Fusing Differentially Private Machine Learning with Vectorization and JIT Compilation

Pranav Subramani[*] Gautam Kamath[†]

September 27, 2020

Abstract

A frequent criticism of differentially private machine learning is the significant running time overhead incurred by Differentially Private Stochastic Gradient Descent (DPSGD), which may be as large as two orders of magnitude. We empirically demonstrate that the recently introduced machine learning framework JAX can consistently and (in some cases) dramatically reduce these overheads, an important step towards practical private machine learning. These gains are facilitated by JAX's rich support for key primitives such as vectorization and just-in-time compilation. We observe that these running time improvements can be as large as a 40x speedup in comparison to the best alternative. In addition to outperforming all other solutions for this problem, JAX is also simpler, easier to extend, and applicable in settings with varying levels of compute.

1 Introduction

Over the last couple decades, machine learning has experienced tremendous growth. We have witnessed the proliferation of artificial neural networks, facilitated by advances in both techniques and methodology as well as computing hardware. As a result, neural networks have been applied to solve learning 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. `g@csail.mit.edu`. Supported by an NSERC Discovery grant (une subvention à la découverte par CRSNG), a Compute Canada RRG grant (une subvention RGR par Calcul Canada), 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 microbatch, 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 clipping and noising operations. Intuitively, the clipping operation bounds the influence of any individual gradient, and adding noise blurs its contribution to the overall result. While these changes 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 focus solely on mitigating the running time overhead of DPSGD,[1] the causes of which (as well as the solution) we describe in the sequel.

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

[1]This article concentrates on DPSGD, as it is the most commonly run private algorithm on large-scale datasets. However, we note that this 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.

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 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. In 2015, motivated by an application to optimization based on importance sampling [ZZ15], Goodfellow proposed a high level solution (i.e., utilizing constructs within the machine learning framework) for computing per-example ℓ_2-norms of the gradients for fully-connected networks [Goo15],[2] an approach that has recently been extended to convolutional neural networks by Rochette, Manoel, and Tramel [RMT19]. Other proposed solutions work by either obtaining a quantity like the Jacobian [DKH20] and calculate the gradient via a Jacobian-Vector product or by implementing a feature to parallelize over the batch dimension [AG19]. We note that several of these approaches are are restricted to specific types of architectures – [Goo15] is restricted to fully-connected layers, though [RMT19] extends this to convolutional layers. BackPACK [DKH20] currently only supports traditional feed-forward networks, 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. Very recently (less than a month prior to the posting of our paper), Facebook released Opacus, the release version of their differentially private machine learning library previously known as PyTorch-DP [Fac20]. One of their primary advertised points is speed and scalability, which is achieved via a combination of the techniques mentioned above (more discussion appears in Section 2). We take their emphasis on computation as strong evidence for the necessity of fast differentially private machine learning. Finally, in the context of DPSGD, the strategy of microbatching modifies the algorithm itself. Rather than clipping and averaging the individual gradients, this approach first averages the (unclipped) gradients within "microbatches" (small sets consisting of a few datapoints), and the results are then clipped and averaged as before.[3] Though this reduces the number of clipping operations (and thus the overall running time), microbatching also requires us to increase the variance of the added Gaussian noise to maintain the same privacy guarantee. In other words: microbatching saves time, but at the cost of an *additional* drop in accuracy, on top of whatever penalty we would already pay for privacy with DPSGD. A more thorough description of all these approaches is provided in Section 2.

As mentioned in the literature (and thoroughly explored later in this paper), all existing approaches, both low- and high-level, seem to incur moderate to severe running time overhead when compared to 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] 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. This raises the question of whether, given current technologies, differentially private machine learning is *unavoidably* slow.

[2]Note that these are the exact same objects we require for DPSGD.

[3]Observe that the standard algorithm corresponds to microbatch size of 1.

1.1 Results – Fast Private Machine Learning via JAX

Our message is simple: JAX enables fast differentially private machine learning. JAX is a recently developed machine learning framework by researchers at Google, which combines just-in-time compilation with auto differentiation for high-performance machine learning [FJL18, BFH+18]. As we demonstrate in this paper, the core functionalities of JAX, namely, vectorization and just-in-time compilation and optimization, make it perfectly suited for privately training neural networks with DPSGD. We thoroughly benchmark JAX against the aforementioned alternatives, and find that JAX consistently comes out on top. While all other methods pay a significant computational cost to guarantee privacy, we find that JAX drastically reduces the overhead in comparison to non-private training. Our in-depth experimental results explore precisely which settings JAX affords the largest gains.

We outline our primary contributions below:

1. We present a thorough comparison of several frameworks and libraries for differential privacy (including JAX) and show that JAX is better in every metric considered. We perform the comparison across devices ranging from personal to industry level GPUs.

2. We prototype several architectures in the context of differential privacy and achieve runtimes that are state of the art compared to existing libraries.

3. We also release the code to perform these experiments for the purposes of reproducibility and also to allow researchers and engineers to prototype and build models in JAX faster as a result of our work.

We summarize some of our experimental results in Table 1, which displays median running time per epoch for a variety of architectures and frameworks.

First, we observe dramatic improvements for LSTM and embedding networks, potentially significant enough to bring these models from impractical into the realm of feasibility. In particular, JAX is able to privately train these models 5x and 37x faster than the best alternate private solution. JAX works efficiently out of the box, whereas all other libraries and frameworks either leave key modules unimplemented due to complexity, scale poorly due to memory usage, or are forced to revert to the naïve solution of processing points sequentially. Examining the overhead introduced due to privacy: JAX incurs a multiplicative increase in running time by factors of 2.04x and 1.04x, which are significantly smaller than the previous best overheads of 21.8x and 34.4x. 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 other architectures, we observe that with respect to training time under differential privacy, JAX often performs much better than the best alternative (and never worse), which can enable more rapid prototyping of private models. Again, this is out of the box, whereas several of the other frameworks introduce significant complexity to handle private training. Sometimes the improvement can still be significant: focusing on a feedforward neural network, JAX is 1.8x faster than the best private alternative, and incurs only a 10% overhead for privacy, compared to the 47% overhead of the best alternative. Recall that these are per-epoch times: while an improvement of 0.5 seconds might seem insignificant, this can add up when training for many epochs.

We comment that, while there has been significant work towards making DPSGD run faster (namely, all the approaches we compare with), they are without exception built *on top* of frameworks such as PyTorch and TensorFlow. By suggesting a switch to JAX, 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. Fast DPSGD in JAX is enabled by

	Private Training		Non-Private Training	
Architecture	JAX	Best Alternative	JAX	Best Alternative
Logistic Regression	**0.58**	0.91	0.54	0.76
Feedforward NN	**0.64**	1.17	0.58	0.79
Convolutional NN	**4.06**	4.13	0.53	2.71
Embedding Network	**1.80**	67.05	1.73	1.95
LSTM	**39.59**	189.30	19.43	8.69

Table 1: Median running time (s) per epoch of training various models, both with and without differential privacy, in JAX and other frameworks. JAX is consistently the fastest, and in many cases such as LSTMs or embedding networks, dramatically so.

the confluence of two core features of the language: a vectorized map function (which allows one to trivially parallelize per-gradient operations) combined with just-in-time (JIT) compilation and optimization via XLA (Accelerated Linear Algebra, a domain-specific compiler). In PyTorch, these features are in early development and relatively immature, respectively, and even once developed, one would need to rewrite the many libraries which depend on them to exploit these new features. TensorFlow is in a more promising state, as it has support for both these features, though the latter is not currently supported by TensorFlow Privacy. The main barrier appears to be the transition of TensorFlow Privacy to version 2.0 of TensorFlow, which is currently in progress. Once this is complete, we conjecture that TensorFlow will enjoy similar performance improvements as enjoyed by JAX. In short, to simultaneously enable the critical features of vectorized mapping and JIT compilation in the context of differential privacy seems to be a significant engineering effort in either PyTorch or TensorFlow, while these are supported as core language features in JAX.

Given the simplicity and effectiveness of our proposed solution, combined with the importance of the problem, one might suspect that the strong utility of JAX is already well known to the community at large. To the best of our knowledge, this is not the case. There indeed seem to be a small number of differential privacy experts who are aware of the benefits it confers [Tal20], and the official JAX GitHub repository contains a toy demonstration [Cre19]. That said, we are unaware of any research papers or even blog posts which describe the suitability of JAX for differentially private machine learning. At the time of posting this paper, we found 16 results on Google Scholar for "`JAX "differential privacy"`," only two of which actually use JAX for differential privacy [WC20, PTS+20], and the latter of which was released while we were preparing this paper. Neither of these works emphasizes or even comments on the computational advantages of JAX. Thus we must conclude that, at best, the advantages are only known to a very limited group of insiders. We hope that our investigation will help document and democratize this information, and encourage others to try JAX for their private machine learning needs.

We note that there are other settings in machine learning where per-example gradients are useful, including applications to optimization based on importance sampling [ZZ15]. We anticipate that JAX will enable similar speedups in these settings as well, and encourage researchers and practitioners in these areas to experiment with JAX; however, thorough investigations of each application are outside the scope of our work.

2 Description of Approaches

JAX [FJL18, BFH$^+$18]. JAX is a recently introduced framework for machine learning, defined by its automatic differentiation library and JIT compilation via XLA. A program written in pure Python and NumPy can be translated to an intermediate language (XLA-HLO) and optimized by way of kernel-fusions, reusing buffers and more to produce a much more efficient compiled version of the same program. Additionally, one of the core functions present in JAX is `jax.vmap` which allows for batch level parallelism that is fundamental to DPSGD. As we will demonstrate, these features enable the fastest approach for DPSGD that we are currently aware of.

Chain-Rule-Based Per-Example Gradients [Goo15, RMT19]. The first suite of techniques are implemented on top of PyTorch. They support per-example gradients for fully-connected layers by noting that their computation reduces to an outer product [Goo15] which is efficiently computable on a GPU, and further utilize methods and attributes within PyTorch's convolutional layers to allow for efficient extraction of per-example gradients [RMT19] which we describe in the detail in the following paragraphs. We refer to this as CRB in 4

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 which we refer to as 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. PyTorch's automatic differentiation cannot be parallelized across the batch dimension (b in the above equation) [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 $(1, B, C, T)$. First note that the initial convolution is a 1-dimensional convolution, while in the expression above we are dealing with an added dimension. Similarly, in order to allow for backpropagation through a k-dimensional convolutional layer, this method requires a $(k+1)$-dimensional convolutional layer. 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]. Consider the following expression for the gradient of a loss function, based on the chain rule:

$$\nabla_{\theta^{(i)}} \ell(\theta) = (J_{\theta^{(i)}} z_n^{(i)})^T (J_{z_n^{(i)}} z_n^{(i+1)})^T \cdots (J_{z_n^{(L-1)}} z_n^{(L)})^T (\nabla_{z_n^{(L)}} \ell_n(\theta))$$

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 [DKH20], the authors 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 $(N, d^{(i)})$. It is also important to note that BackPACK provides efficient computation for the transpose of the Jacobian as well as the Jacobian. The paper goes on to compare the runtime of BackPACK against the cost of individually extracting gradients in a for-loop and shows that BackPACK is significantly faster. The authors mention that the cost of extracting the individual gradients does come at a minor overhead when compared to regular training.[4]

Opacus [Fac20]. Opacus is one of the latest frameworks that have been released which supports per-example gradients in PyTorch. Opacus utilizes PyTorch's forward and backward hooks in order to propagate gradients. They provide support for several PyTorch layers including LSTM layers, which are not supported in either of the prior 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. We use Opacus as a benchmark because of the optimizations that have been made on it will offer a fair comparison against JAX.

PyVacy. Prior to the release of Opacus (and its predecessor PyTorch-DP), PyVacy was the most popular library for differentially private machine learning built on top of PyTorch. PyVacy has no custom support for parallelizing across the batch dimension for any layer since it processes each sample individually (by way of a for-loop). This leads to a large increase in runtime for models trained using PyVacy, however, since it is one of the better known frameworks in PyTorch, we compare against it as well.

TensorFlow Privacy. TensorFlow Privacy is a library for differentially private machine learning, built on top of TensorFlow. TensorFlow Privacy has general support for vectorized implementation of DPSGD. In particular, it has a function `vectorized_map` which allows it to parallelize across the batch dimension which is necessary to extract per-example gradients. It should be noted that according to the documentation of the function, 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.

3 Empirical Findings

In this section we present our empirical results, demonstrating the computational advantages enjoyed by JAX in the context of DPSGD. The versions of BackPACK, Opacus, and CRB are 1.1.1, 0.9.0 and the release version. We perform a thorough empirical evaluation of JAX against the other frameworks mentioned in Section 2. The section is roughly divided into an explanation of the experiment and some inferences we can draw from what we observe. The primary metrics of interest are median runtime per epoch and total memory utilization. We also measure the difference between the private and non-private running time to quantify the overhead of privacy. We

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

comment that accuracy and privacy gurantees are outside the scope of our investigation, as any faithful implementation of the base algorithm should have identical guarantees.[5]

For each experiment below, we train the model for 60 epochs, reporting the median over these epochs. Additionally, we attempt to train each model with batch sizes ranging from 20 to 120, though we observe that certain batch sizes are infeasible in certain frameworks due to memory overheads. For the logistic regression, feedforward neural network and the conv-net architectures, we do not test against PyVacy because it skews the y-axis and the distinction between the other models are harder to observe. We run our experiments in two settings:

- A consumer setup. The GPU is an Nvidia GTX 1050 Ti Max-Q with 4GB of VRAM. The CPU used for the experiments is an Intel i7-8850H.
- A high-performance computing setup. Experiments were run on the Graham cluster supported by Compute Canada. The GPU is an Nvidia P100 with 11GB of VRAM. The CPU used for the experiments is an Intel E5-2683 v4 Broadwell.

The results below are with the former consumer setup and the high-performance setup is in the appendix. Note that the high-performance setup has similar results to the consumer setup, except that the runtimes are lower.

3.1 Logistic Regression

We first examine performance for a logistic regression task. Specifically, we use the Adult dataset [DG17] following some processing as done in [WT18]. After processing, the dataset has 45220 datapoints with 104 features, and the resulting model has 105 parameters.

[5]One strategy, microbatching, involves a tradeoff between running time and accuracy by averaging "microbatches" into single examples, thus reducing the number of clipping operations. However, since this generally leads to significant drops in accuracy for modest running time improvements, we do not benchmark this strategy at this time.

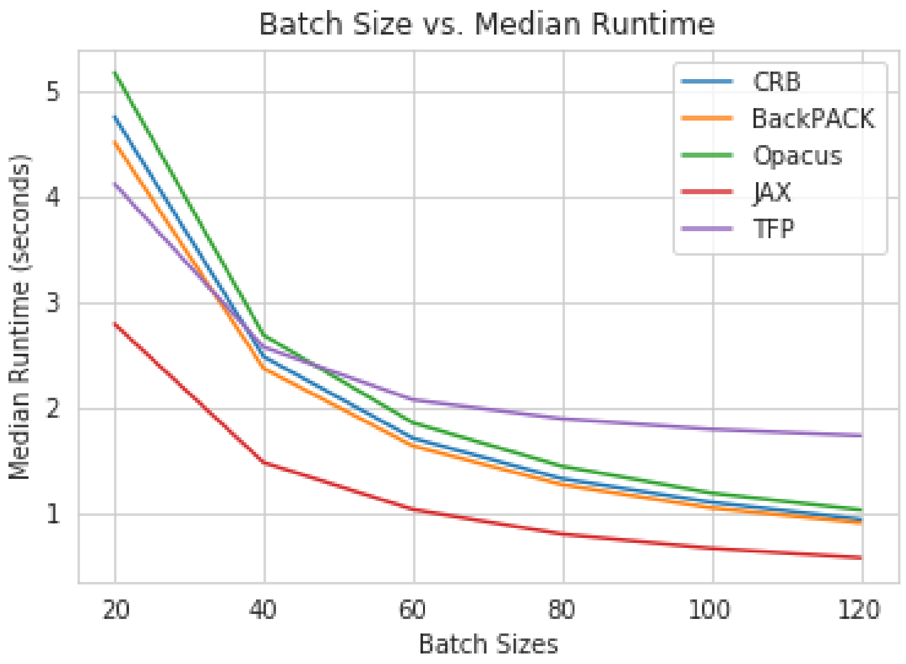

Figure 1: Median per-epoch running time of DPSGD for (private) logistic regression in various frameworks. JAX is fastest for all batch sizes.

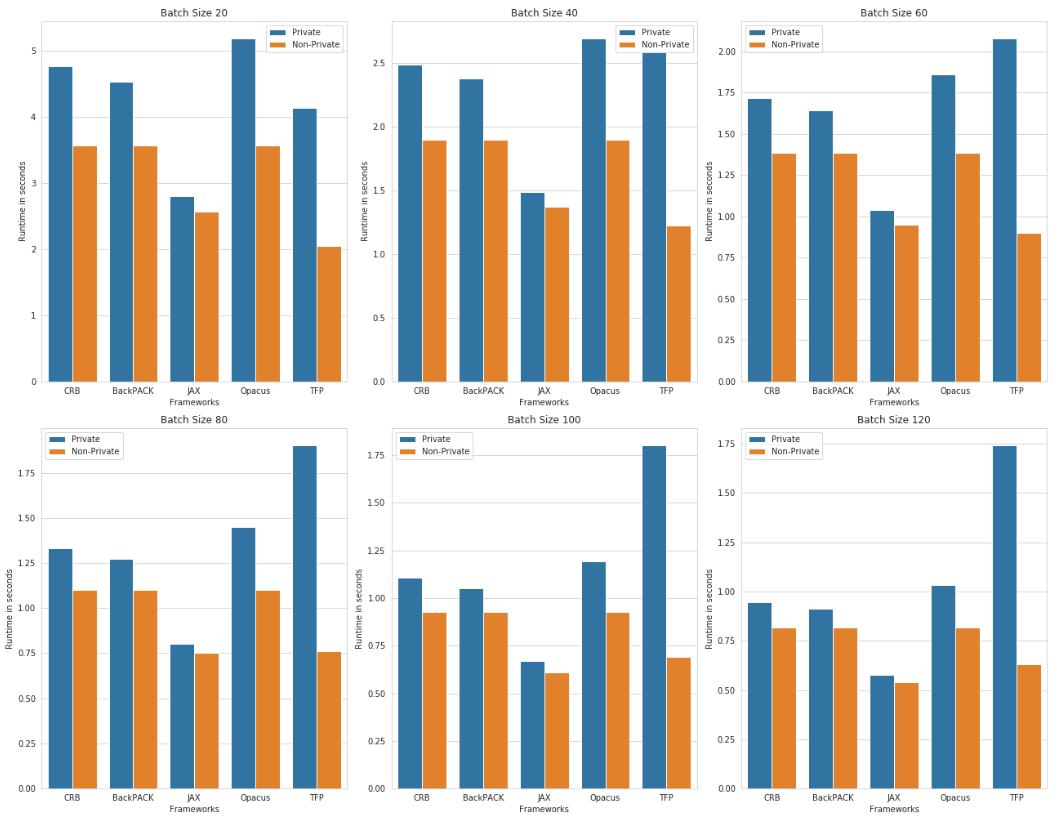

Figure 2: Median per-epoch running time of DPSGD and SGD for logistic regression in various frameworks. JAX incurs the smallest overhead for privacy, and is the fastest private solution.

Figure 1 shows that JAX is faster at this task than all alternative frameworks. This is especially pronounced at lower batch sizes which can largely be attributed to the JIT compiler that JAX employs, we elaborate in Section 4. Next, we compare the runtimes of private-SGD (DPSGD) against non-private-SGD (Vanilla SGD). Figure 2 compares the private and non-private runtimes across the different batch sizes. In all cases, there is some overhead for privacy, but this overhead is smallest for JAX. As observed before, JAX is the fastest for all batch sizes.

3.2 Feedforward Neural Network

Next, we benchmark performance for a feedforward neural network architecture, with two fully connected layers separated by a ReLU layer. The main difference between this model and the previous one is the size of the model and the non-linearity employed. We use the same Adult dataset (which has 45220 datapoints and 104 features) and the model has 5352 parameters.

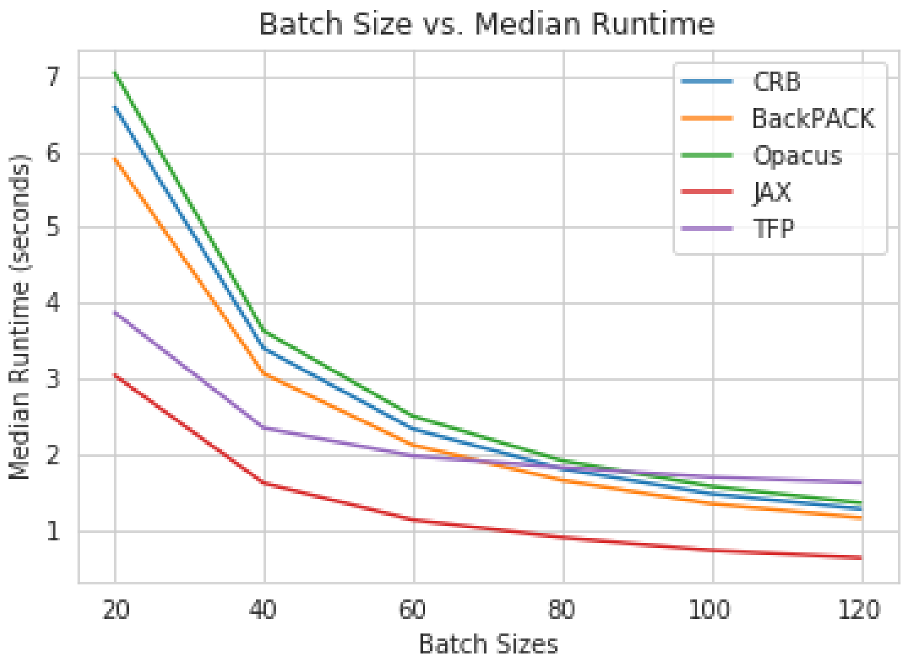

Figure 3: Median per-epoch running time of DPSGD on a feedforward network in various frameworks. JAX is fastest for all batch sizes.

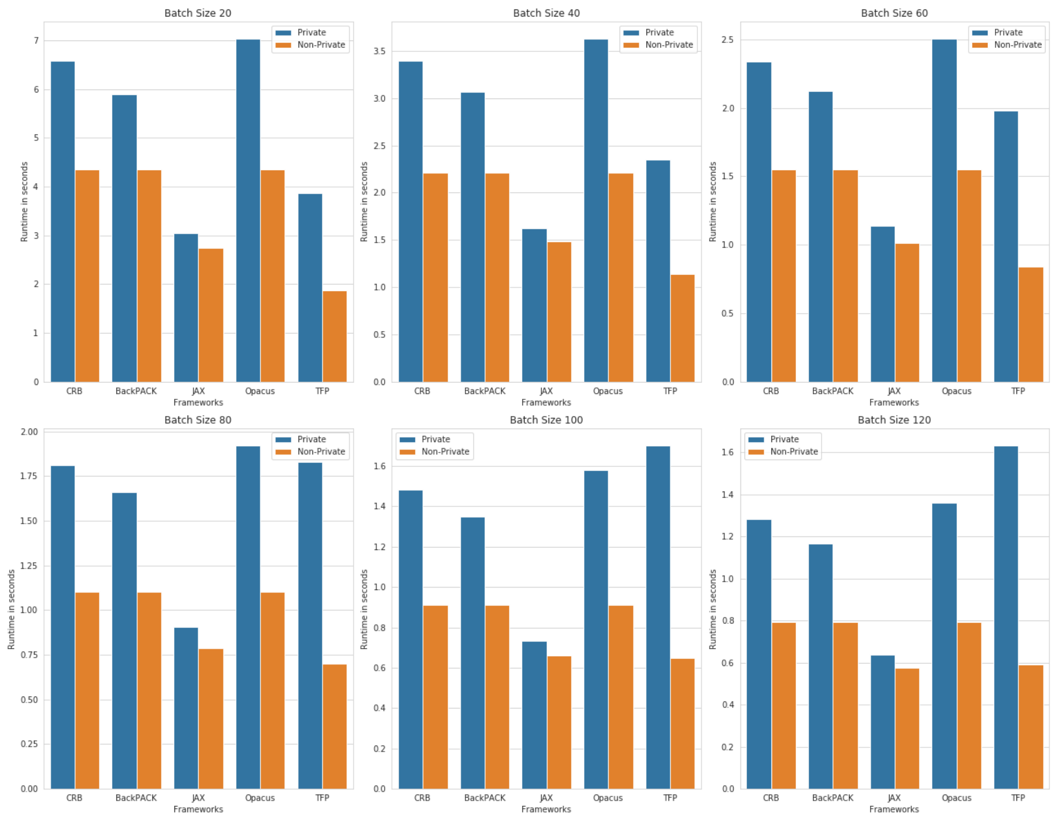

Figure 4: Median per-epoch running time of DPSGD and SGD on a feedforward network in various frameworks. JAX incurs the smallest overhead for privacy, and is the fastest private solution.

Figures 3 and 4 tell a very similar story as our logistic regression experiments. JAX is consistently the fastest, and has the least overhead for privacy. Note that the advantages are even more pronounced here than for logistic regression, despite the similar structure.

3.3 Convolutional Neural Network

Our next model is a convolutional neural network. Our architecture is as follows: a convolutional layer, a ReLU activation, a max-pooling layer, a repetition of the previous three layers, a flattening of the output tensor, and then two linear transformations with a ReLU between them. The loss function used here is the cross-entropy loss and the optimizer is DPSGD (or SGD, in the non-private experiments). The model has 26010 parameters, and the dataset is of size 60000 and dimensionality 784. The reader might be concerned about the relatively limited size of this model, in comparison to the scale of modern neural network architectures in non-private settings. However, under the constraint of differential privacy, models of this size might be more common. Indeed, DPSGD adds noise which depends on the square root of the number of parameters, and thus modestly-sized models frequently strike the appropriate balance between expressivity and magnitude of noise required [PTS+20].

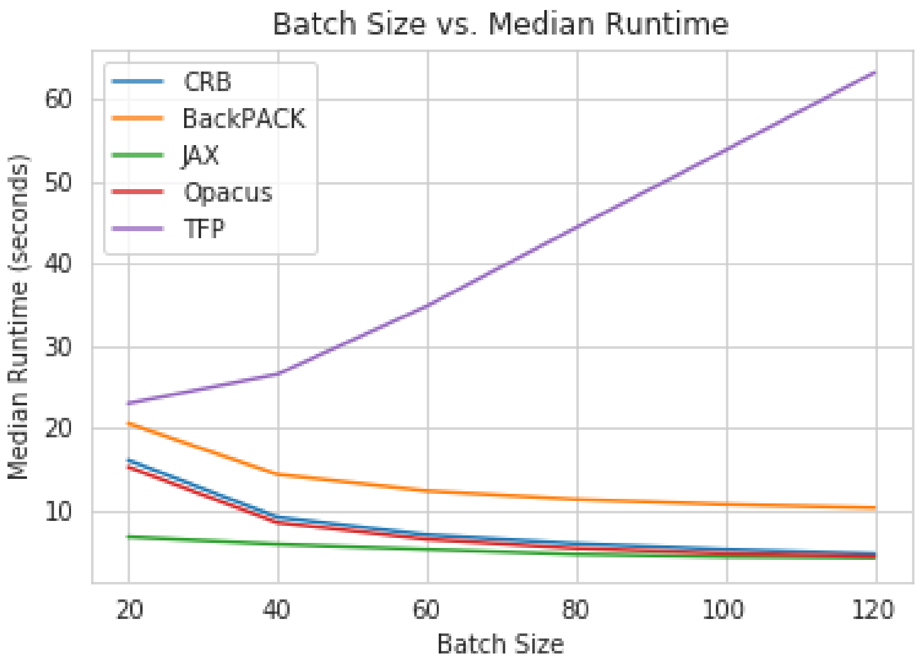

Figure 5: The runtime of the different frameworks in the feed forward network model on the MNIST dataset across a variety of batch sizes. We observe that JAX is the fastest across all frameworks. The speedup, however, is the least across all experiments in Section 3

From this figure, we observe that the runtime of JAX across all the batch sizes is significantly lower than the runtimes of the other methods. In figure 6 we observe a comparison of running times across the private and non-private optimizers in the convolutional neural network architecture. Here too we observe that JAX is faster than all competing frameworks. However, the difference is that at the final batch size (120), we do notice that the runtimes are roughly equivalent to one another, at around 5s. The runtime of BackPACK also appears to be slower by a significant margin compared to the rest of the methods evaluated in this experiment.

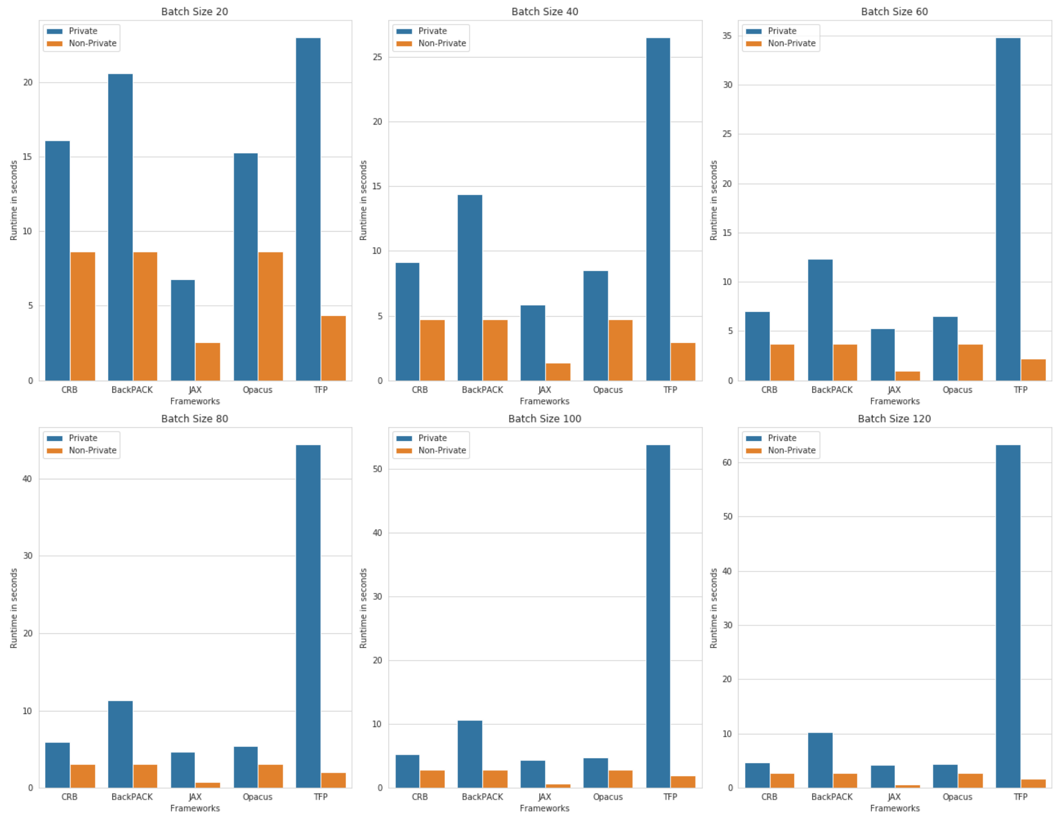

Figure 6: The comparative runtimes of the convolutional neural network across different batch sizes

Again, we observe that the private running times of JAX is lower than the rest of the models. For larger batch sizes we also notice that the difference in running times between the private and non-private optimizers increases.

3.4 Embedding Network

Differentially private machine learning has also focused on sentiment analysis for movie reviews, evidenced by public code available for this experiment on Opacus and TensorFlow Privacy. The dataset being used for this task is the IMDb dataset [MDP+11] which contains 25000 training examples of movie reviews and associated sentiments, which can be positive or negative. The network we use for this is the same one that is used in the tutorials for Opacus and TensorFlow Privacy. The total number of parameters for this network is 160370 which is substantially larger than the previous models. The network itself consists of an embedding layer, followed by an average pooling layer over the dimension representing the sequence length, followed by two fully connected layers with a non-linearity between them. Below is the runtime between JAX's implementation of the model against Opacus. Here, we do not benchmark CRB and BackPACK since they do not support the Embedding module in PyTorch as of writing this.

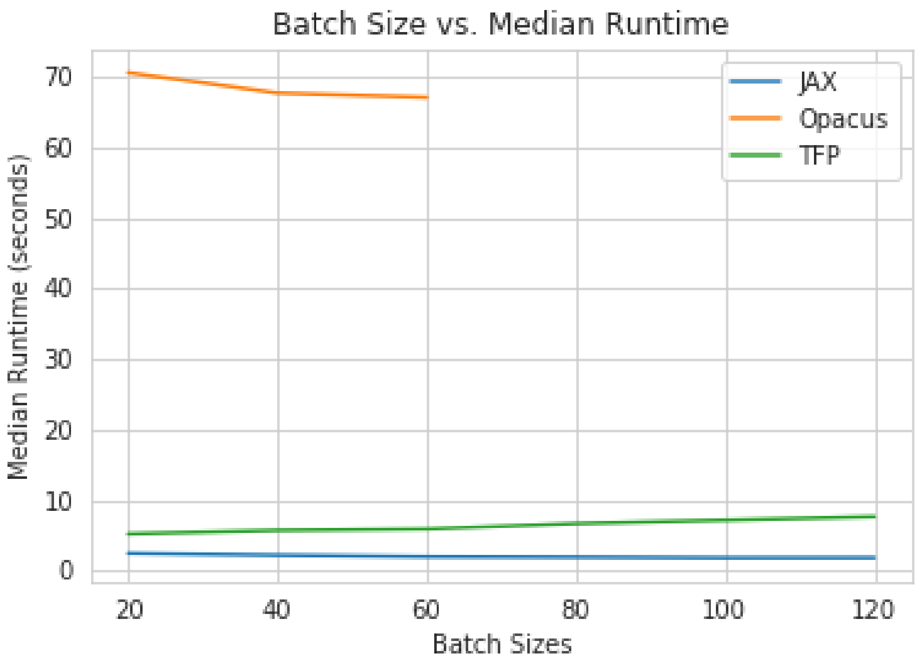

Figure 7: The runtime of JAX and Opacus on the Embedding Network on the IMDb dataset across a variety of batch sizes

As we discuss in the Section 4 Opacus' peak memory usage scales quadratically in the batch size, in comparison to the linear scaling of JAX. As a result, Opacus was unable to run on batches larger than 60. Unfortunately, in the hardware that was present, Opacus was unable to run at a batch size larger than 60. We elaborate on this in the interpretation section where we discuss how different frameworks reduce their memory footprint. Note that JAX is significantly faster than Opacus for this task.

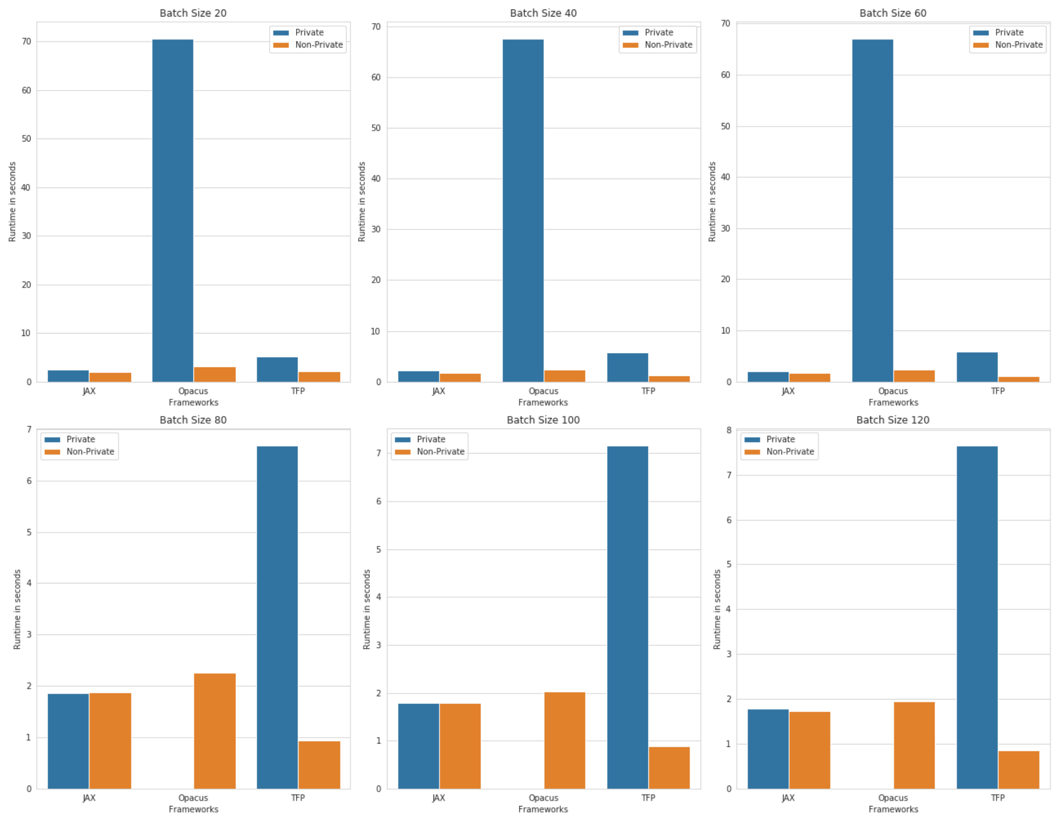

Figure 8: The runtime of JAX and Opacus on the Embedding Network on the IMDb dataset across a variety of batch sizes. JAX is faster than Opacus by a significant margin for this task.

As can be seen in figure 8, the JAX implementation has a very small decrease in runtime between the private and non-private versions and Opacus' implementation is roughly as fast as the non-private versions in PyTorch. For the bottom three histograms, there is no comparison for private-Opacus due to lack of memory.

3.5 Recurrent Neural Network (with LSTM)

The final experiment incorporate a Long-Short-Term-Memory [HS97] (LSTM) layer. The dataset used for this task is also the IMDb dataset, which has 25000 training examples, and the preprocessing is exactly the same as in the the embedding network. The sequence length in the previous network was fixed at 256, and the LSTM is over this dimension. The rest of the network is identical to the previous experiment. The total number of parameters in this model is 108100, making it our largest experiment. To utilize the LSTM with JAX, we also use a function which iterates over the sequence length dimension called `static_unroll` that is present in Haiku [HCNB20], is a framework for machine learning in JAX. We discuss more about this choice in section 4. Below we have the runtime plots comparing PyVacy and JAX. The codebase using the CRB method and BackPACK do not accommodate LSTM units and while Opacus does accommodate this with a custom implementation of the LSTM layer, called `DPLSTM`, it runs out of memory quickly.

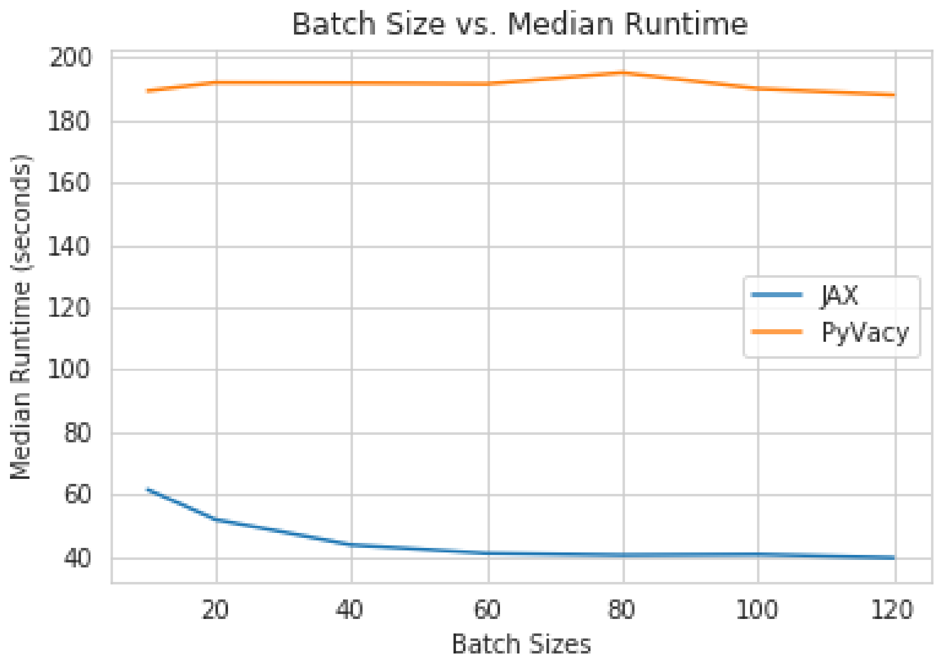

Figure 9: The runtime of JAX and Opacus on the LSTM network on the IMDb dataset across a variety of batch sizes

Due to Opacus' sizeable memory requirements for the backward pass, it could only be run for a batch size of up to 10. We fix this batch size and compare the different For this batch size, we compare the results between the different approaches. To enable Opacus to run with the largest possible batch size, we actually advantaged it in this experiment. Specifically, JAX and PyVacy were run on an Nvidia 1050TI (the consumer setup) while the Opacus results were on a P100 GPU (the high performance setup). Even so, the difference in runtimes is quite significant.

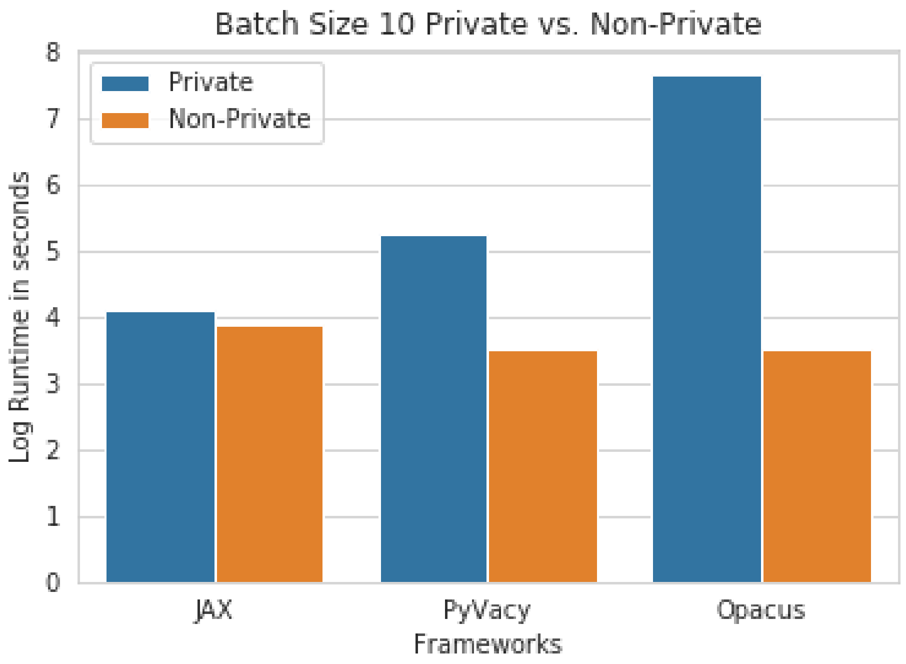

Figure 10: The runtime of JAX, Opacus and PyVacy compared on a batch size of 10 for the LSTM network on the IMDb dataset

Notice that the y-axis is on the log scale, so the true wall clock time difference between these models is more significant. Unfortunately, to run it for a batch size of 20 (which is the next batch size that we compare), we required significantly more memory than we had access to. One might ask whether, given sufficient memory, Opacus could outperform JAX. The answer is no, and we elaborate with a synthetic experiment in section 4. Thus, we only compare PyVacy and JAX in figure 10.

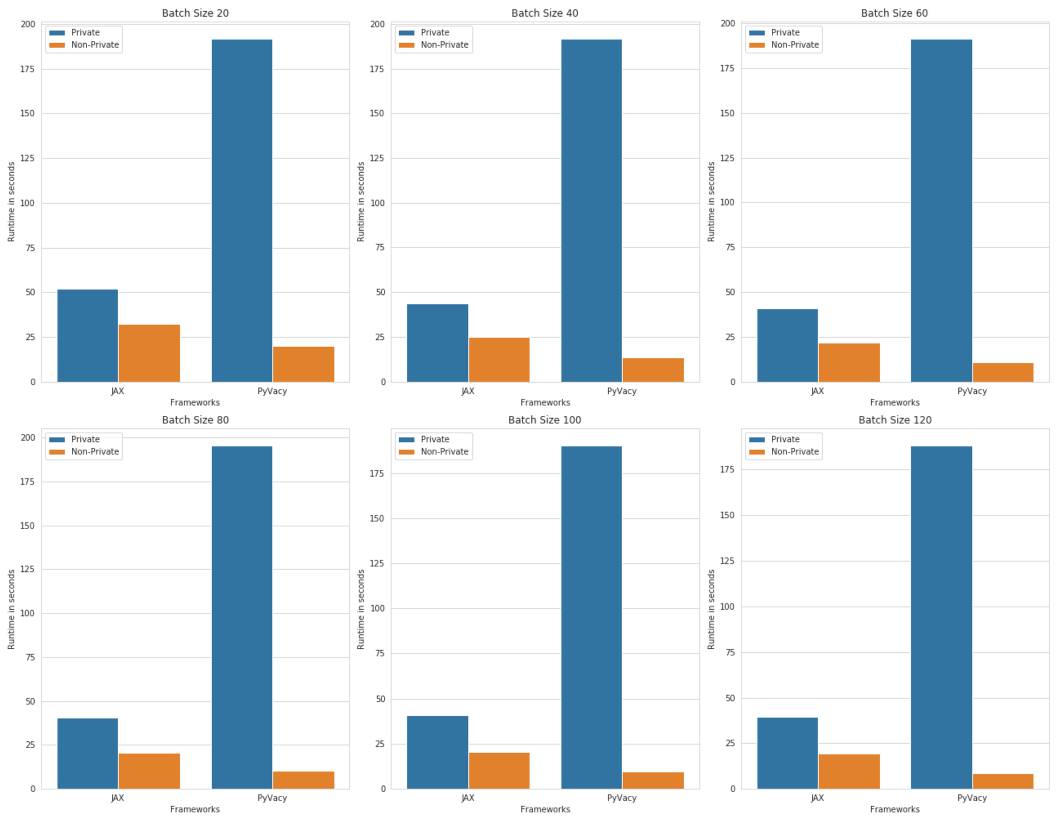

Figure 11: The runtime of JAX and PyVacy on the LSTM network on the IMDb dataset contrasted with the non-private running times

An interesting observation is that the runtimes of non-private PyVacy (i.e., a vanilla PyTorch network) with the SGD optimizer is even faster than the non-private JAX benchmark. This is behaviour that was not observed in our previous experiments, and explanation is provided in section 4. That said, JAX's private running times are significantly faster than PyVacy's private running times.

4 Interpretation and Discussion

In this section we go into detail regarding the reasons for the runtime and memory discrepancies we see in the previous section. We divide our explanations into 2 subsections. First, we provide an explanation for the superior running time seen by JAX and TensorFlow Privacy. Next, we comment about the possible reasons for the variation in memory consumed by different frameworks. There are auxiliary benefits and other observations that one might make after taking a look at the plots and we cover them in the Appendix (A)

4.1 Running Time

After observing the running times between the different frameworks, we notice that JAX are significantly faster across all batch sizes in comparison to the other frameworks (see Figures 1, 3, 5, 7, 9). Here we give an explanation for why this is the case. The primary improvements to running time are as a result of XLA optimization, JIT compilation and the function `vmap` (or `vectorized_map` in

TensorFlow). Combining these tools allows for a reduction in Python-related overhead (by way of JIT), reduction in GPU runtime through kernel-fusion (by way of XLA) and fast batch-level parallelism (by way of `vmap`). In JAX, when the decorator `@JIT` is placed on a function, the function will be JIT compiled and optimized with XLA primitives.[6] Note that this does place a restriction on the permissible operations allowed within the function since using JIT has minimum requirements such as the shapes of tensors need to be decipherable during compilation. Having just one of these components does indeed reduce the running time, however, combining all of them provides the maximum improvement in running time across the frameworks in this study. It is important to note that depending on the architecture, the improvements in running time might differ from the addition of JIT or `vmap`. To illustrate this, we have an experiment below where we have the following setup:

1. JIT + `vmap`: Utilize both JIT + `vmap` and observe the running time of DPSGD across a neural network
2. JIT: Utilize only JIT compilation without `vmap`. This is equivalent to utilizing JAX with a minibatch size of 1.
3. `vmap`: Utilize only `vmap` without the JIT compilation and XLA optimization
4. Neither: Apply neither JIT nor `vmap`.

The goal of the aforementioned experiment is to provide an understanding of how much gain can be siphoned from the addition of one of these features. An auxiliary benefit of providing JIT compilation is the decreased runtime at lower batch sizes. At smaller batch sizes, the overhead incurred by Python dominates the overall runtime and as a result, frameworks that do not JIT compile tend to be slower. This would imply that frameworks like JAX, incur a smaller cost if the dataset is very high dimensional and only small batches can fit in memory. Conversely, one of the downsides to JIT compilation is that there is a slowdown in the first epoch as a result of the compilation procedure (sometimes called the "warm-up time"). If the model has a large number of parameters then the compilation time increases. Despite this, the wall clock time of frameworks employing JIT is still significantly lower than frameworks that do not.

PyTorch does support JIT compilation as well, however, compilation of the PyTorch Module into a ScriptModule failed which is likely due to the incompatibility between certain operations used in the codebases for Opacus/CRB/BackPACK and JIT. Assuming that these codebases were rewritten to support JIT compilation [7], it is unlikely that the performance will be as good as JAX. This can be attributed to the availability of XLA optimization under the hood and the `vmap` functionality present in JAX and not in PyTorch [8]. One would expect that if all of these features were ported to the PyTorch ecosystem at the same maturity that they are present in the JAX/TensorFlow ecosystem then the runtimes would be indistinguishable.

4.2 Memory Consumption

The next metric we focus on is the memory consumption. JAX and TensorFlow Privacy utilize less memory than the other frameworks built using PyTorch. The primary reason for this is due to the XLA optimization which has been noted to reduce memory usage. It achieves this through

[6]While XLA and JIT are fundamentally different concepts, whenever we specify JIT with respect to JAX, we mean JIT and XLA in unison.

[7]We implicitly assume that the JIT compilation between PyTorch and TensorFlow/Jax are equivalent here

[8]There is however, an RFC in PyTorch for `vmap` https://github.com/pytorch/pytorch/issues/42368

two approaches: (1) Analyzing and scheduling memory usage and (2) reutilizing buffers where possible. As a result of these techniques, XLA-enhanced frameworks tend to work at larger batch sizes. As the batch size increases, so does the amount of memory required as well, especially in the case of private-SGD since a gradient needs to be stored in memory for each sample in a batch. Consequently, larger models can be prototyped using XLA on a device with lesser memory as a result of the memory scheduling that XLA performs. We also notice in the LSTM and Embedding experiment that the memory utilized by Opacus far outweighs the memory utilized by JAX. This is evidenced by the batch size that is accommodable by Opacus compared to the batch size that is supported by the XLA-optimized frameworks. Below we conduct an experiment with JAX to showcase the maximum batch size as a result of using XLA-optimization against not using XLA-optimization.

PyTorch does have a version of XLA that is public at this moment. However, it was released more recently than the TensorFlow XLA version (which is used by JAX as well) and thus lacks the maturity and features that are present in the latter. If these features were appended to PyTorch's XLA, then the running times of private-SGD between the frameworks would be the same (assuming that `vmap` is present as well).

4.3 Discussion

In this section we will go into some detail about the potential downsides of adopting JAX as the framework for differentially private machine learning. The first argument we present is that JAX's documentation is lacking in comparison to popular frameworks like PyTorch, TensorFlow and Keras. While we do not claim that there is no documentation, given the lifetime of the popular frameworks, it is understandable that there are many more examples, use-cases documented both in and outside of the primary websites for the frameworks. As a result, it may be more intimidating for new users to dive into JAX given that they may be breaking new ground. The maturity in online forums also might be a hinderence to the development of new projects. Since JAX is a newer framework, there are fewer roadblocks that people have encountered compared to TensorFlow and PyTorch, and as a result, there are well documented solutions to said roadblocks.

The ecosystem developed around TensorFlow and PyTorch also are much larger than that of JAX. For example, TensorFlow and PyTorch have well established libraries and frameworks built on top of them that are used in both research and the industry. An example of this is, PyTorch has support for deployment on mobile devices [9] while the same cannot be said for JAX. Mobile support is just one of the few features present, there are a variety of tools and an entire ecosystem of libraries built in these frameworks that allow the user to perform their desired task with a toolset specific to their problem. Additionally, given that PyTorch and TensorFlow gave tools to developers to port their models to production, several companies adopted these frameworks and libraries as part of their production pipeline. Switching from one framework to another in a large company might be significantly more difficult than switching frameworks for an independent researcher.

Another potential downside to JAX is that using XLA primitives to optimize a deep learning model has its pros and cons. On one hand, XLA primitives present the unique advantage of generalizing to a variety of use-cases and are domain agnostic but on the other hand they might be suboptimal compared to specific hand crafted kernels to perform a particular task. An example of this was seen in the paper with regards to LSTMs where the runtime of PyTorch was signficantly less that the runtime of JAX purely as a function of the implementation of the fused cuDNN-RNN kernel. While we note that XLA's optimization will continually improve in the future, it is possible

[9] https://pytorch.org/mobile/home/

that with handcrafted kernels for all possible differentially private optimization algorithms, we may see that PyTorch and TensorFlow outperform JAX.

Finally, it should be mentioned that JAX deviates from the usual style of programming that TensorFlow 2.0 and PyTorch have adopted, on that thought, it may require more time for a user to be on-boarded into the programming model that JAX employs. Additionally, JAX adopts a more functional programming approach which might require some overhead in terms of learning which is potentially intimidating to a newcomer to high performance machine learning. Furthermore, given that JAX's bread and butter is the JIT compiler and XLA-optimization, there are certain restrictions to what can and cannot be done within the JAX framework. In fact, JAX documents some of the 'gotchas' [LJ19] which are pitfalls that programmers can run into while using JAX. Some of these include working with loops, conditionals and random number generators. This might also imply that building a machine learning model that utilizes some of these features would be more time consuming to build in JAX compared to alternative frameworks.

5 Conclusion and Future Work

Some work that was not a part of this paper that we would hope to work on in the future is the use of parallelization across multiple GPUs with respect to JAX. JAX provides a function called `pmap` which enables GPU and CPU level parallelization which can be extremely useful for applications involving federated learning [KMA+19, MMR+17] and reinforcement learning [SA18, ESM+18, HQB+18, LLN+18, KOQ+18]. A comparison between PyTorch and TensorFlow's mechanisms for distributed training and JAX's methods is left open for future work. We hope to have thoroughly demonstrated a variety of use cases for differentially private machine learning models and how JAX is a suitable framework in these domains with regards to the metrics that we discussed. We hope that this encourages people in the differentially private machine learning community to utilize JAX for their experiments and that this provides the incentive for non-experts to experiment with differential private machine learning since it can be run on commodity GPUs with the help of JAX.

References

[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.

[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. http://github.com/google/jax, 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.

[Cre19] Elliot Creager. differentially_private_sgd.py. https://github.com/google/jax/blob/master/examples/differentially_private_sgd.py, April 2019.

[DG17] Dheeru Dua and Casey Graff. UCI machine learning repository, 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.

[ESM$^+$18] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. *arXiv preprint arXiv:1802.01561*, 2018.

[Fac20] Facebook. Opacus. https://opacus.ai/, August 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.

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

[HCNB20] Tom Hennigan, Trevor Cai, Tamara Norman, and Igor Babuschkin. Haiku: Sonnet for JAX, 2020.

[HQB$^+$18] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado Van Hasselt, and David Silver. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*, 2018.

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

[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.

[KMA$^+$19] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Keith Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. Advances and open problems in federated learning. *arXiv preprint arXiv:1912.04977*, 2019.

[KOQ$^+$18] Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. Recurrent experience replay in distributed reinforcement learning. In *International conference on learning representations*, 2018.

[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.

[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. https://jax.readthedocs.io/en/latest/notebooks/Common_Gotchas_in_JAX.html, September 2019.

[LLN$^+$18] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning*, pages 3053–3062, 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.

[MMR$^+$17] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial Intelligence and Statistics*, pages 1273–1282. PMLR, 2017.

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

[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.

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

[SA18] Adam Stooke and Pieter Abbeel. Accelerated methods for deep reinforcement learning. *arXiv preprint arXiv:1803.02811*, 2018.

[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.

[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.

[WT18] Lun Wang and Om Thakkar. https://github.com/sunblaze-ucb/dpml-benchmark, March 2018.

[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 Appendix