Machine Learning with Differential Privacy in TensorFlow

Mar 26, 2019

by Nicolas Papernot

Differential privacy is a framework for measuring the privacy guarantees provided by an algorithm. Through the lens of differential privacy, we can design machine learning algorithms that responsibly train models on private data. Learning with differential privacy provides provable guarantees of privacy, mitigating the risk of exposing sensitive training data in machine learning. Intuitively, a model trained with differential privacy should not be affected by any single training example, or small set of training examples, in its data set.

You may recall our previous blog post on PATE, an approach that achieves private learning by carefully coordinating the activity of several different ML models [Papernot et al.]. In this post, you will learn how to train a differentially private model with another approach that relies on Differentially Private Stochastic Gradient Descent (DP-SGD) [Abadi et al.]. DP-SGD and PATE are two different ways to achieve the same goal of privacy-preserving machine learning. DP-SGD makes less assumptions about the ML task than PATE, but this comes at the expense of making modifications to the training algorithm.

Indeed, DP-SGD is a modification of the stochastic gradient descent algorithm, which is the basis for many optimizers that are popular in machine learning. Models trained with DP-SGD have provable privacy guarantees expressed in terms of differential privacy (we will explain what this means at the end of this post). We will be using the TensorFlow Privacy library, which provides an implementation of DP-SGD, to illustrate our presentation of DP-SGD and provide a hands-on tutorial.

The only prerequisite for following this tutorial is to be able to train a simple neural network with TensorFlow. If you are not familiar with convolutional neural networks or how to train them, we recommend reading this tutorial first to get started with TensorFlow and machine learning.

Upon completing the tutorial presented in this post, you will be able to wrap existing optimizers (e.g., SGD, Adam, ...) into their differentially private counterparts using TensorFlow (TF) Privacy. You will also learn how to tune the parameters introduced by differentially private optimization. Finally, we will learn how to measure the privacy guarantees provided using analysis tools included in TF Privacy.

Getting started

Before we get started with DP-SGD and TF Privacy, we need to put together a script that trains a simple neural network with TensorFlow.

In the interest of keeping this tutorial focused on the privacy aspects of training, we've included such a script as companion code for this blog post in the `walkthrough` subdirectory of the `tutorials` found in the TensorFlow Privacy repository. The code found in the file `mnist_scratch.py` trains a small convolutional neural network on the MNIST dataset for handwriting recognition. This script will be used as the basis for our exercise below.

Next, we highlight some important code snippets from the `mnist_scratch.py` script.

The first snippet includes the definition of a convolutional neural network using `tf.keras.layers`. The model contains two convolutional layers coupled with max pooling layers, a fully-connected layer, and a softmax. The model's output is a vector where each component indicates how likely the input is to be in one of the 10 classes of the handwriting recognition problem we considered. If any of this sounds unfamiliar, we recommend reading this tutorial first to get started with TensorFlow and machine learning.

```
input_layer = tf.reshape(features['x'], [-1, 28, 28, 1])
y = tf.keras.layers.Conv2D(16, 8,
                           strides=2,
                           padding='same',
                           activation='relu').apply(input_layer)
y = tf.keras.layers.MaxPool2D(2, 1).apply(y)
y = tf.keras.layers.Conv2D(32, 4,
                           strides=2,
                           padding='valid',
                           activation='relu').apply(y)
y = tf.keras.layers.MaxPool2D(2, 1).apply(y)
y = tf.keras.layers.Flatten().apply(y)
y = tf.keras.layers.Dense(32, activation='relu').apply(y)
logits = tf.keras.layers.Dense(10).apply(y)
predicted_labels = tf.argmax(input=logits, axis=1)
```

The second snippet shows how the model is trained using the `tf.Estimator` API, which takes care of all the boilerplate code required to form minibatches used to train and evaluate the model. To prepare ourselves for the modifications we will be making to provide differential privacy, we still expose the loop over different epochs of learning: an epoch is defined as one pass over all of the training points included in the training set.

```
steps_per_epoch = 60000 // FLAGS.batch_size
for epoch in range(1, FLAGS.epochs + 1):
  # Train the model for one epoch.
  mnist_classifier.train(input_fn=train_input_fn, steps=steps_per_epoch)

  # Evaluate the model and print results
  eval_results = mnist_classifier.evaluate(input_fn=eval_input_fn)
  test_accuracy = eval_results['accuracy']
  print('Test accuracy after %d epochs is: %.3f' % (epoch, test_accuracy))
```

We are now ready to train our MNIST model without privacy. The model should achieve above 99% test accuracy after 15 epochs at a learning rate of 0.15 on minibatches of 256 training points.

```
python mnist_scratch.py
```

Stochastic Gradient Descent

Before we dive into how DP-SGD and TF Privacy can be used to provide differential privacy during machine learning, we first provide a brief overview of the stochastic gradient descent algorithm, which is one of the most popular optimizers for neural networks.

Stochastic gradient descent is an iterative procedure. At each iteration, a batch of data is randomly sampled from the training set (this is where the *stochasticity* comes from). The error between the model's prediction and the training labels is then computed. This error, also called the loss, is then differentiated with respect to the model's parameters. These derivatives (or *gradients*) tell us how we should update each parameter to bring the model closer to predicting the correct label. Iteratively recomputing gradients and applying them to update the model's parameters is what is referred to as the *descent*. To summarize, the following steps are repeated until the model's performance is satisfactory:

1. Sample a minibatch of training points `(x, y)` where `x` is an input and `y` a label.
2. Compute loss (i.e., error) `L(theta, x, y)` between the model's prediction `f_theta(x)` and label `y` where `theta` represents the model parameters.
3. Compute gradient of the loss `L(theta, x, y)` with respect to the model parameters `theta`.
4. Multiply these gradients by the learning rate and apply the product to update model parameters `theta`.

Modifications needed to make stochastic gradient descent a differentially private algorithm

Two modifications are needed to ensure that stochastic gradient descent is a differentially private algorithm.

First, the sensitivity of each gradient needs to be bounded. In other words, we need to limit how much each individual training point sampled in a minibatch can influence the resulting gradient computation. This can be done by clipping each gradient computed on each training point between steps 3 and 4 above. Intuitively, this allows us to bound how much each training point can possibly impact model parameters.

Second, we need to randomize the algorithm's behavior to make it statistically impossible to know whether or not a particular point was included in the training set by comparing the updates stochastic gradient descent applies when it operates with or without this particular point in the training set. This is achieved by sampling random noise and adding it to the clipped gradients.

Thus, here is the stochastic gradient descent algorithm adapted from above to be differentially private:

1. Sample a minibatch of training points `(x, y)` where `x` is an input and `y` a label.
2. Compute loss (i.e., error) `L(theta, x, y)` between the model's prediction `f_theta(x)` and label `y` where `theta` represents the model parameters.
3. Compute gradient of the loss `L(theta, x, y)` with respect to the model parameters `theta`.
4. Clip gradients, per training example included in the minibatch, to ensure each gradient has a known maximum Euclidean norm.
5. Add random noise to the clipped gradients.
6. Multiply these clipped and noised gradients by the learning rate and apply the product to update model parameters `theta`.

Implementing DP-SGD with TF Privacy

It's now time to make changes to the code we started with to take into account the two modifications outlined in the previous paragraph: gradient clipping and noising. This is where TF Privacy kicks in: it provides code that wraps an existing TF optimizer to create a variant that performs both of these steps needed to obtain differential privacy.

As mentioned above, step 1 of the algorithm, that is forming minibatches of training data and labels, is implemented by the `tf.Estimator` API in our tutorial. We can thus go straight to step 2 of the algorithm outlined above and compute the loss (i.e., model error) between the model's predictions and labels.

```
vector_loss = tf.nn.sparse_softmax_cross_entropy_with_logits(
    labels=labels, logits=logits)
```

TensorFlow provides implementations of common losses, here we use the cross-entropy, which is well-suited for our classification problem. Note how we computed the loss as a vector, where each component of the vector corresponds to an individual training point and label. This is required to support per example gradient manipulation later at step 4.

We are now ready to create an optimizer. In TensorFlow, an optimizer object can be instantiated by passing it a learning rate value, which is used in step 6 outlined above. This is what the code would look like *without* differential privacy:

```
optimizer = tf.train.GradientDescentOptimizer(FLAGS.learning_rate)
train_op = optimizer.minimize(loss=scalar_loss)
```

Note that our code snippet assumes that a TensorFlow flag was defined for the learning rate value.

Now, we use the `optimizers.dp_optimizer` module of TF Privacy to implement the optimizer with differential privacy. Under the hood, this code implements steps 3-6 of the algorithm above:

```
optimizer = optimizers.dp_optimizer.DPGradientDescentGaussianOptimizer(
    l2_norm_clip=FLAGS.l2_norm_clip,
    noise_multiplier=FLAGS.noise_multiplier,
    num_microbatches=FLAGS.microbatches,
```

```
    learning_rate=FLAGS.learning_rate,


    population_size=60000)
train_op = optimizer.minimize(loss=vector_loss)
```

In these two code snippets, we used the stochastic gradient descent optimizer but it could be replaced by another optimizer implemented in TensorFlow. For instance, the `AdamOptimizer` can be replaced by `DPAdamGaussianOptimizer`. In addition to the standard optimizers already included in TF Privacy, most optimizers which are objects from a child class of `tf.train.Optimizer` can be made differentially private by calling `optimizers.dp_optimizer.make_gaussian_optimizer_class()`.

As you can see, only one line needs to change but there are a few things going on that are best to unwrap before we continue. In addition to the learning rate, we passed the size of the training set as the `population_size` parameter. This is used to measure the strength of privacy achieved; we will come back to this accounting aspect later.

More importantly, TF Privacy introduces three new hyperparameters to the optimizer object: `l2_norm_clip`, `noise_multiplier`, and `num_microbatches`. You may have deduced what `l2_norm_clip` and `noise_multiplier` are from the two changes outlined above.

Parameter `l2_norm_clip` is the maximum Euclidean norm of each individual gradient that is computed on an individual training example from a minibatch. This parameter is used to bound the optimizer's sensitivity to individual training points. Note how in order for the optimizer to be able to compute these per example gradients, we must pass it a *vector* loss as defined previously, rather than the loss averaged over the entire minibatch.

Next, the `noise_multiplier` parameter is used to control how much noise is sampled and added to gradients before they are applied by the optimizer. Generally, more noise results in better privacy (often, but not necessarily, at the expense of lower utility).

The third parameter relates to an aspect of DP-SGD that was not discussed previously. In practice, clipping gradients on a per example basis can be detrimental to the performance of our approach because computations can no longer be batched and parallelized at the granularity of minibatches. Hence, we introduce a new granularity by splitting each minibatch into multiple microbatches [McMahan et al.]. Rather than clipping gradients on a per example basis, we clip them on a microbatch basis. For instance, if we have a minibatch of 256 training examples, rather than clipping each of the 256 gradients individually, we would clip 32 gradients averaged over microbatches of 8 training examples when `num_microbatches=32`. This allows for some degree of parallelism. Hence, one can think of `num_microbatches` as a parameter that allows us to trade off performance (when the parameter is set to a small value) with utility (when the parameter is set to a value close to the minibatch size).

Once you've implemented all these changes, try training your model again with the differentially private stochastic gradient optimizer. You can use the following hyperparameter values to obtain a reasonable model (95% test accuracy):

```
learning_rate=0.25
noise_multiplier=1.3
l2_norm_clip=1.5
batch_size=256
epochs=15
num_microbatches=256
```

Measuring the privacy guarantee achieved

At this point, we made all the changes needed to train our model with differential privacy. Congratulations! Yet, we are still missing one crucial piece of the puzzle: we have not computed the privacy guarantee achieved. Recall the two modifications we made to the original stochastic gradient descent algorithm: clip and randomize gradients.

It is intuitive to machine learning practitioners how clipping gradients limits the ability of the model to overfit to any of its training points. In fact, gradient clipping is commonly employed in machine learning even when privacy is not a concern. The intuition for introducing randomness to a learning algorithm that is already randomized is a little more subtle but this additional randomization is required to make it hard to tell which behavioral aspects of the model defined by the learned parameters came from randomness and which came from the training data. Without randomness, we would be able to ask questions like: "What parameters does the learning algorithm choose when we train it on this specific dataset?" With randomness in the learning algorithm, we instead ask questions like: "What is the probability that the learning algorithm will choose parameters in this set of possible parameters, when we train it on this specific dataset?"

We use a version of differential privacy which requires that the probability of learning any particular set of parameters stays roughly the same if we change a single training example in the training set. This could mean to add a training example, remove a training example, or change the values within one training example. The intuition is that if a single training point does not affect the outcome of learning, the information contained in that training point cannot be memorized and the privacy of the individual who contributed this data point to our dataset is respected. We often refer to this probability as the privacy budget: smaller privacy budgets correspond to stronger privacy guarantees.

Accounting required to compute the privacy budget spent to train our machine learning model is another feature provided by TF Privacy. Knowing what level of differential privacy was achieved allows us to put into perspective the drop in utility that is often observed when switching to differentially private optimization. It also allows us to compare two models objectively to determine which of the two is more privacy-preserving than the other.

Before we derive a bound on the privacy guarantee achieved by our optimizer, we first need to identify all the parameters that are relevant to measuring the potential privacy loss induced by training. These are the `noise_multiplier`, the sampling ratio `q` (the probability of an individual training point being included in a minibatch), and the number of `steps` the optimizer takes over the training data. We simply report the `noise_multiplier` value provided to the optimizer and compute the sampling ratio and number of steps as follows:

```
noise_multiplier = FLAGS.noise_multiplier
sampling_probability = FLAGS.batch_size / 60000
steps = FLAGS.epochs * 60000 // FLAGS.batch_size
```

At a high level, the privacy analysis measures how including or excluding any particular point in the training data is likely to change the probability that we learn any particular set of parameters. In other words, the analysis measures the difference between the distributions of model parameters on neighboring training sets (pairs of any training sets with a Hamming distance of 1). In TF Privacy, we use the Rényi divergence to measure this distance between distributions. Indeed, our analysis is performed in the framework of Rényi Differential Privacy (RDP), which is a generalization of pure differential privacy [Mironov]. RDP is a useful tool here because it is particularly well suited to analyze the differential privacy guarantees provided by sampling followed by Gaussian noise addition, which is how gradients are randomized in the TF Privacy implementation of the DP-SGD optimizer.

We will express our differential privacy guarantee using two parameters: `epsilon` and `delta`.

- Delta bounds the probability of our privacy guarantee not holding. A rule of thumb is to set it to be less than the inverse of the training data size (i.e., the population size). Here, we set it to `10^-5` because MNIST has 60000 training points.

- Epsilon measures the strength of our privacy guarantee. In the case of differentially private machine learning, it gives a bound on how much the probability of a particular model output can vary by including (or removing) a single training example. We usually want it to be a small constant. However, this is only an upper bound, and a large value of epsilon could still mean good practical privacy.

The TF Privacy library provides two methods relevant to derive privacy guarantees achieved from the three parameters outlined in the last code snippet: `compute_rdp` and `get_privacy_spent`. These methods are found in its `analysis.rdp_accountant` module. Here is how to use them.

First, we need to define a list of orders, at which the Rényi divergence will be computed. The first method `compute_rdp` returns the Rényi differential privacy achieved by the Gaussian mechanism applied to gradients in DP-SGD, for each of these orders.

```
orders = [1 + x / 10. for x in range(1, 100)] + list(range(12, 64))
rdp = compute_rdp(q=sampling_probability,
                  noise_multiplier=FLAGS.noise_multiplier,
                  steps=steps,
                  orders=orders)
```

Then, the method `get_privacy_spent` computes the best `epsilon` for a given `target_delta` value of delta by taking the minimum over all orders.

```
epsilon = get_privacy_spent(orders, rdp, target_delta=1e-5)[0]
```

Running the code snippets above with the hyperparameter values used during training will estimate the `epsilon` value that was achieved by the differentially private optimizer, and thus the strength of the privacy guarantee which comes with the model we trained. Once we computed the value of `epsilon`, interpreting this value is at times difficult. One possibility is to purposely insert secrets in the model's training set and measure how likely they are to be leaked by a differentially private model (compared to a non-private model) at inference time [Carlini et al.].

Putting all the pieces together

We covered a lot in this blog post! If you made all the changes discussed directly into the `mnist_scratch.py` file, you should have been able to train a differentially private neural network on MNIST and measure the privacy guarantee achieved.

However, in case you ran into an issue or you'd like to see what a complete implementation looks like, the "solution" to the tutorial presented in this blog post can be found in the tutorials directory of TF Privacy. It is the script called `mnist_dpsgd_tutorial.py`.

Acknowledgements

The author would like to thank Ilya Mironov, Ulfar Erlingsson, and Nicholas Carlini for very helpful comments on a draft of this document.