Recap: Private Logistic Regression

Algorithm:

sensitivity $\Delta L = \frac{2}{N}$

1) solve $w^* = \arg\min_w L(w)$

2) draw η with $p(\eta) \propto \exp(-\frac{\Delta L}{\epsilon}||\eta||)$

3) return $w = w^* + \eta$

With a strongly convex loss, we were able to bound the sensitivity of the loss evaluated at its minimizer.

"Output perturbation": Apply vector analogue of Laplace mechanism to non-private solution for (eps, 0)-DP.

Other options

- objective perturbation
- Input perturbation

Private Non-Convex Learning

For non-convex losses:

- We don't know whether our optimizer will (asymptotically) find the global optimum
- We can not bound the loss function at the optimum or anywhere else

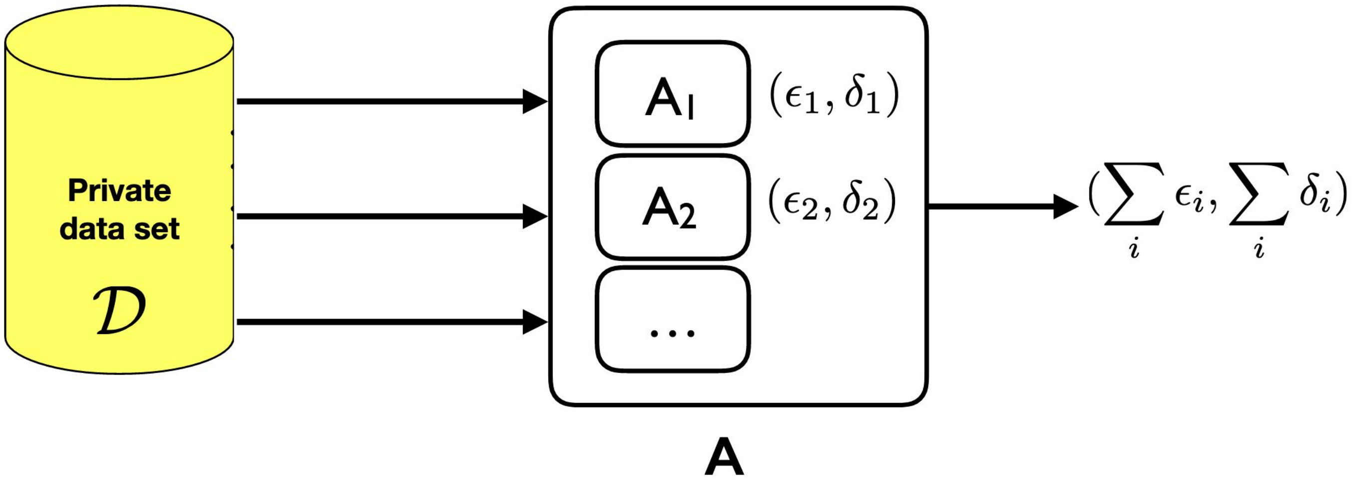
Instead we will make every step of the iterative optimization algorithm private, and somehow account for the overall privacy loss at the end

Private Non-Convex Learning

To discuss DPSGD algorithm, we need to understand

- Basic composition of mechanisms
- “Advanced” composition in an adaptive setup
- Privacy amplification by subsampling
- Per-example gradient computation

Recall Composition Theorem



Total privacy loss is the sum of privacy losses

(Better composition possible — coming up later)

Advanced Composition

What is composition?

- Repeated use of DP algorithms on the same database
- Repeated use of DP algorithms on the different databases that nevertheless may contain shared information about individuals

We can show that the privacy loss over all possible outcomes has a Markov structure, which hints at a better composition

Advanced Composition

Theorem 3.20 (Advanced Composition). For all $\varepsilon, \delta, \delta' \geq 0$, the class of (ε, δ)-differentially private mechanisms satisfies $(\varepsilon', k\delta + \delta')$-differential privacy under k-fold adaptive composition for:

$$\varepsilon' = \sqrt{2k \ln(1/\delta')}\varepsilon + k\varepsilon(e^{\varepsilon} - 1).$$

Privacy by Subsampling

Lemma 3 (Amplification via sampling) If A is 1-differentially private, then for any $\epsilon \in (0,1)$, $A'(\epsilon, \cdot)$ is 2ϵ-differentially private.

Suppose A is a 1-differentially private algorithm that expects data sets from a domain D as input. Consider a new algorithm A', which runs A on a random subsample of $\approx \epsilon n$ points from its input:

Proof: Fix an event S in the output space of A', and two data sets x, x' that differ by a single individual, say $x = x' \cup \{i\}$.

Consider a run of A' on input x. If i is not included in the sample T, then the output is distributed the same as a run of A' on $x' = x \setminus \{i\}$, since the inclusion of i in the sample is independent of the inclusion of other elements. On the other hand, if i is included in the sample T, then the behavior of A on T is only a factor of e off from the behavior of A on $T \setminus \{i\}$. Again, because of independence, the distribution of $T \setminus \{i\}$ is the same as the distribution of T conditioned on the omission of i. For a set $T \subseteq D$, let p_T denote the distribution of $A(T)$. In symbols, we have that for any event S:

$$p_x(S \mid i \notin T) = p_{x'}(S) \quad \text{and} \quad p_x(S \mid i \in T) \in e^{\pm 1} p_{x'}(S).$$

We can put the pieces together, using the fact that i is in T with probability only ϵ:

$$\begin{aligned} p_x(S) &= (1-\epsilon)\cdot p_x(S \mid i \notin T) + \epsilon \cdot p_x(S \mid i \in T) \\ &\le (1-\epsilon)\cdot p_{x'}(S) + \epsilon \cdot e \cdot p_{x'}(S) \\ &= (1+\epsilon(e-1))p_{x'}(S) \\ &\le \exp(2\epsilon)\cdot p_{x'}(S) \end{aligned}$$

We can get a similar lower bound:

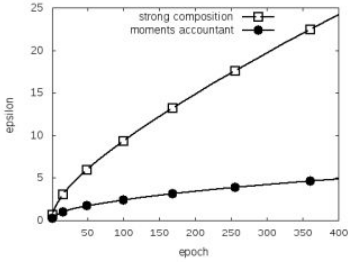
$$\begin{aligned} p_x(S) &= (1-\epsilon)\cdot p_x(S \mid i \notin T) + \epsilon \cdot p_x(S \mid i \in T) \\ &\ge (1-\epsilon)\cdot p_{x'}(S) + \epsilon \cdot \tfrac{1}{e} \cdot p_{x'}(S) \\ &= (1-\epsilon(1-e^{-1}))\cdot p_{x'}(S) \\ &\ge \exp(-\epsilon)\cdot p_{x'}(S) \end{aligned}$$

The last inequality uses the fact that $\epsilon \le 1$. □

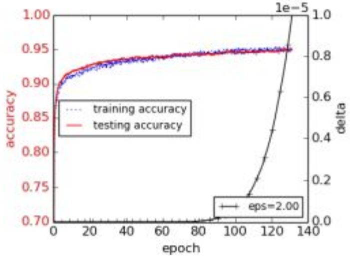
https://adamdsmith.wordpress.com/2009/09/02/sample-secrecy/

Differentially Private SGD

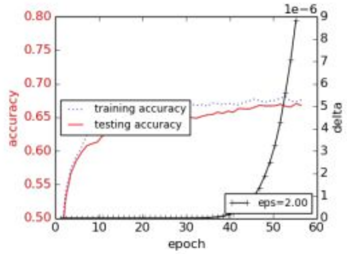
Moments accountant improves bounds



MNIST epoch vs accuracy/privacy



CIFAR-10 epoch vs accuracy/privacy



Algorithm 1 Differentially private SGD (Outline)

Input: Examples $\{x_1, \ldots, x_N\}$, loss function $\mathcal{L}(\theta) = \frac{1}{N}\sum_i \mathcal{L}(\theta, x_i)$. Parameters: learning rate η_t, noise scale σ, group size L, gradient norm bound C.

Initialize θ_0 randomly

for $t \in [T]$ **do**

Take a random sample L_t with sampling probability L/N

Compute gradient

For each $i \in L_t$, compute $\mathbf{g}_t(x_i) \leftarrow \nabla_{\theta_t}\mathcal{L}(\theta_t, x_i)$

Clip gradient

$\bar{\mathbf{g}}_t(x_i) \leftarrow \mathbf{g}_t(x_i) / \max\left(1, \frac{\|\mathbf{g}_t(x_i)\|_2}{C}\right)$

Add noise

$\tilde{\mathbf{g}}_t \leftarrow \frac{1}{L}\left(\sum_i \bar{\mathbf{g}}_t(x_i) + \mathcal{N}(0, \sigma^2 C^2 \mathbf{I})\right)$

Descent

$\theta_{t+1} \leftarrow \theta_t - \eta_t \tilde{\mathbf{g}}_t$

Output θ_T and compute the overall privacy cost (ε, δ) using a privacy accounting method.

[Abadi et al. 2016]

Guarantees final parameters don’t depend too much on individual training examples

Gaussian noise added to the parameter update at every iteration

Privacy loss accumulates over time

The “moments accountant” provides better empirical bounds on (ε,δ)

Differentially Private SGD

Algorithm 1 Differentially private SGD (Outline)

Input: Examples $\{x_1, \ldots, x_N\}$, loss function $\mathcal{L}(\theta) = \frac{1}{N}\sum_i \mathcal{L}(\theta, x_i)$. Parameters: learning rate η_t, noise scale σ, group size L, gradient norm bound C.

Initialize θ_0 randomly

for $t \in [T]$ **do**

Take a random sample L_t with sampling probability L/N

Compute gradient

For each $i \in L_t$, $\boxed{\text{compute } \mathbf{g}_t(x_i) \leftarrow \nabla_{\theta_t} \mathcal{L}(\theta_t, x_i)}$

Clip gradient

$\bar{\mathbf{g}}_t(x_i) \leftarrow \mathbf{g}_t(x_i) / \max\left(1, \frac{\|\mathbf{g}_t(x_i)\|_2}{C}\right)$

Add noise

$\tilde{\mathbf{g}}_t \leftarrow \frac{1}{L}\left(\sum_i \bar{\mathbf{g}}_t(x_i) + \mathcal{N}(0, \sigma^2 C^2 \mathbf{I})\right)$

Descent

$\theta_{t+1} \leftarrow \theta_t - \eta_t \tilde{\mathbf{g}}_t$

Output θ_T and compute the overall privacy cost (ε, δ) using a privacy accounting method.

← when can we efficiently compute per-example gradients?

[Abadi et al. 2016]

Autodiff Recap

$$y = D(\boldsymbol{c}), \quad \boldsymbol{c} = C(\boldsymbol{b}), \quad \boldsymbol{b} = B(\boldsymbol{a}), \quad \boldsymbol{a} = A(\boldsymbol{x})$$

$$F'(\boldsymbol{x}) = \frac{\partial y}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \cdots & \frac{\partial y}{\partial x_n} \end{bmatrix}$$

$$F'(\boldsymbol{x}) = \frac{\partial y}{\partial \boldsymbol{c}} \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}$$

$$\frac{\partial y}{\partial \boldsymbol{c}} = D'(\boldsymbol{c}) \qquad \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} = C'(\boldsymbol{b}) \qquad \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} = B'(\boldsymbol{a}) \qquad \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}} = A'(\boldsymbol{x})$$

http://www.cs.toronto.edu/~duvenaud/talks/Johnson-Automatic-Differentiation.pdf

Autodiff Recap

$$F'(\boldsymbol{x}) = \frac{\partial y}{\partial \boldsymbol{c}}\left(\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}}\left(\underbrace{\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}}\quad\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}}\right)\right)$$

$$\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial b_1}{\partial x_1} & \cdots & \frac{\partial b_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial b_m}{\partial x_1} & \cdots & \frac{\partial b_m}{\partial x_n} \end{bmatrix}$$

Forward accumulation

$$F'(\boldsymbol{x}) = \left(\left(\underbrace{\frac{\partial y}{\partial \boldsymbol{c}}\quad\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}}}\right)\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}}\right)\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}$$

$$\frac{\partial y}{\partial \boldsymbol{b}} = \begin{bmatrix} \frac{\partial y}{\partial b_1} & \cdots & \frac{\partial y}{\partial b_m} \end{bmatrix}$$

Reverse accumulation

http://www.cs.toronto.edu/~duvenaud/talks/Johnson-Automatic-Differentiation.pdf

Autodiff Recap

$$\tilde{F}(x) \in \mathbb{R}^k \text{ (a vector-valued function)}$$

$$\nabla F(x) = \left(\left(\underbrace{\frac{dy}{dc}}_{\in \mathbb{R}^{N_c \times k}} \underbrace{\frac{c}{db}}_{\in \mathbb{R}^{N_b \times N_b}} \right) \frac{db}{da} \right) \frac{da}{dx}$$

Generically, reverse-mode autodiff runs *k* times slower for functions ouputting length-*k* vectors

For the dense layers in MLPs there are tricks to efficiently compute per-example gradient *norms* (see Ian Goodfellow's app note: arxiv:1510.01799)

Autodiff Recap

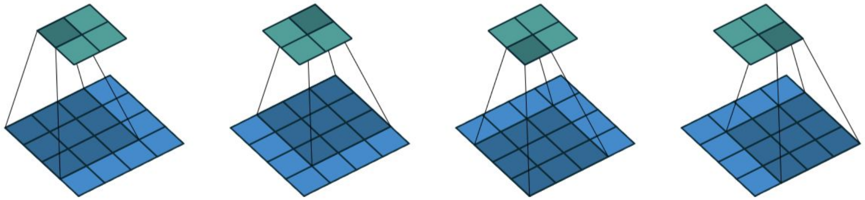


Figure 2.1: (No padding, unit strides) Convolving a 3×3 kernel over a 4×4 input using unit strides (i.e., $i = 4$, $k = 3$, $s = 1$ and $p = 0$).

Take for example the convolution represented in Figure 2.1. If the input and output were to be unrolled into vectors from left to right, top to bottom, the convolution could be represented as a sparse matrix $\mathbf{C}$ where the non-zero elements are the elements $w_{i,j}$ of the kernel (with i and j being the row and column of the kernel respectively):

$$\begin{pmatrix} w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 & 0 \\ 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 \\ 0 & 0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} \end{pmatrix}$$

This linear operation takes the input matrix flattened as a 16-dimensional vector and produces a 4-dimensional vector that is later reshaped as the 2×2 output matrix.

Arxiv: 1603.07285

However in conv nets the picture gets more complicated...

The Nuts and Bolts

Algorithm 1 Differentially private SGD (Outline)

Input: Examples $\{x_1, \ldots, x_N\}$, loss function $\mathcal{L}(\theta) = \frac{1}{N}\sum_i \mathcal{L}(\theta, x_i)$. Parameters: learning rate η_t, noise scale σ, group size L, gradient norm bound C.

Initialize θ_0 randomly

for $t \in [T]$ **do**

Take a random sample L_t with sampling probability L/N

Compute gradient

For each $i \in L_t$, compute $\mathbf{g}_t(x_i) \leftarrow \nabla_{\theta_t}\mathcal{L}(\theta_t, x_i)$

Clip gradient

$\bar{\mathbf{g}}_t(x_i) \leftarrow \mathbf{g}_t(x_i) / \max\left(1, \frac{\|\mathbf{g}_t(x_i)\|_2}{C}\right)$

Add noise

$\tilde{\mathbf{g}}_t \leftarrow \frac{1}{L}\left(\sum_i \bar{\mathbf{g}}_t(x_i) + \mathcal{N}(0, \sigma^2 C^2 \mathbf{I})\right)$

Descent

$\theta_{t+1} \leftarrow \theta_t - \eta_t \tilde{\mathbf{g}}_t$

Output θ_T and compute the overall privacy cost (ε, δ) using a privacy accounting method.

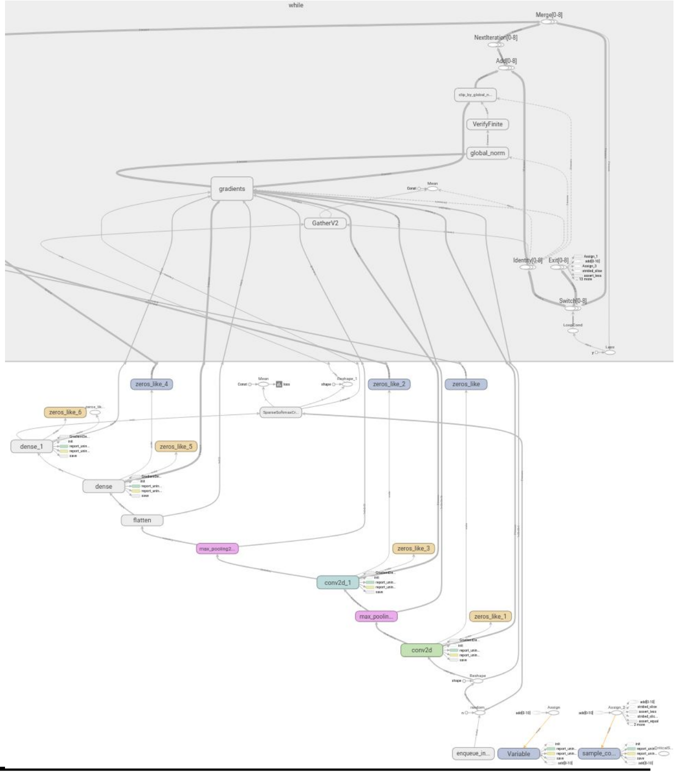
[Abadi et al. 2016]

Tensorflow implementation requires differentiating a vector loss

Privately training convolutional layers is impractical with TF (> 2 min/epoch on MNIST)

Abadi et al. learn conv layers from “public” CIFAR-100 and privately fine-tune dense layers on CIFAR-100

One current research/production approach for large-scale private learning at Google: TPUs



JAX to the Rescue

Algorithm 1 Differentially private SGD (Outline)

Input: Examples $\{x_1, \dots, x_N\}$, loss function $\mathcal{L}(\theta) = \frac{1}{N}\sum_i \mathcal{L}(\theta, x_i)$. Parameters: learning rate η_t, noise scale σ, group size L, gradient norm bound C.

Initialize θ_0 randomly

for $t \in [T]$ **do**

Take a random sample L_t with sampling probability L/N

Compute gradient

For each $i \in L_t$, compute $\mathbf{g}_t(x_i) \leftarrow \nabla_{\theta_t}\mathcal{L}(\theta_t, x_i)$

Clip gradient

$\bar{\mathbf{g}}_t(x_i) \leftarrow \mathbf{g}_t(x_i) / \max\left(1, \frac{\|\mathbf{g}_t(x_i)\|_2}{C}\right)$

Add noise

$\tilde{\mathbf{g}}_t \leftarrow \frac{1}{L}\left(\sum_i \bar{\mathbf{g}}_t(x_i) + \mathcal{N}(0, \sigma^2 C^2 \mathbf{I})\right)$

Descent

$\theta_{t+1} \leftarrow \theta_t - \eta_t \tilde{\mathbf{g}}_t$

Output θ_T and compute the overall privacy cost (ε, δ) using a privacy accounting method.

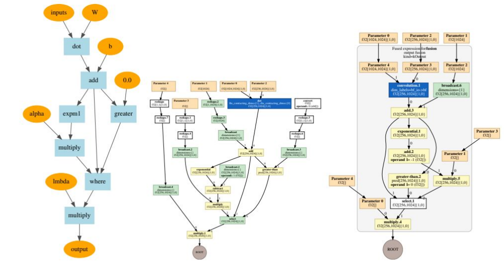
```python
def private_grad(params, batch, rng, l2_norm_clip, noise_multiplier,
                 batch_size):
  """Return differentially private gradients for params, evaluated on batch."""

  def _clipped_grad(params, single_example_batch):
    """Evaluate gradient for a single-example batch and clip its grad norm."""
    grads = grad(loss)(params, single_example_batch)

    # NOTE: this could be achieved in a slightly more readable way using
    # flatten_util.tree_ravel, but that version runs 10% slower :'(
    nonempty_grads, tree_def = tree_util.tree_flatten(grads)
    total_grad_norm = np.linalg.norm(
        [np.linalg.norm(neg.ravel()) for neg in nonempty_grads])
    divisor = stop_gradient(np.amax((total_grad_norm / l2_norm_clip, 1.)))
    normalized_nonempty_grads = [g / divisor for g in nonempty_grads]
    return tree_util.tree_unflatten(tree_def, normalized_nonempty_grads)

  px_clipped_grad_fn = vmap(partial(_clipped_grad, params))
  sum_ = lambda n: np.sum(n, 0)  # aggregate
  std_dev = l2_norm_clip * noise_multiplier
  noise_ = lambda n: n + std_dev * random.normal(rng, n.shape)
  normalize_ = lambda n: n / float(batch_size)
  tree_map = tree_util.tree_map
  aggregated_clipped_grads = tree_map(sum_, px_clipped_grad_fn(batch))
  noised_aggregated_clipped_grads = tree_map(noise_, aggregated_clipped_grads)
  normalized_noised_aggregated_clipped_grads = (
      tree_map(normalize_, noised_aggregated_clipped_grads)
  )
  return normalized_noised_aggregated_clipped_grads
```

```python
@jit
def private_update(rng, i, opt_state, batch):
  params = optimizers.get_params(opt_state)
  rng = random.fold_in(rng, i)  # get new key for new random numbers
  return opt_update(
      i,
      private_grad(params, batch, rng, FLAGS.l2_norm_clip,
                   FLAGS.noise_multiplier, FLAGS.batch_size), opt_state)
```



(a) JAX trace. (b) HLO before fusion. (c) HLO after fusion.

[Frostig et al. 2018]

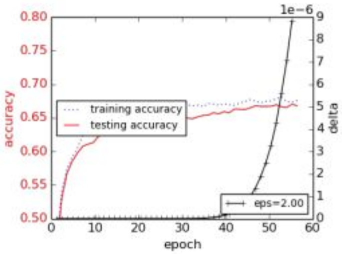
JAX just-in-time compiles NumPy code to GPU/TPU-friendly XLA instructions

DPSGD-TF takes > 2 mins/epoch on MNIST

DPSGD-JAX runs ~4 secs/epoch after tens of seconds to XLA compile (30X speedup!)

https://github.com/google/jax/blob/master/examples/differentially_private_sgd.py

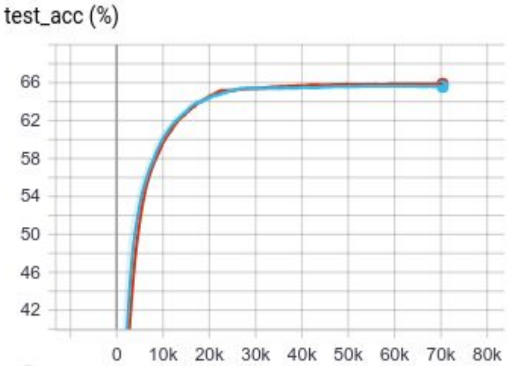
JAX to the Rescue

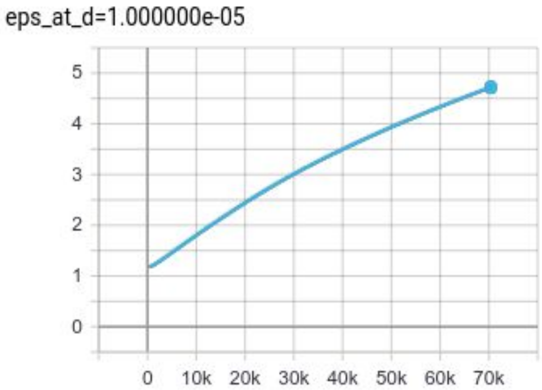


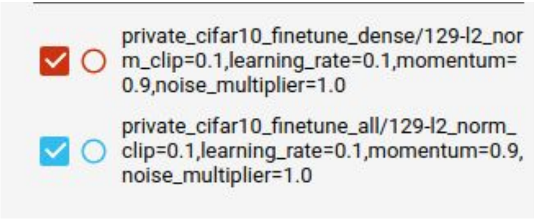
[Abadi et al. 2016]

Now we can train the whole conv net!

But it didn't help for the benchmark problems described by Abadi et al...

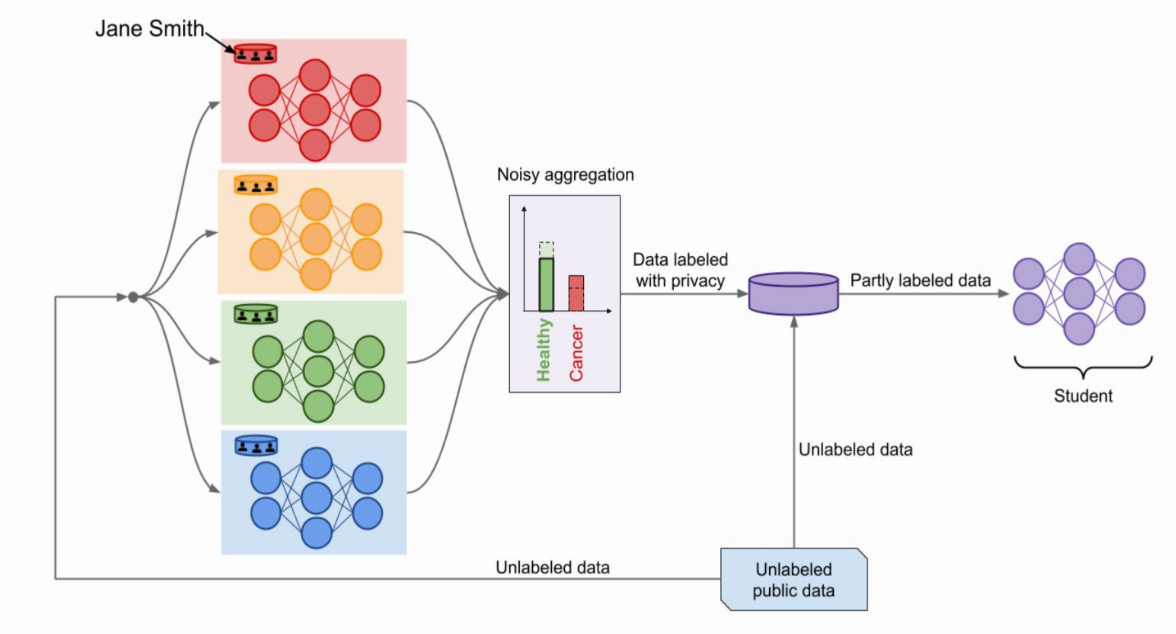






https://github.com/google/jax/blob/master/examples/differentially_private_sgd.py

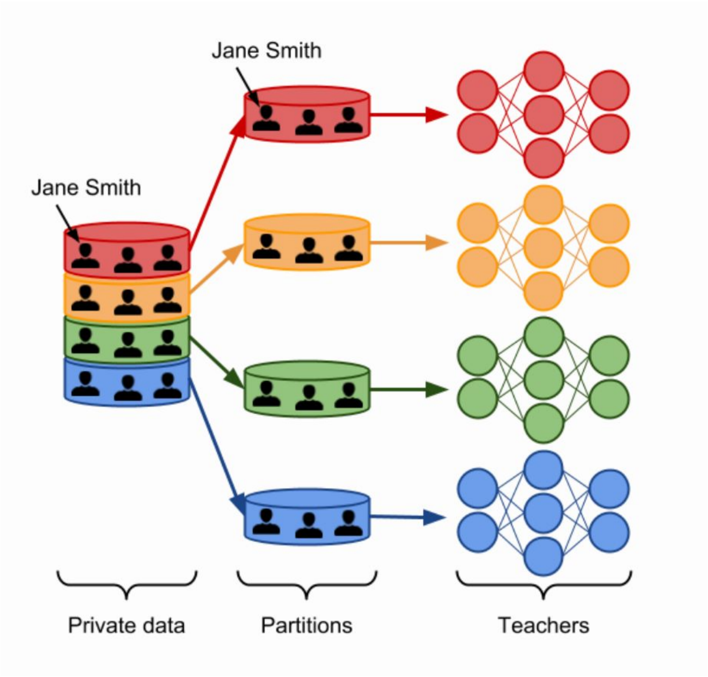
PATE



Private Aggregation of Teacher Ensembles [Papernot et al 2017, Papernot et al 2018]

Key idea: instead of adding noise to gradients, add noise to *labels*

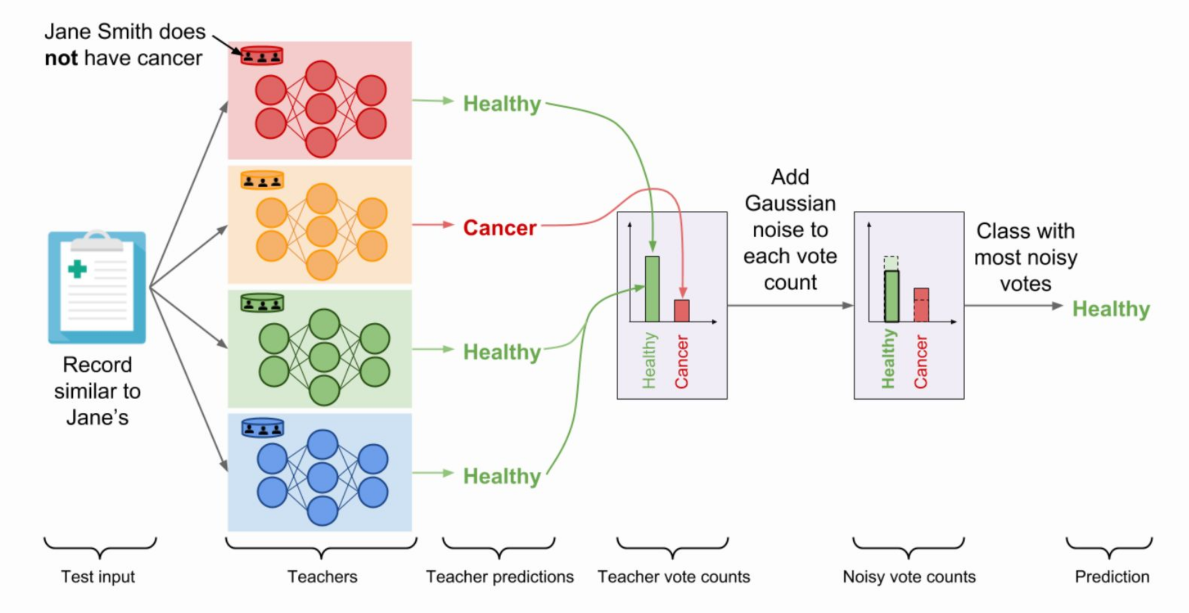
PATE



Start by partitioning private data into disjoint sets

Each teacher trains (non-privately) on its corresponding subset

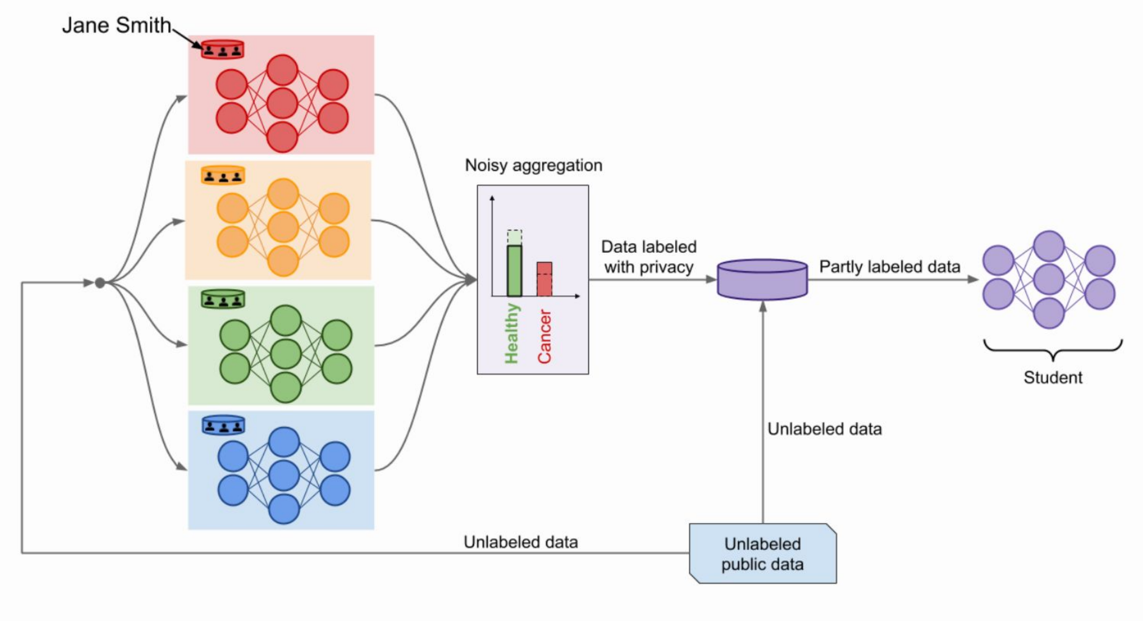
PATE



Private predictions can now be generated via the exponential mechanism, where the "score" is computed with an election amongst teachers - output the noisy winner

We now have private inference, but we lose privacy every time we predict. We would like the privacy loss to be constant at test time.

PATE

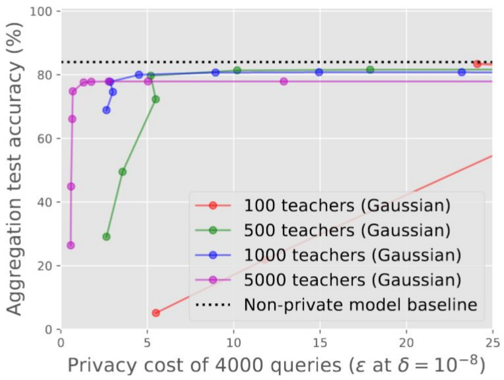


We can instead use the noisy labels provided by the teachers to train a student

We leak privacy during training but at test time we lose no further privacy (due to post-processing thm)

Because the student should use as few labels as possible, unlabeled public data is leveraged in a semi-supervised setup.

PATE



Dataset	Aggregator	Queries answered	Privacy bound ε	Accuracy Student	Accuracy Baseline
MNIST	LNMax (Papernot et al., 2017)	100	2.04	98.0%	99.2%
	LNMax (Papernot et al., 2017)	1,000	8.03	98.1%	
	Confident-GNMax (T=200, σ_1=150, σ_2=40)	286	**1.97**	**98.5%**	
SVHN	LNMax (Papernot et al., 2017)	500	5.04	82.7%	92.8%
	LNMax (Papernot et al., 2017)	1,000	8.19	90.7%	
	Confident-GNMax (T=300, σ_1=200, σ_2=40)	3,098	**4.96**	**91.6%**	
Adult	LNMax (Papernot et al., 2017)	500	2.66	83.0%	85.0%
	Confident-GNMax (T=300, σ_1=200, σ_2=40)	524	**1.90**	**83.7%**	
Glyph	LNMax	4,000	4.3	72.4%	82.2%
	Confident-GNMax (T=1000, σ_1=500, σ_2=100)	10,762	2.03	**75.5%**	
	Interactive-GNMax, two rounds	4,341	**0.837**	73.2%	

https://arxiv.org/pdf/1802.08908.pdf