extract more performance out of TensorFlow graphs than Grappler, but does not always accomplish this due to Grappler's maturity.

JAX and XLA. JAX was built from the ground up to leverage XLA, and so many of its operations map directly to XLA primitives. We often observe that JAX is able to extract better performance out of XLA than TensorFlow.

Pytorch and Static Graphs. Recently, PyTorch has released the capability to JIT compile its code through `torch.jit` or PyTorch XLA [Fac20b]. Due to the early nature of these two efforts, they were not successful in JIT compiling the methods we tried, thus we do not consider them further.

3 Empirical Findings

In this section, we present our experimental findings, comparing and benchmarking various alternatives for DPSGD. Our code is publicly available at `https://github.com/TheSalon/fast-dpsgd`.

We evaluate the aforementioned implementations of DPSGD in runtime and memory consumption on four datasets: **MNIST** [LeC98], a handwritten digit recognition dataset with 60,000 training images of size 28×28 each, **CIFAR10** [Kri09], a dataset of small colour images with 60,000 training examples of size 32×32×3 each, **IMDb** [MDP$^+$11], a movie review sentiment classification dataset with 25,000 training examples padded to a sequence length of 256 each, and **Adult** [DG17], containing 45,220 examples with 104 features, which was preprocessed via methods from [INS$^+$19].

We perform our evaluations on six different architectures. We start with the smallest dataset, Adult, training a 105-parameter logistic regression model and a 5,532-parameter fully-connected neural network (FCNN). Next, we train an MNIST classifier, using a convolutional neural network architecture with 26,010 parameters which we refer to as MNIST CNN. Then, we train a CIFAR10 convolutional neural network classifier architecture with 605,226 parameters used by Papernot et al. [PTS$^+$20]. We refer to this network as CIFAR10 CNN. For IMDb, we use an LSTM network and Embedding network with 1,081,002 and 160,098 parameters, respectively. Full descriptions of our architectures appear in Section B. This selection covers the common data and architecture types at realistic sizes for differentially private learning. In particular, we did not consider the exceptionally large models which are now prevalent in non-private machine learning. This is because the noise introduced by DPSGD scales with the square root of the number of parameters, so such large models typically get trivial accuracy, and more modest-sized architectures are preferred. These architectures were selected because they resemble the models used in the DPSGD tutorials released by TensorFlow Privacy and Opacus. The fully connected network and the logistic regression model are a mainstay in classification and therefore were included. The LSTM model with this architecture is undocumented to the best of our knowledge and is an example of a larger model.

We compare a number of different methods for fast DPSGD, including JAX [FJL18, BFH$^+$18], BackPACK [DKH20], the chain-rule based method (CRB) [Goo15, RMT19], Opacus [Fac20a], PyVacy [Wai19], TensorFlow Privacy (TFP) [PGC18] and our modification of TFP, dubbed Custom TFP. For TensorFlow based frameworks, we evaluate performance both with and without XLA JIT compilation. We refer to TensorFlow 2 and JAX as *modern XLA-compiled libraries*, since the XLA integration in the TensorFlow 1 implementations is limited. When possible, we base our implementations on the idiomatic examples provided by the libraries.

These architectures and datasets are evaluated in terms of runtime at batch sizes 16, 32, 64, 128, and 256. This showcases a comparison of runtimes across a variety of batch sizes to present a holistic picture of the running times as opposed to edge-case performances, as well as demonstrating the impact of memory utilization on runtime. Each experiment was run for 20 epochs and the median epoch run-time is reported. Outside of the initial compilation time required for the static graph frameworks, the runtime showed little variance between epochs, resulting in narrow confidence intervals for the epoch runtime (data available upon request). We preprocess all the data in advance and use an identical generator-based dataloader for all frameworks, to ensure consistency. The data is stored in the framework-appropriate array/tensor in advance on the CPU, then is transferred to the GPU before the forward pass (as is common practice).

All experiments were run on Ubuntu 18.04 with an Intel Core i7-7800X CPU (3.50 GHz, 6 cores), NVIDIA GTX Titan V GPU (12GB VRAM), and 32GB of RAM. Though we do not include these results in the paper, we verified that the relative performance of these libraries generalized to a consumer grade setup by running these experiments with an Intel i7-8850H (2.60GHz, 6 cores), NVIDIA GTX 1050 Ti Max-Q (4GB VRAM), and 16GB of RAM.

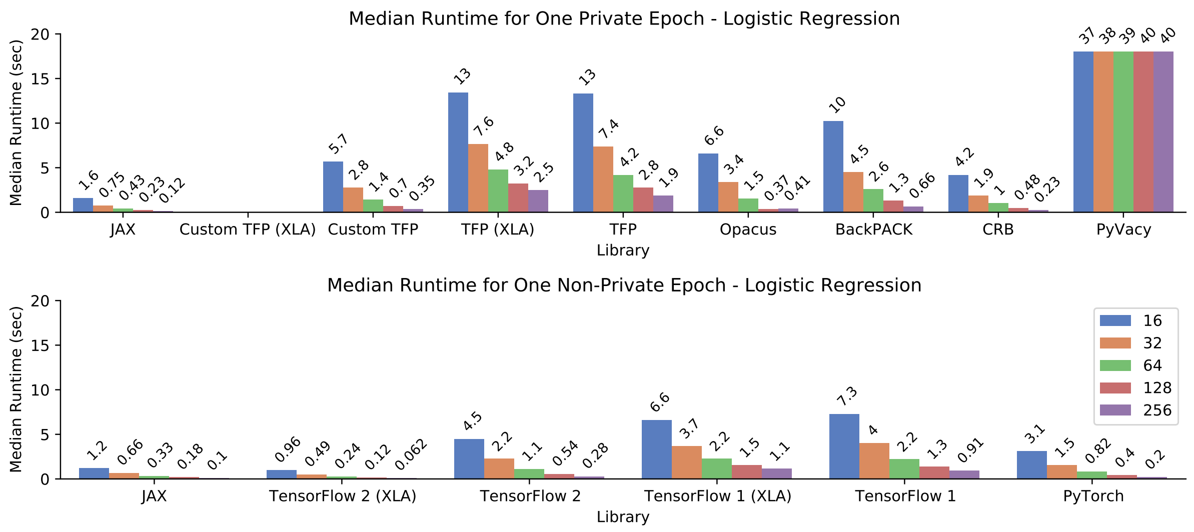

Figure 1: **Runtimes for logistic regression on the Adult dataset**. With privacy, JAX is the fastest, comparable to the non-private runtimes. We were unable to benchmark Custom TFP due to an open TensorFlow 2 bug [Vad20a]. The y-axis is truncated for clarity.

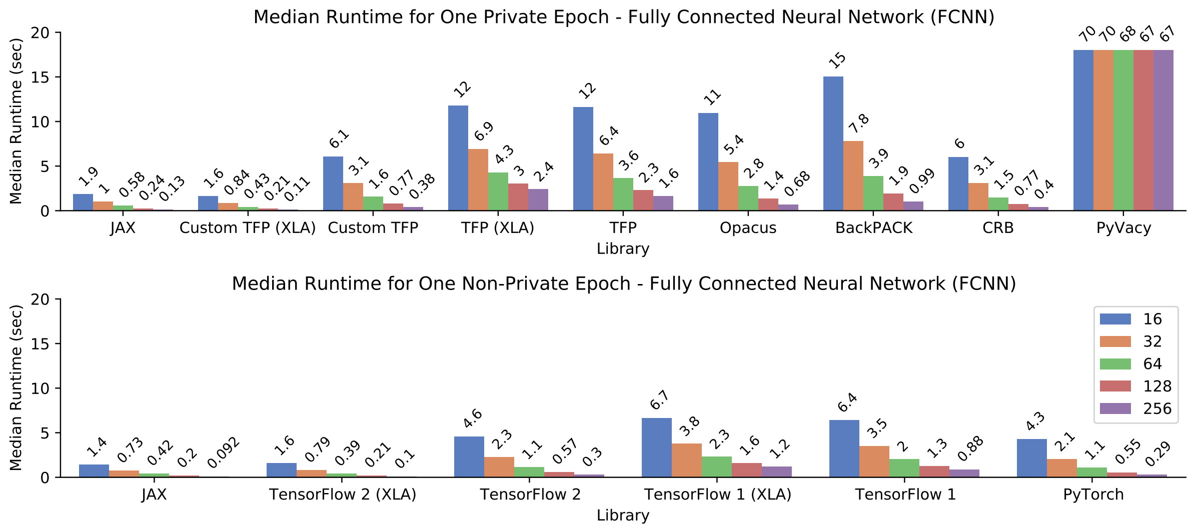

Figure 2: **Runtimes for the fully connected network on the Adult dataset**. We observe that JAX and Custom TFP are the fastest by a large margin in both settings, with DPSGD having low overhead over the non-private setting. The y-axis is truncated for clarity.

Our first experiment (Figure 1) is on the runtime performance of the logistic regression model. We see that JAX has by far the most performant implementation of DPSGD. Custom TFP was unable to run due to a confirmed bug in TensorFlow 2 [Vad20a]. However, considering TensorFlow 2 (XLA)'s performance in the non-private setting, we would expect similarly strong performance in the private setting once this is resolved.

Next, we evaluate the FCNN model (Figure 2), where we observe that the two modern XLA-compiled implementations (JAX and Custom TFP) are significantly faster than the other options

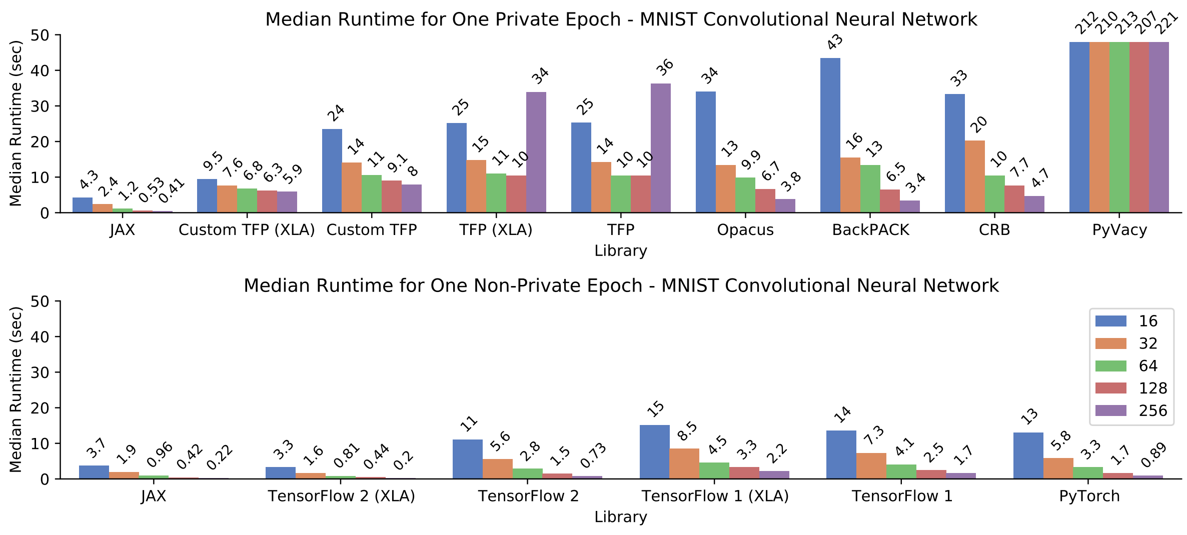

Figure 3: **Runtimes for the CNN on the MNIST dataset.** We observe that JAX has the fastest runtime in the private case while TensorFlow 2 with XLA is the fastest in the non-private case. TFP struggles at the largest batch size due to an inability to properly parallelize per-gradient computation with this level of memory consumption. The y-axis is truncated for clarity.

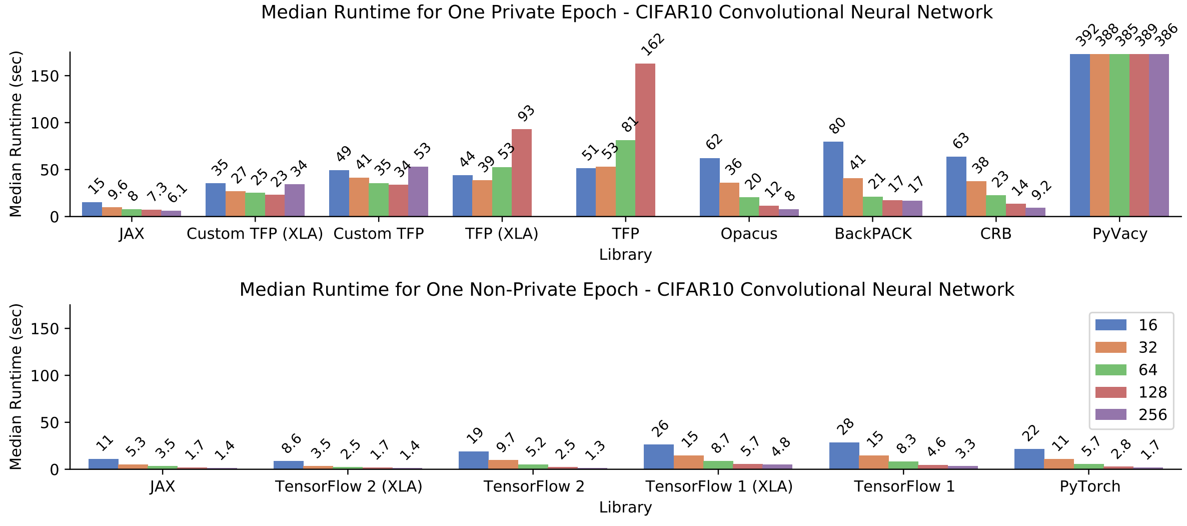

Figure 4: **Runtimes for the CNN on the CIFAR10 dataset.** We observe that JAX has the fastest runtime in the private case while TensorFlow 2 with XLA and JAX are the fastest in the non-private case at most batch sizes. Similar to the MNIST case, TFP struggles at the largest batch size due to an inability to properly parallelize per-gradient computation with this level of memory consumption. The y-axis is truncated for clarity.

in both the private and non-private setting. With such a simple architecture, the compiler can perform significant optimizations. Notably, JAX and Custom TFP show little overhead over their non-private counterparts. The non-statically compiled frameworks (apart from PyVacy) remain competitive in this setting due to the shallow network size and low parameter count.

We then evaluate the MNIST CNN model (Figure 3), where JAX is by far the most performant implementation in the private case, and the modern XLA-compiled frameworks are the fastest non-private methods. Custom TFP is noticeably slower than JAX—we conjecture that this is due to different utilization of the JIT compiler, see Section 4. We see the PyTorch-based frameworks are much slower at small batch sizes: in this regime, the Python overhead is too high for the asynchronous execution to hide the latency. Also, at the largest batch size, TFP's performance deteriorates due to the memory consumption, so it is no longer able to effectively vectorize the per-example gradient computation.

The CIFAR10 CNN model runtimes (Figure 4) demonstrate similar trends to the MNIST CNN. In the TFP case, we observe a more exaggerated performance deterioration at larger batch sizes due to the larger model, and begin to see similar effects in Custom TFP as well, showing performance worse than the PyTorch frameworks at the largest batch sizes. The PyTorch libraries show closer performance to the compiled libraries at lower batch sizes unlike the MNIST case: since the model and data are larger, PyTorch's asynchronous execution is able to hide the Python latency.

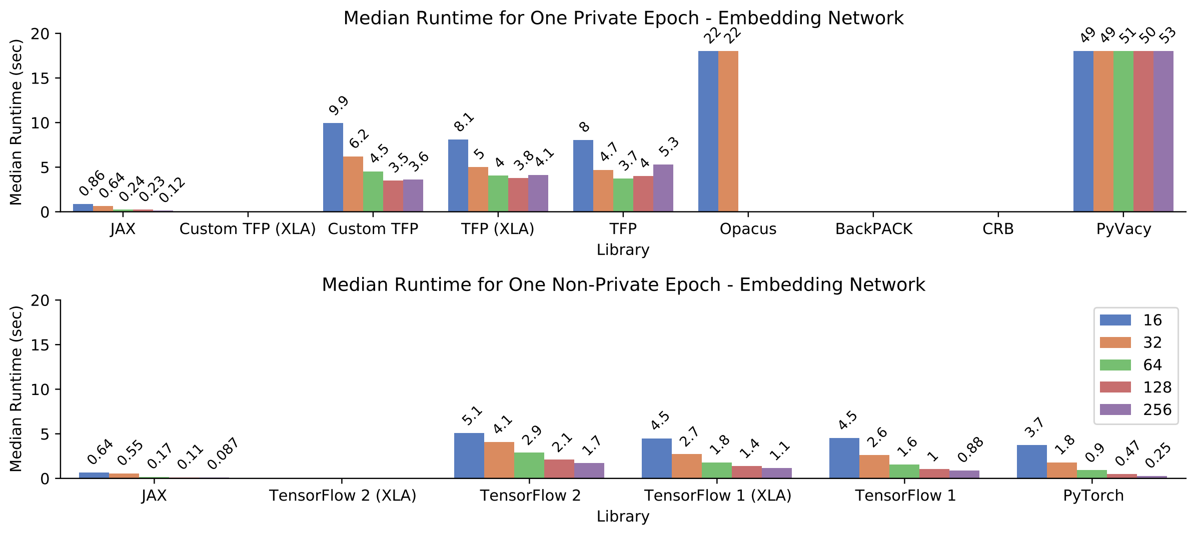

Figure 5: **Runtimes for the Embedding network on the IMDb dataset.** JAX is an order of magnitude faster than the best alternatives in the private setting. The quadratic memory cost of Opacus prevents evaluation at batch sizes larger than 32. An open TensorFlow 2 bug prevents us from evaluating Custom TFP (XLA) in this setting [Vad20b]. BackPACK and CRB do not support embedding layers. The y-axis is truncated for clarity.

Next, we evaluate the embedding network (Figure 5), where we see a more exaggerated difference between JAX and the other frameworks. A (different and also confirmed) TensorFlow 2 bug [Vad20b] and lack of support from BackPACK and CRB prevent us from evaluating this setting on those implementations. We see a similar phenomenon to the MNIST CNN case for TFP: at larger batch sizes, the library fails to parallelize, thus pessimizing the runtime.

For the final runtime experiment we evaluate the LSTM network (Figure 6) and observe similar trends to the embedding network. Here, however, TensorFlow and PyTorch benefit from a fast cuDNN LSTM implementation in the non-private case which they fail to leverage in the private case, explaining the significant difference in performance. JAX, on the other hand, uses an LSTM implementation based on primitive operations, which allows it to retain similar performance in both the private and non-private settings.

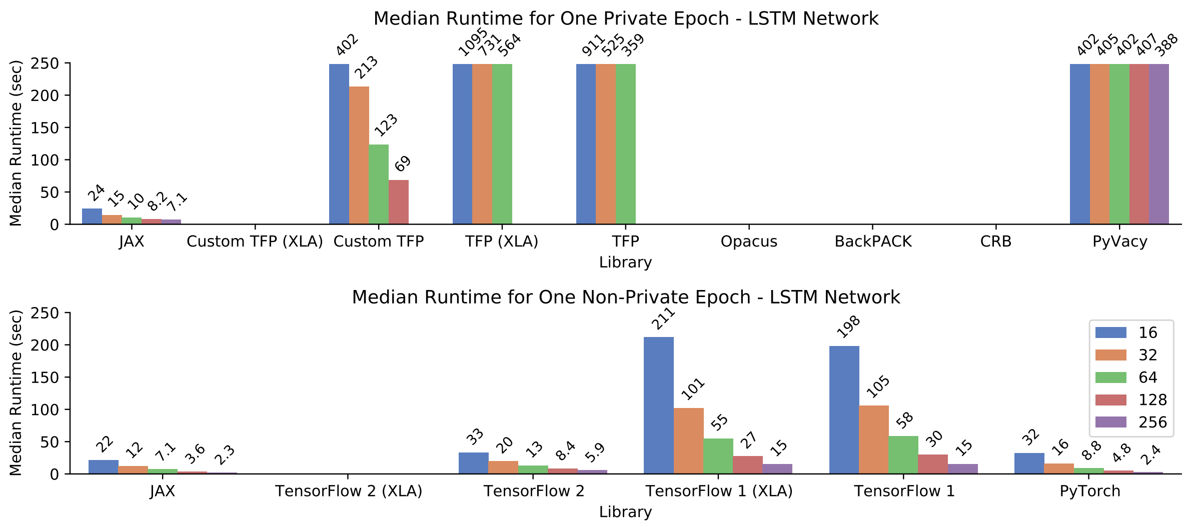

Figure 6: **Runtimes for the LSTM network on the IMDb dataset.** JAX is by far the fastest option, resulting in a roughly 50x speedup for batch size 256. The quadratic memory cost of Opacus prevents us from evaluating this implementation at these batch sizes, however, we observe a median runtime of 1024.16s at batch size 10. Excessive memory consumption prevent us from evaluating Custom TFP and TFP at larger batch sizes. An open TensorFlow 2 bug prevents us from evaluating Custom TFP (XLA) in this setting [Vad20b]. BackPACK and CRB do not support embedding layers. The y-axis is truncated for clarity.

`VMAP`	`JIT`	Logistic	FCNN	MNIST CNN	CIFAR10 CNN	Embedding
		1040	1240	1530	3840	856
✓		16.4	19.2	24.6	64.8	14.4
	✓	1.08	2.85	22.3	84.0	3.80
✓	✓	**0.231**	**0.239**	**0.535**	**7.28**	**0.231**

Table 2: **Ablation of JIT compilation and vectorization in JAX.** Median runtime per epoch (seconds) for a run of 20 epochs at batch size 128 for DPSGD in JAX. While JIT provides the largest boost in most cases, VMAP also provides a significant performance improvement. The two together provide the largest improvement. We exclude LSTMs, as JAX runs out of memory without JIT compilation for this model.

To understand the importance of vectorization (via `VMAP`) and JIT compilation (`JIT`), we ablate JAX's performance on these tasks with and without these two components (Table 2). We observe that `JIT` alone provides up to a 963x improvement, and `VMAP` alone provides up to a 64x improvement. While in most cases `JIT` provides the larger improvement, on the CIFAR10 CNN we observe that `VMAP` provides a greater reduction. This is because the model is large enough to sufficiently utilize the GPU with its convolutions and hide the Python overhead through asynchronous execution. When used in tandem both complement each other, providing a 5160x improvement.

Similarly, we ablate the components in Custom TFP (Table 3). In TensorFlow, vectorized map will automatically JIT compile your code, meaning we cannot ablate that component separately. We observe that XLA in TensorFlow 2 is able to reclaim performance lost by not vectorizing, seeing that the non-VMAP XLA performance comes close to VMAP Graph performance in many settings.

Mode	Logistic	FCNN	MNIST CNN	CIFAR10 CNN	Embedding	LSTM
Eager	86.3	147	303	561	163	717
Graph	9.93	12.7	32.5	85.5	22.2	361
XLA	0.76	1.83	19.5	71.5		
Graph+Vectorization	**0.703**	0.770	9.07	33.8	**3.53**	68.6
XLA+Vectorization		**0.209**	**6.28**	**23.3**		

Table 3: **Ablation of JIT compilation and vectorization in Custom TFP.** Median runtime per epoch for a run of 20 epochs at batch size 128 for DPSGD in Custom TFP. Eager uses no compilation or vectorization. Graph refers to TensorFlow's normal graph mode compilation, while XLA represents JIT compilation through XLA. In TensorFlow, one can not use vectorization without graph compilation, which is why there is no standalone vectorization. Empty entries are due to confirmed active bugs in TensorFlow [Vad20a, Vad20b].

We further see that the non-compiled runtimes are not as extreme as seen in JAX: TensorFlow is better rounded and can run in a reasonable amount of time in all settings.

Library	MNIST CNN	CIFAR10 CNN	IMDb LSTM
JAX	187,136	10,448	**11,984**
TensorFlow 2 (XLA)	**271,104**	**15,040**	
TensorFlow 2	179,328	11,328	9,221
TensorFlow 1 (XLA)	157,312	10,880	5,070
TensorFlow 1	186,368	11,480	5,264
PyTorch	113,664	10,752	9,943
JAX (DP)	116,480	**4,264**	**2,487**
Custom TFP (XLA)	**137,856**	3,144	
Custom TFP	57,216	1,944	137
TFP (XLA)	560	168	88
TFP	36,608	104	105
Opacus	36,608	1,920	10
BackPACK	34,048	1,216	
CRB	40,192	2,184	
PyVacy	∞	∞	∞

Table 4: **Maximum batch size supported by each library before encountering out of memory errors.** The top shows non-private training, while the bottom shows DPSGD. Hardware was a NVIDIA Titan V with 12 GB of VRAM. Missing entries represent missing functionality or bugs in the frameworks. PyVacy handles examples sequentially one at a time regardless of batch size, giving constant memory consumption with respect to batch size. We observe XLA-compiled frameworks (JAX, TensorFlow 2, TensorFlow 1) have superior memory management, except TensorFlow 1 in the MNIST CNN case, which we believe to be due to the limited capabilities of the XLA compiler in TensorFlow 1.

Finally, we explore the memory consumption behaviour of these implementations (Table 4), observing that running time has a strong negative correlation to memory consumption. The modern XLA-compiled libraries provide impressive batch size capability: JAX in the private setting for the

MNIST CNN can achieve a larger maximum batch size than PyTorch in the non-private case! Also, all the frameworks except JAX and PyVacy struggle with batch sizes on the LSTM. Since PyVacy processes examples sequentially, it has a constant memory consumption with respect to batch size, effectively trading off running time for optimal memory use. The other frameworks are crippled without access to their fused cuDNN LSTM implementation, while JAX has no issues as its LSTM is composed of primitives. Finally, due to the specialized per-example gradient computation afforded by CRB for convolutions, it shows the best memory utilization among the PyTorch frameworks, even beating Custom TFP (without XLA) in the CIFAR10 CNN case.

4 Discussion

JAX and Custom TFP's runtime advancements can be primarily attributed to the advancement of the static graph compiler present in both of these languages. The XLA compiler performs a variety of operations ranging from memory scheduling to kernel fusion. The memory optimizations are vital for larger models where DPSGD becomes a memory-bound algorithm. One of the core features of XLA is buffer reutilization which has a significant impact on the maximum memory used [Gooe]. Furthermore, the memory scheduler can mitigate peak memory usage to prevent a runtime exception for overusing available memory.

The effectiveness of XLA is demonstrated through the peak batch size experiment: in both the private and non-private settings, XLA far exceeds alternatives in the peak batch size it supports. Through this experiment, we also see the benefits of using small operation primitives as opposed to large fused kernels: TensorFlow and PyTorch both leverage the optimized cuDNN kernel in the non-private setting for performance [Gooc, Fac], but cannot in the private setting, leading to significantly worse performance. JAX instead focuses on optimizing operation primitives, so even in foreign computational circumstances, its performance is comparatively strong. Succinctly, JAX sacrifices the ability to use highly optimized fused kernels for generalizability.

We also observe TFP's significant performance deterioration with XLA in this experiment. Due to the nascency of the XLA compiler, it is more deeply integrated into the newer TensorFlow 2 than TensorFlow 1. Thus, the compiler does not as effectively optimize TensorFlow 1 code, and can sometimes lead to pessimization of performance. The primary documented and observed optimization is autoclustering of operations, which empirically results in suboptimal performance. This carries through to the remaining experiments as well: although we were able to use TensorFlow 1 with XLA, its performance is nowhere near as strong as JAX or TensorFlow 2.

We notice that the runtimes are different between Custom TFP and JAX despite having the same backend compiler. The primary reason for this is in difference in translation and compilation procedures between the two of them. For example, JAX has a feature called *omnistaging* [Joh20] which allows more computation to be visible to the compiler, enabling further optimizations. We did not find evidence or documentation for similar functionality in TensorFlow.

We also observe that simple code segments get translated to different XLA code. Moreover, these differences compound for the larger models dealt with in Section 3. In Section A, we present a running example to demonstrate the divergence between JAX and TensorFlow's XLA assembly (which is a representation of the XLA-compiled module). The difference begins to arise with a call to the gradient function in both frameworks. In our toy example, JAX is able to compile the entire computation graph into a single fused kernel, while TensorFlow compiles the graph into two kernels. In this case, we observe that TensorFlow is slower than JAX (albeit not by a significant margin). Similarly, there are larger differences in the XLA logs of the models discussed in Section 3 and these likely contribute to the different performance that is seen across JAX and TensorFlow notwithstanding the same backend.

However, it is not true by default that JAX's compiled code runs faster than TensorFlow 2's. A counter-example to this is seen in Figure 2 where we notice that JAX is slower than Custom TFP in the private setting. Thus, while evident that XLA and vectorization present tremendous speedups without loss of generality, their behaviour is rather different depending on the frontend.

Through the ablation study of `VMAP` and `JIT` in JAX shown in Table 2, we observe that both components complement each other and enable the level of performance we see. In general, we observe that JIT provides the larger performance gain, as shown with the FCNN and Embedding result. For the CNN, the large matrix operations coupled with JAX's asynchronous execution [LJ19] allow reasonable utilization of the GPU even without `VMAP`, which is why we observe less of an improvement from `JIT` alone in this experiment. The runtimes without these two primitives is significantly slower than the other frameworks—this is because JAX was built ground-up to leverage these primitives [BFH$^+$18].

In the ablation for Custom TFP in Table 3, we see some key differences with JAX. First, the mechanism for `VMAP` in TensorFlow 2 is different from that in JAX: JAX performs op-by-op batching without compilation, while TensorFlow does not [Good]. We observe that the non-compiled code still runs in a reasonable amount of time since TensorFlow is optimized to have a competitive eager execution when compared to PyTorch, while JAX is not. Also, in TensorFlow, XLA's JIT compilation without vectorization is often able to bring the runtime performance close to that of the graph mode with vectorization, implying that XLA is able to recognize and implement some parallelization even when the user does not explicitly request it. Finally, we observe the benefit of having a fast, fused implementation for LSTMs: while JAX ran out of memory outside of a compiled and vectorized context, Custom TFP is able to achieve reasonable runtimes by leveraging the fused kernel.

In the process of conducting our experiments, we observed a new release of JAX (version 0.1.75 to 0.2.0) during which we saw better compilation times and improved running times. This is a major advantage of aligning the differentially private machine learning community with language primitives in JAX and Custom TFP. Researchers and engineers alike can take advantage of advancements made to the language core. Additionally, the generalizability of JAX and Custom TFP are vital for enabling DPSGD across a variety of domains, and the inability of other frameworks to do so stagger the development of private machine learning.

4.1 Recommendations

JAX and TensorFlow present language primitives that are absent or upcoming in PyTorch [Zho20]. We encourage language developers to incorporate `VMAP` and utilize XLA as a compilation path to yield the best runtimes for DPSGD. While these constructs may not be the only way to enable fast DPSGD, given the relevancy of machine learning frameworks, they seem to be the shortest path to fast DPSGD across in the context of machine learning.

In terms of developing future differential privacy libraries and frameworks, there are several key takeaways. First, leveraging small, generalizable operations and compiler infrastructure are more reliable than hand tuned kernels. Especially in an emerging field like differential privacy, it is impossible to anticipate how a library or framework will be used, so optimizing for generality will aid in new research. We observe this in our setting, particularly with the LSTM: as soon as TensorFlow was unable to use the highly optimized cuDNN kernel, its performance dropped significantly, unlike JAX. Second, maintaining a small, precise API surface will produce a more robust system, as well as one more amenable to optimization. JAX's small API combined with its strict functional style allow its developers to more easily produce a library that is often more performant than mature counterparts. Larger APIs such as TensorFlow, while convenient for users, makes it much more prone to bugs and much harder to optimize, as evidenced by the bugs we faced with Custom TFP with XLA.

In industry, engineers must port models to production environments and in this setting, Custom TFP is an ideal choice. TensorFlow possesses a custom library for porting models to production, TensorFlow Extended (TFX) [Gooa]. As a result of the runtime reduction, engineers would be able to prototype and transport models from development environments to production environments while staying within the same framework. This mitigates potential bugs that could arise from differences in development and production. JAX is slowly making progress in this area with the experimental `jax2tf` converter [Goo20], which allows developers to convert a JAX function into a function which uses only TensorFlow operations, allowing deployment of JAX-developed models using TensorFlow infrastructure.

4.2 Drawbacks

While integrating XLA into DPSGD presents a massive runtime and memory advantage, there is a cost. XLA is a subset of all permissable operations in JAX and TensorFlow. As a result, a researcher or engineer has to be cognizant of the permissible operations to prevent possible errors when incorrectly using XLA. For example, if dynamic behavior such as `np.unique(x)` is enabled within an XLA compiled function, an exception is raised. More subtle errors involving unintended recompilation of code are also possible which can lead to enormous slowdowns.

Now we consider some of the criticisms that are specific to JAX. JAX's primary programming model is functional which diverges from PyTorch and TensorFlow's existing APIs. While this may be an advantage for users experienced in functional programming, there would be a cost to port a user over since it requires understanding the JAX programming paradigm. Moreover, JAX is also in version 0.2.0 compared to PyTorch's 1.6.0 and TensorFlow's 2.4.0 and as a result, its ecosystem is not as mature as its counterparts. This can present a roadblock to users of who rely on libraries built on top of PyTorch/TensorFlow for research and production.

5 Conclusion and Future Work

We have demonstrated that language primitives like vectorization, JIT compilation, and static graph optimization can dramatically improve the running time of private machine learning, realized by JAX and our Custom TFP. In particular, we find that using JAX can almost entirely remove the computational overhead introduced by DPSGD, thus alleviating a major pain point of private machine learning practitioners.

In our work, we focus on conventional set-ups for academic researchers; for future work, it would be insightful to explore the performance of distributed DPSGD, as distributed set-ups are becoming increasingly commonplace. Furthermore, implementing a PyTorch `JIT` compatible version of DPSGD could provide an alternative to TensorFlow and JAX, particularly if said implementation is compatible with PyTorch XLA. Though these two compilation systems are immature compared to TensorFlow and JAX, they are rapidly improving and should not be ignored. Apart from the facilities available in Python, there are powerful autodifferentiation methods in other more perfomant languages such as Julia [Jul] and Swift [WZRC] which are worthy of study.

Acknowledgments

We would like to thank Xi He and Om Thakkar for valuable feedback on drafts of this work. We would also like to thank several JAX, TensorFlow, and Opacus developers who helped answer our issues, including James Bradbury, Peter Buchlovsky, Peter Hawkins, Matthew Johnson, Karthik Prasad, Github handle ravikyram, and Qianli Scott Zhu.

References

[ABC+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiangg Zhen. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, pages 265–283. USENIX Association, 2016.

[ACG+16] Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *Proceedings of the 2016 ACM Conference on Computer and Communications Security*, CCS '16, pages 308–318, New York, NY, USA, 2016. ACM.

[act17] act65. Custom gradient aggregation methods. https://github.com/tensorflow/tensorflow/issues/15760, December 2017.

[AG19] Ashish Agarwal and Igor Ganichev. Auto-vectorizing tensorflow graphs: Jacobians, auto-batching and beyond. *arXiv preprint arXiv:1903.04243*, 2019.

[ale18] alexdepremia. [feature request] expanding gradient function with variances. https://github.com/pytorch/pytorch/issues/8897, June 2018.

[Ano21] Anonymous. Fast differentially private-sgd via jl projections. In *Submitted to International Conference on Learning Representations*, 2021. Under review.

[BDF+18] Abhishek Bhowmick, John Duchi, Julien Freudiger, Gaurav Kapoor, and Ryan Rogers. Protection against reconstruction and its applications in private federated learning. *arXiv preprint arXiv:1812.00984*, 2018.

[BDKU20] Sourav Biswas, Yihe Dong, Gautam Kamath, and Jonathan Ullman. Coinpress: Practical private mean and covariance estimation. *arXiv preprint arXiv:2006.06618*, 2020.

[BEM+17] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnes, and Bernhard Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, SOSP '17, pages 441–459, New York, NY, USA, 2017. ACM.

[BFH+18] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. JAX: composable transformations of Python+NumPy programs, 2018.

[BST14] Raef Bassily, Adam Smith, and Abhradeep Thakurta. Private empirical risk minimization: Efficient algorithms and tight error bounds. In *Proceedings of the 55th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '14, pages 464–473, Washington, DC, USA, 2014. IEEE Computer Society.

[Chi17] Soumith Chintala. https://discuss.pytorch.org/t/gradient-w-r-t-each-sample/1433/2, March 2017.

[Chi18] Soumith Chintala. https://github.com/pytorch/pytorch/issues/8897/#issuecomment-400412917, June 2018.

[CLE$^+$19] Nicholas Carlini, Chang Liu, Úlfar Erlingsson, Jernej Kos, and Dawn Song. The secret sharer: Evaluating and testing unintended memorization in neural networks. In *28th USENIX Security Symposium*, USENIX Security '19, pages 267–284. USENIX Association, 2019.

[Dan19] Felix Dangel. Support for recurrent units. https://github.com/f-dangel/backpack/issues/16, 2019.

[DG17] Dheeru Dua and Casey Graff. UCI machine learning repository. http://archive.ics.uci.edu/ml, 2017.

[Dif17] Differential Privacy Team, Apple. Learning with privacy at scale. https://machinelearning.apple.com/docs/learning-with-privacy-at-scale/appledifferentialprivacysystem.pdf, December 2017.

[DKH20] Felix Dangel, Frederik Kunstner, and Philipp Hennig. BackPACK: Packing more into backprop. In *Proceedings of the 8th International Conference on Learning Representations*, ICLR '20, 2020.

[DKY17] Bolin Ding, Janardhan Kulkarni, and Sergey Yekhanin. Collecting telemetry data privately. In *Advances in Neural Information Processing Systems 30*, NIPS '17, pages 3571–3580. Curran Associates, Inc., 2017.

[DLS$^+$17] Aref N. Dajani, Amy D. Lauger, Phyllis E. Singer, Daniel Kifer, Jerome P. Reiter, Ashwin Machanavajjhala, Simson L. Garfinkel, Scot A. Dahl, Matthew Graham, Vishesh Karwa, Hang Kim, Philip Lelerc, Ian M. Schmutte, William N. Sexton, Lars Vilhuber, and John M. Abowd. The modernization of statistical disclosure limitation at the U.S. census bureau, 2017. Presented at the September 2017 meeting of the Census Scientific Advisory Committee.

[DMNS06] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Proceedings of the 3rd Conference on Theory of Cryptography*, TCC '06, pages 265–284, Berlin, Heidelberg, 2006. Springer.

[EPK14] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. RAPPOR: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the 2014 ACM Conference on Computer and Communications Security*, CCS '14, pages 1054–1067, New York, NY, USA, 2014. ACM.

[Fac] FaceBook. Lstm. https://pytorch.org/docs/stable/_modules/torch/nn/modules/rnn.html#LSTM.

[Fac20a] Facebook. Opacus. https://opacus.ai/, August 2020.

[Fac20b] FaceBook. Pytorch xla, 2020.

[FJL18] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. In *The 1st Conference on Systems and Machine Learning*, SysML '18, 2018.

[Gooa] Google. Tensorflow extended. https://www.tensorflow.org/tfx.

[Goob] Google. Tensorflow graph optimization with grappler. https://www.tensorflow.org/guide/graph_optimization.

[Gooc] Google. tf.keras.layers.lstm. https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTM.

[Good] Google. tf.vectorized_map. https://www.tensorflow.org/api_docs/python/tf/vectorized_map.

[Gooe] Google. Xla: Optimizing compiler for machine learning. https://www.tensorflow.org/xla.

[Goo15] Ian Goodfellow. Efficient per-example gradient computations. *arXiv preprint arXiv:1510.01799*, 2015.

[Goo19] Google. differentially_private_sgd.py. https://github.com/google/jax/blob/master/examples/differentially_private_sgd.py, April 2019.

[Goo20] Google. Jax to tensorflow converter, 2020.

[HMvdW+20] Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with numpy. *Nature*, 585(7825):357–362, 2020.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[INS+19] Roger Iyengar, Joseph P Near, Dawn Song, Om Thakkar, Abhradeep Thakurta, and Lun Wang. Towards practical differentially private convex optimization. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, SP '19, pages 299–316, Washington, DC, USA, 2019. IEEE Computer Society.

[Joh20] Matthew James Johnson. Omnistaging. https://github.com/google/jax/blob/master/design_notes/omnistaging.md, September 2020.

[Jul] Julia. Juliadiff. https://www.juliadiff.org/.

[KLSU19] Gautam Kamath, Jerry Li, Vikrant Singhal, and Jonathan Ullman. Privately learning high-dimensional distributions. In *Proceedings of the 32nd Annual Conference on Learning Theory*, COLT '19, pages 1853–1902, 2019.

[Kri09] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

[KSU20] Gautam Kamath, Vikrant Singhal, and Jonathan Ullman. Private mean estimation of heavy-tailed distributions. In *Proceedings of the 33rd Annual Conference on Learning Theory*, COLT '20, 2020.

[Kun18] Frederik Kunstner. [feature request] simple and efficient way to get gradients of each element of a sum. `https://github.com/pytorch/pytorch/issues/7786`, May 2018.

[KV18] Vishesh Karwa and Salil Vadhan. Finite sample differentially private confidence intervals. In *Proceedings of the 9th Conference on Innovations in Theoretical Computer Science*, ITCS '18, pages 44:1–44:9, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[LBBH98] 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.

[LeC98] Yann LeCun. The mnist database of handwritten digits. *http://yann.lecun.com/exdb/mnist/*, 1998.

[Lip16] Zachary C. Lipton. Gradients of non-scalars (higher rank jacobians). `https://github.com/tensorflow/tensorflow/issues/675`, January 2016.

[LJ19] Anselm Levskaya and Matthew James Johnson. Asynchronous dispatch. `https://jax.readthedocs.io/en/latest/async_dispatch.html`, 2019.

[LK20] Jaewoo Lee and Daniel Kifer. Scaling up differentially private deep learning with fast per-example gradient clipping. *arXiv preprint arXiv:2009.03106*, 2020.

[MAE$^+$18] H Brendan McMahan, Galen Andrew, Ulfar Erlingsson, Steve Chien, Ilya Mironov, Nicolas Papernot, and Peter Kairouz. A general approach to adding differential privacy to iterative training procedures. *arXiv preprint arXiv:1812.06210*, 2018.

[MDP$^+$11] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics.

[Pas18] Adam Paszke. `https://github.com/pytorch/pytorch/issues/7786/#issuecomment-391308732`, May 2018.

[PGC18] Nicolas Papernot, Andrew Galen, and Steven Chien. Tensorflow privacy. `https://github.com/tensorflow/privacy`, December 2018.

[PGM$^+$19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[PTS+20] Nicolas Papernot, Abhradeep Thakurta, Shuang Song, Steve Chien, and Úlfar Erlingsson. Tempered sigmoid activations for deep learning with differential privacy. *arXiv preprint arXiv:2007.14191*, 2020.

[Ral81] Louis B Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer, 1981.

[RMT19] Gaspar Rochette, Andre Manoel, and Eric W Tramel. Efficient per-example gradient computations in convolutional neural networks. *arXiv preprint arXiv:1912.06015*, 2019.

[SCS13] Shuang Song, Kamalika Chaudhuri, and Anand D Sarwate. Stochastic gradient descent with differentially private updates. In *Proceedings of the 2013 IEEE Global Conference on Signal and Information Processing*, GlobalSIP '13, pages 245–248, Washington, DC, USA, 2013. IEEE Computer Society.

[see16] seerdecker. Provide unaggregated gradients tensors. https://github.com/tensorflow/tensorflow/issues/4897, October 2016.

[SS98] Pierangela Samarati and Latanya Sweeney. Generalizing data to provide anonymity when disclosing information. In *Proceedings of the 20th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '98, page 188, New York, NY, USA, 1998. ACM.

[TAD+20] Aleena Thomas, David Adelani, Ali Davody, Aditya Mogadala, and Dietrich Klakow. Investigating the impact of pre-trained word embeddings on memorization in neural networks. In *Proceedings of the 23rd International Conference on Text, Speech and Dialogue*, TSD '20, 2020.

[Tal20] Kunal Talwar. Personal communication, July 2020.

[Vad20a] Nicholas Vadivelu. Xla compilation bug. https://github.com/tensorflow/tensorflow/issues/43723, October 2020.

[Vad20b] Nicholas Vadivelu. Xla compilation does not work with embeddings layer. https://github.com/tensorflow/tensorflow/issues/43687, October 2020.

[Wai19] Chris Waites. Pyvacy. https://github.com/ChrisWaites/pyvacy/network, March 2019.

[WC20] Chris Waites and Rachel Cummings. Differentially private normalizing flows for privacy-preserving density estimation. *ICML Workshop on Invertible Neural Networks, Normalizing Flows, and Explicit Likelihood Models*, 2020.

[WZRC] Richard Wei, Dan Zheng, Marc Rasi, and Bart Chrzaszcz. Differentiable programming manifesto.

[Zho20] Richard Zhou. RFC: torch.vmap. https://github.com/pytorch/pytorch/issues/42368, 2020.

[ZZ15] Peilin Zhao and Tong Zhang. Stochastic optimization with importance sampling for regularized loss minimization. In *Proceedings of the 32nd International Conference on Machine Learning*, ICML '15, pages 1–9. JMLR, Inc., 2015.

A XLA Log Comparison

We first construct the simplest component to a neural network, an affine transformation (Figure 7), and observe the XLA logs post-optimization for them. The logs for the JAX and TensorFlow code are in Figures 8 and 9, respectively. Note that Figure 8 and Figure 9 are both text files, so they are not indented.

```
import jax.numpy as jnp
import jax
import numpy as np
import tensorflow as tf

W = np.random.randn(5, 5).astype(np.float32)  # 5 x 5 Weight Matrix
b = np.random.randn(5).astype(np.float32)  # 5 x 1 bias vector
x = np.random.randn(5).astype(np.float32)  # input x

@jax.jit  # enables XLA + JIT
def matvec(W, x, b):
    return jnp.dot(W, x) + b  # W * x + b

@tf.function(experimental_compile=True)  # enables XLA + JIT
def matvec2(W, x, b):
    return tf.linalg.matvec(W, x) + b
```

Figure 7: Matrix-Vector Product in JAX and TensorFlow

```
HloModule jit_fn1.8

%scalar_add_computation (scalar_lhs: f32[], scalar_rhs: f32[]) -> f32[] {
  %scalar_lhs = f32[] parameter(0)
  %scalar_rhs = f32[] parameter(1)
  ROOT %add = f32[] add(f32[] %scalar_lhs, f32[] %scalar_rhs)
}

%fused_computation (param_0.1: f32[5], param_1.2: f32[5,5],
 param_2.2: f32[5]) -> f32[5] {
  %param_1.2 = f32[5,5]{1,0} parameter(1)
  %param_2.2 = f32[5]{0} parameter(2)
  %broadcast.1 = f32[5,5]{1,0} broadcast(f32[5]{0} %param_2.2),
   dimensions={1}
  %multiply.1 = f32[5,5]{1,0} multiply(f32[5,5]{1,0} %param_1.2,
  f32[5,5]{1,0} %broadcast.1)
  %constant_1 = f32[] constant(0)
  %reduce.1 = f32[5]{0} reduce(f32[5,5]{1,0} %multiply.1, f32[] %constant_1),
            dimensions={1}, to_apply=%scalar_add_computation,
            metadata={op_type="dot_general" op_name="jit(fn1)/

dot_general[ dimension_numbers=(((1,), (0,)), ((), ()))
\n precision=None ]"
             source_file="<ipython-input-3-8e7c515249a1>"
              source_line=3}
  %param_0.1 = f32[5]{0} parameter(0)
  ROOT %add.1 = f32[5]{0} add(f32[5]{0} %reduce.1,
                f32[5]{0} %param_0.1),
                metadata={op_type="add" op_name="jit(fn1)/add"
                 source_file="<ipython-input-3-8e7c515249a1>"
                 source_line=3}
}

ENTRY %jit_fn1.8 (parameter.1: f32[5,5], parameter.2: f32[5],
 parameter.3: f32[5]) -> (f32[5]) {
  %parameter.3 = f32[5]{0} parameter(2)
  %parameter.1 = f32[5,5]{1,0} parameter(0)
  %parameter.2 = f32[5]{0} parameter(1)
  %fusion = f32[5]{0} fusion(f32[5]{0} %parameter.3,
  f32[5,5]{1,0} %parameter.1, f32[5]{0} %parameter.2),
   kind=kLoop, calls=%fused_computation,
   metadata={op_type="add" op_name="jit(fn1)/add"
    source_file="<ipython-input-3-8e7c515249a1>" source_line=3}
  ROOT %tuple.7 = (f32[5]{0}) tuple(f32[5]{0} %fusion)
}
```

Figure 8: JAX XLA Logs

```
HloModule a_inference_loss_nograd_15__XlaMustCompile_true_config_proto___
n_007_n_003CPU_020_001_n_007_n_003GPU_020_0012_005__0010J_0008_
001_202_001_000__executor_type____.15

%scalar_add_computation (scalar_lhs: f32[], scalar_rhs: f32[]) -> f32[] {
  %scalar_lhs = f32[] parameter(0)
  %scalar_rhs = f32[] parameter(1)
  ROOT %add = f32[] add(f32[] %scalar_lhs, f32[] %scalar_rhs)
}

%fused_computation (param_0.1: f32[5], param_1.2: f32[5,5],
                    param_2.2: f32[5]) -> f32[5] {
  %param_1.2 = f32[5,5]{1,0} parameter(1)
  %param_2.2 = f32[5]{0} parameter(2)
  %broadcast.1 = f32[5,5]{1,0} broadcast(f32[5]{0} %param_2.2),
   dimensions={1}
  %multiply.1 = f32[5,5]{1,0} multiply(f32[5,5]{1,0} %param_1.2,
                f32[5,5]{1,0}
                %broadcast.1)
  %constant_1 = f32[] constant(0)
  %reduce.1 = f32[5]{0} reduce(f32[5,5]{1,0} %multiply.1, f32[] %constant_1),
            dimensions={1}, to_apply=%scalar_add_computation,
             metadata={op_type="Squeeze" op_name="MatVec/Squeeze"}
  %param_0.1 = f32[5]{0} parameter(0)
  ROOT %add.1 = f32[5]{0} add(f32[5]{0} %reduce.1, f32[5]{0} %param_0.1),
                metadata={op_type="AddV2" op_name="add"}
}

ENTRY %a_inference_loss_nograd_15__XlaMustCompile_true_config_proto___
n_007_n_003CPU_020_001_n_007_n_003GPU_020_0012_005__0010J_0008_001_
202_001_000__executor_type____.15
(arg0.1: f32[5,5], arg1.2: f32[5], arg2.3: f32[5]) -> f32[5] {
  %arg2.3 = f32[5]{0} parameter(2), parameter_replication={false},
  metadata={op_name="XLA_Args"}
  %arg0.1 = f32[5,5]{1,0} parameter(0), parameter_replication={false},
   metadata={op_name="XLA_Args"}
  %arg1.2 = f32[5]{0} parameter(1), parameter_replication={false},
   metadata={op_name="XLA_Args"}
  ROOT %fusion = f32[5]{0} fusion(f32[5]{0} %arg2.3, f32[5,5]{1,0} %arg0.1,
                f32[5]{0} %arg1.2), kind=kLoop, calls=%fused_computation,
                metadata={op_type="AddV2" op_name="add"}
}
```

Figure 9: The XLA logs for the TensorFlow function

Figure 8 and Figure 9 both possess a single fused kernel performing the same operation. We observe barring language-specific XLA syntax, the computation graph generated in the XLA log files are the same. This implies that the computation path taken by JAX and TensorFlow in this case are equivalent. Now, in Figure 10, we extend the code in Figure 7 to also take the gradients with respect to W and b, which are the parameters of the network. We now modify the example as follows:

```
@jax.jit
def fn1(W, x, b):
    return jnp.dot(W, x) + b

def loss(W, x, b, targ):
    out = fn1(W, x, b)
    return jnp.sum((out - targ)**2)

g = jax.jit(jax.grad(loss))(W, x, b, targ)  # computes gradient w.r.t W, b

@tf.function(experimental_compile=True)
    def loss(W, x, b, targ):
    return (tf.linalg.matvec(W, x) + b - targ)**2

@tf.function(experimental_compile=True)
def grad_loss(W, x, b, targ):
    with tf.GradientTape() as tape:
      tape.watch((W, x))
      loss_value = loss(W, x, b, targ)
return tape.gradient(loss_value, (W, b))
```

Figure 10: Matrix-Vector Product with Gradients in JAX and TensorFlow

These two code blocks do precisely the same computation once again, however, in this case result in different XLA log files. See Figures 11 and 12.

```
%fused_computation (param_0.5: f32[5], param_1.10: f32[5],
                    param_2.6: f32[5], param_3.5: f32[5,5]) -> f32[5,5] {
  %constant_16 = f32[] constant(2), metadata={op_type="mul"
   op_name="jit(loss)/jit(jvp(loss))/mul" source_file="<ipython-input-73-db04383b747a>"
    source_line=8}
  %broadcast.6 = f32[5]{0} broadcast(f32[] %constant_16),
   dimensions={}, metadata={op_type="mul"
   op_name="jit(loss)/jit(jvp(loss))/mul" source_file="<ipython-input-73-db04383b747a>"
    source_line=8}
  %param_3.5 = f32[5,5]{1,0} parameter(3)
  %param_2.6 = f32[5]{0} parameter(2)
  %broadcast.10 = f32[5,5]{1,0} broadcast(f32[5]{0} %param_2.6), dimensions={1}
  %multiply.9 = f32[5,5]{1,0} multiply(f32[5,5]{1,0} %param_3.5,
  f32[5,5]{1,0} %broadcast.10)
  %constant_20 = f32[] constant(0)
  %reduce.4 = f32[5]{0} reduce(f32[5,5]{1,0} %multiply.9, f32[] %constant_20), dimensions={1},
   to_apply=%scalar_add_computation,
   metadata={op_type="dot_general"
   op_name="jit(loss)/jit(jvp(loss))/jit(jvp(fn1))/
   dot_general[ dimension_numbers=(((1,), (0,)), ((), ()))  \n precision=None ]"
    source_file="<ipython-input-73-db04383b747a>"  source_line=3}
  %param_1.10 = f32[5]{0} parameter(1)
  %add.3 = f32[5]{0} add(f32[5]{0} %reduce.4,
  f32[5]{0} %param_1.10), metadata={op_type="add"
  op_name="jit(loss)/jit(jvp(loss))/jit(jvp(fn1))/add"
  source_file="<ipython-input-73-db04383b747a>"  source_line=3}
  %param_0.5 = f32[5]{0} parameter(0)
  %subtract.1 = f32[5]{0} subtract(f32[5]{0} %add.3,
   f32[5]{0} %param_0.5), metadata={op_type="sub"
      op_name="jit(loss)/jit(jvp(loss))/sub"
    source_file="<ipython-input-73-db04383b747a>" source_line=8}
  %multiply.6 = f32[5]{0} multiply(f32[5]{0} %broadcast.6,
   f32[5]{0} %subtract.1), metadata={op_type="mul" op_name="jit(loss)/jit(jvp(loss))/mul"
   source_file="<ipython-input-73-db04383b747a>" source_line=8}
  %broadcast.5 = f32[5,5]{1,0} broadcast(f32[5]{0} %multiply.6),
  dimensions={0}
  ROOT %multiply.5 = f32[5,5]{1,0} multiply(f32[5,5]{1,0} %broadcast.5,
  f32[5,5]{1,0} %broadcast.10),  metadata={op_type="dot_general" op_name="jit(loss)/
  jit(transpose(jvp(loss)))/jit(transpose(jvp(fn1)))/dot_general[
       dimension_numbers=(((), ()), ((), ()))\n precision=None ]"
       source_file="<ipython-input-73-db04383b747a>" source_line=3}
}
```

Figure 11: JAX post-optimization Gradient XLA Kernel. Note that there is only one fused kernel titled `fused_computation`. For brevity, the entry point to the XLA computation graph and auxiliary information are omitted.

```
%fused_computation (param_0.5: f32[5], param_1.7: f32[5], param_2.5:
 f32[5], param_3.3: f32[5]) -> f32[5,5] {
  %constant_1 = f32[] constant(2), metadata={op_type="Mul"
  op_name="PartitionedCall_1/gradients/pow_grad/mul_1"}
  %broadcast.4 = f32[5]{0} broadcast(f32[] %constant_1), dimensions={}
  , metadata={op_type="Mul" op_name="PartitionedCall_1/gradients/
  pow_grad/mul_1"}
  %param_1.7 = f32[5]{0} parameter(1)
  %param_2.5 = f32[5]{0} parameter(2)
  %add.1 = f32[5]{0} add(f32[5]{0} %param_1.7, f32[5]{0} %param_2.5),
  metadata={op_type="AddV2" op_name="PartitionedCall/add"}
  %param_0.5 = f32[5]{0} parameter(0)
  %subtract.0 = f32[5]{0} subtract(f32[5]{0} %add.1, f32[5]{0}
  %param_0.5), metadata={op_type="Sub" op_name="PartitionedCall/sub_0"}
  %multiply.3 = f32[5]{0} multiply(f32[5]{0} %broadcast.4, f32[5]{0}
  %subtract.0), metadata={op_type="Mul" op_name="PartitionedCall_1/
  gradients/pow_grad/mul_1"}
  %broadcast.3 = f32[5,5]{1,0} broadcast(f32[5]{0} %multiply.3),
  dimensions={0}
  %param_3.3 = f32[5]{0} parameter(3)
  %broadcast.5 = f32[5,5]{1,0} broadcast(f32[5]{0} %param_3.3),
  dimensions={1}
  ROOT %multiply.2 = f32[5,5]{1,0} multiply(f32[5,5]{1,0} %broadcast.
  3, f32[5,5]{1,0} %broadcast.5), metadata={op_type="MatMul"
  op_name="PartitionedCall_1/gradients/MatVec/MatMul_grad/MatMul"}
}

%fused_computation.1 (param_0.4: f32[5,5], param_1.8: f32[5]) -> f32
[5] {
  %param_0.4 = f32[5,5]{1,0} parameter(0)
  %param_1.8 = f32[5]{0} parameter(1)
  %broadcast.6 = f32[5,5]{1,0} broadcast(f32[5]{0} %param_1.8),
  dimensions={1}
  %multiply.4 = f32[5,5]{1,0} multiply(f32[5,5]{1,0} %param_0.4, f32[5,
  5]{1,0} %broadcast.6)
  %constant_2 = f32[] constant(0)
  ROOT %reduce.1 = f32[5]{0} reduce(f32[5,5]{1,0} %multiply.4, f32[]
  %constant_2), dimensions={1}, to_apply=%scalar_add_computation,
  metadata={op_type="Squeeze" op_name="PartitionedCall/MatVec/Squeeze"}
}
```

Figure 12: TensorFlow post-optimization Gradient XLA Kernel. Note that there are two fused kernels titled `fused_computation` and `fused_computation.1`. For brevity, the entry point to the XLA computation graph and auxiliary information are omitted.

Here we notice a considerable difference. Observe that the XLA compiler for JAX creates one large fused kernel that runs, as opposed to TensorFlow's multiple kernels that run. In this case, JAX is faster than TensorFlow as a consequence of minor overhead involved with two kernel launches compared to one. However, one should not expect that multiple kernel launches always implies slower performance. In certain cases, with appropriate kernel scheduling, multiple launches can actually be faster, as we observe in our runtime experiments when Custom TFP is faster than JAX.

B Architectures

We describe the precise architectures of the deep networks used for the experiments. All the networks are sequential, that is, the output of any layer is fed into only the subsequent layer. All networks use the ReLU activation function between linear layers, so we exclude them from our definitions for brevity. Further documentation of these architectures appears in our Github repository at `https://github.com/TheSalon/fast-dpsgd`.

Layer Type	Input Features	Output Neurons
Fully Connected	104	50
Fully Connected	50	10

Table 5: **FFNN Architecture**

Layer Type	Input Filters/Features	Output Filters/Neurons	Filter Size	Stride	Padding
2D Convolution	1	16	8×8	2	3
2D MaxPool	16	16	2×2	1	0
2D Convolution	16	32	4×4	1	0
2D MaxPool	32	32	2×2	1	0
2D Convolution	32	32	4×4	1	0
Fully Connected	512	32			
Fully Connected	32	10			

Table 6: **MNIST CNN Architecture**

Layer Type	Input Filters	Output Filters	Filter Size	Stride	Padding
2D Convolution	3	32	3×3	1	1
2D Convolution	32	32	3×3	1	1
2D Average Pool	32	32	2×2	2	0
2D Convolution	32	64	3×3	1	1
2D Convolution	64	64	3×3	1	1
2D Average Pool	64	64	2×2	2	0
2D Convolution	64	128	3×3	1	1
2D Convolution	128	128	3×3	1	1
2D Average Pool	128	128	2×2	2	0
2D Convolution	128	256	3×3	1	1
2D Convolution	256	10	3×3	1	1
2D Global Average Pool	10	10			

Table 7: **CIFAR10 CNN Architecture**. The Global Average Pool reduces all the spatial dimensions so the resulting tensor only has 10 activations (corresponding to the 10 CIFAR10 classes).

Layer Type	Input Features	Output Neurons
Embedding	256	256×16
1D Average Pool	256×16	16
Fully Connected	16	2

Table 8: **Embedding Network Architecture**. The embedding layer has a vocabulary size of 10,004.

Layer Type	Input Features	Output Neurons
Embedding	256	256×100
LSTM	256×100	256×100
1D Average Pool	256×100	100
Fully Connected	100	2

Table 9: **LSTM Network Architecture** The embedding layer has a vocabulary size of 10,004. Notice for the LSTM, we return the entire sequence of outputs (not just the final activations). We use static unrolling for JAX with `JIT`, TFP, and Custom TFP. Otherwise, the dynamic LSTM implementation is used. In PyTorch and TensorFlow, the dynamic LSTM implementation uses the optimized cudNN kernel (which is not usable in the private case). In JAX, the dynamic LSTM uses the `lax.scan` primitive, which is slower than unrolling in a compiled context, but faster outside of a compiled context.