<DRAFT> Mitigating Forgetting in Small Federated Learning Networks

Bill Paseman <bill@rarekidneycancer.org>
Paseman & Associates Saratoga, California, 95070 U.S.A.

(Abstract) We describe a series of "Federated Learning" experiments which create "Deep Learning" models while preserving the privacy of their distributed, siloed datasets. We do this by creating randomized equal length mini-batches in each silo at the beginning of each epoch, running Stochastic Gradient Descent locally, then combining the results and looping to the next epoch. Scheduling can be done either peer-to-peer or using a central server. This approach avoids the effect of "forgetting" (model detuning) which occurs when a fully-programmed model is passed to each silo in succession for training. It is suited to organizations which cannot overtly make their data public such as pharmaceutical and healthcare organizations who want to jointly create a Deep Learning model using all their datasets without exposing their data (due to HIPAA or competitive reasons). Questions can be directed to `bill@rarekidneycancer.org.`

1. INTRODUCTION

Cancer drug development is slow and costly. Just 6.6% of cancer patients currently see benefits from existing drugs. Also, at the current rate of progress, it would take more than 200 years for all existing patients to be helped. [20180424] One way to mitigate both time and cost is to automate early stages of the drug development pipeline. Several stages of various pipelines now utilize Deep Learning models to assist in this.

Unfortunately, Deep Learning models require a great deal of data and most of that data is fragmented and resides behind the paywalls of disparate organizations. Collecting the data into a central repository is difficult due to a variety of competitive, legal and privacy constraints (such as HIPAA[HIPAA]). What is needed, and what is described here, are Federated Learning mechanisms whereby various organizations can collaborate while maintaining control over their own data. Here, we describe several Federated Learning mechanisms which overcome model “detuning” (called “Catastrophic Forgetting” in the literature) which can come about when implementing “Federated Learning” in a distributed environment.

2. BACKGROUND

2.1. Deep Learning

Deep Learning (DL) is a machine learning method based on learning data representations, as opposed to task-specific algorithms. It is especially attractive in an environment like pharmaceutical development where (arguably) datasets, not algorithms are the key limiting factor to progress. Here, DL uses Deep Neural Nets (DNNs) which consist of multiple layers of Artificial Neural Networks (ANNs) between the DNN input and output layers. ANNs in turn consist of arrays of neurons, which are linear regressions followed by a non-linear activation operator (such as sigmoid).

In order to train the DNN, a back-propagation algorithm is used to derive gradients for each layer. Stochastic gradient descent (SGD) and its extensions are central to optimizing the operation of most DL algorithms, and to our implementations in particular.

We illustrate our findings using the DL code at [neuralnetworksanddeeplearning.com] and the MNIST dataset [MNIST] which contains tens of thousands of scanned images of handwritten digits, together with their correct classifications.

2.2. Federated Learning

As described in “Federated Learning: Collaborative Machine Learning without Centralized Training Data” [20170406] , Federated Learning is a “learning task .. solved by a loose federation of participating devices (which we refer to as clients) which are coordinated by a central server. Each client has a local training dataset which is never uploaded to the server. Instead, each client computes an update to the current global model maintained by the server, and only this update is communicated.” Federated Learning works by “decoupling of model training from the need for direct access to the raw training data.” This early research focused on parallel utilization of a slow network of many mobile phones each containing a relatively small amount of data to implement a variation of Stochastic gradient descent. There are several issues here for us.

- **Few “data heavy” clients vs many “data lite” clients** - We are not as interested in the gigabytes of data stored on 1,000’s of mobile phones (“data lite”) as we are in HIPAA [HIPAA] compliant medical datasets containing petabytes siloed at dozens of institutions (“data heavy”). Recent research [20181208], [20190311] has noted that moving from a “data lite” to “data heavy” operating point impacts
    - Statistics - Samples are more likely to be independent and identically distributed (iid).
    - Communication costs-more examples per client, so less communication is done.
    - Security - more data is aggregated per client, so it is less likely that data can be reverse engineered from the model.
- **Peer-to-Peer vs. Central Server** - Like [20181208], [20190311], our operating point deals with substantially fewer, larger compute

nodes. As such, we also consider peer-to-peer processing as well as server centric processing.

- **Forgetting** - Finally, although SGD and its variants are probably the most used optimization algorithms for machine learning, we encountered "Forgetting" when applying this approach to Deep Learning in a distributed environment with few datasources, each containing a lot of data. We have not seen this discussed in the Federated Learning Literature and discuss it more in the next section.

2.3. Catastrophic Forgetting

As described in "Measuring Catastrophic Forgetting in Neural Networks"[20170808], "Once a network is trained to do a specific task, e.g., fine-grained bird classification, it cannot easily be trained to do new tasks, e.g., incrementally learning to recognize additional bird species or learning an entirely different task such as fine-grained flower recognition. When new tasks are added, deep neural networks are prone to catastrophically forgetting previously learned information." In point of fact, this isn't a particularly new problem. After reaching masters level, some chess computer programmers turn off "learning mode" in their chess-playing programs to prevent "detuning" them when they play lesser skilled opponents. Apparently, like chess programs, DL networks are known by the company they keep.

3. ALGORITHMS

3.1. Nielsen Stochastic Gradient Descent

Algorithm 1: Nielsen SGD (NSGD)

```
def NSGD(self, training_data, epochs,
        mini_batch_size, eta, test_data=None):
  """Train the neural network using mini-batch
stochastic gradient descent.  The
"training_data" is a list of tuples "(x, y)"
representing the training inputs and the
desired outputs.  The other non-optional
parameters are self-explanatory.  If
"test_data" is provided then the network will
be evaluated against the test data after each
epoch, and partial progress printed out.  This
is useful for tracking progress, but slows
things down substantially."""
  if test_data: n_test = len(test_data)
  n = len(training_data)
  for j in xrange(epochs):
    random.shuffle(training_data)
    mini_batches = [
      training_data[k:k+mini_batch_size]
      for k in xrange(0, n, mini_batch_size)]
    for mini_batch in mini_batches:
      self.update_mini_batch(mini_batch, eta)
    if test_data:
       print "Epoch {0}: {1} / {2}".format(
         j, self.evaluate(test_data), n_test)
    else:
       print "Epoch {0} complete".format(j)
```

Central to the Deep Learning Algorithm is the SGD (Stochastic Gradient Descent) Routine. Above is the original SGD routine we took from Nielsen's work [neuralnetworks-anddeeplearning.com].

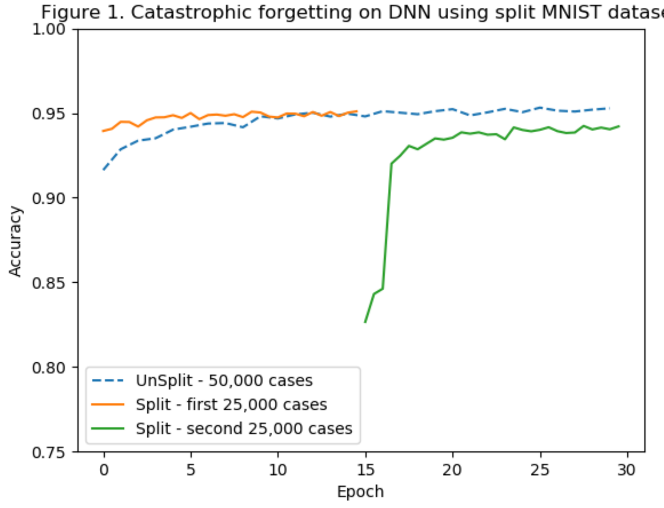
3.2. Forgetting in NSGD

One might think that splitting the Deep Learning Task amongst different data silos is relatively straightforward. Just let silo1 train a model on private data behind its firewall and then pass the model to silo2, letting it extend the model behind its firewall and so on.

We can model this approach for two silos by splitting the MNIST target dataset into two, and measuring the accuracy (the percent of test cases that the resulting model 'passes') for each approach.

The results for the MNIST dataset are shown below.

Figure 1. Catastrophic forgetting on DNN using split MNIST dataset



The blue dashed line labeled "UnSplit" shows the effect of training a network using NSGD on 50,000 MNIST examples. Note that it reaches 95% accuracy after 15 epochs.

The Orange solid line shows the effect of training on the first 25,000 MNIST examples. Note that it also reaches 95% accuracy after 15 epochs. The Green line shows the detuning (or "forgetting") that occurs when the second 25,000 MNIST examples are used to train the network created by the first 25,000 MNIST examples.

The reason this "forgetting" occurs is that the above mechanism breaks a fundamental assumption central to SGD, namely that the samples are independent and identically distributed (iid). Given that the data set was arbitrarily split in two, it would require a fair amount of luck to have the statistics between the first and second halves match one another.

One positive aspect of this approach is that it effectively anonymizes the data. One silo can't "reverse compile" data that the other silo used to create the model. We now discuss a way to get the best of both worlds: anonymized data and "good enough" random data sampling.

3.3. Serial Silo Randomization SGD

Our approach is to try and make the Central limit theorem work for us. Clearly, if we made our mini-batch size equal to one and sampled between silos, everything should work out fine, since that would move Stochastic Gradient Descent to the edge case where it implements Batch Gradient descent. But how about samples (mini-batches) of size 5? or 10? or 20? Here, we will measure the tradeoffs between silo count and mini-batch size, assuming an equal amount of data per silo. This code can be structured in a few ways in a distributed environment. We'll consider a peer-to-peer example first. A simulator for this "Serial Silo" SGD (SSSGD) approach is shown in figure SSSGD. Here's how it works:

Let's assume 32 pieces of training data and mini-batches of length 4 with 4 silos.

training_data [0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31]

Assume also that these data are evenly divided amongst 4 different silos.

Silo ID -	Silo Content
0	[0 1 2 3 4 5 6 7]
1	[8 9 10 11 12 13 14 15]
2	[16 17 18 19 20 21 22 23]
3	[24 25 26 27 28 29 30 31]

We then start processing the first epoch by randomizing the content in each silo.

Silo ID -	Silo Content
0	[0 7 3 4 2 1 6 5]
1	[12 10 11 14 13 9 15 8]
2	[21 18 17 19 23 22 16 20]
3	[26 30 31 29 28 24 25 27]

Then break each silo's contents into mini-batches of length 4.

Silo ID -	Silo Content
0	[0, 7, 3, 4], [2, 1, 6, 5]
1	[12, 10, 11, 14], [13, 9, 15, 8]
2	[21, 18, 17, 19], [23, 22, 16, 20]
3	[26, 30, 31, 29], [28, 24, 25, 27]

In a peer-to-peer architecture, silo0 processes its mini-batches, updates an initialized model and sends the result to silo1. Silo1 updates the model with its mini-batches and sends the result to silo2, which updates the model with its mini-batches and sends it to silo3, which sends its updated model back to silo0. Silo0 then evaluates the model against the test data. If it passes the accuracy criteria or epoch limit, the cycle stops. Otherwise silo0 re-randomizes its content and the cycle continues.

Note that for any particular epoch, since gradients are summed into the model's parameters, order of mini-batch processing does not matter. We could sum Silo2's second mini batch with Silo1's first mini batch and all the rest in any order. But we choose to let each silo sum the gradients resulting from their respective mini batches locally and avoid the overhead of transmitting intermediate results.

This makes communication O(silo count x epochs).

Algorithm 2: Serial Silo SGD (SSSGD)

```
def SSSGD(self, training_data, epochs,
          mini_batch_size, eta, silo_count,
          test_data=None):
  if test_data: n_test = len(test_data)
  n = len(training_data)
  random.shuffle(training_data)
  silo_size=n/silo_count
  silos = []
  for k in xrange(0, n, silo_size):
    silos.append(training_data[k:k+silo_size])

  for j in xrange(epochs):

    mini_batches = []
    for silo in silos:
      random.shuffle(silo)
      for k in xrange(0,len(silo),
                      mini_batch_size)]:
        mini_batches.append(
                      silo[k:k+mini_batch_size])

      for mini_batch in mini_batches:
        self.update_mini_batch(mini_batch, eta)
      if test_data:
        print "Epoch {0}: {1} / {2}".format(
         j, self.evaluate(test_data), n_test)
      else:
        print "Epoch {0} complete".format(j)
```

3.4. Comparing NSGD and SSSGD

Figures 2 illustrates results for NSGD running on a mini-batch size of 5 for 50 trials, 30 epochs and 1 silo. Figure 3 uses identical min-batch, trial and epoch size but runs SSSGD on 2 silos. The curve clusters show expected behavior. They start out in the vicinity of 90% accuracy after the first epoch and converge to about 95% accuracy after 30 epochs. Note that one of the 50 NSGD trials and one of the 50 SSSGD trials converged at about 84%. Note also that two of the SSSGD trials converged after 4 iterations.

This is summarized in the "crossovers" column in "Table 1: NSGD and SSSGD Comparisons", which records the count of trials which have not converged above 90% for 2,4,8,16 and 29 epochs respectively. For example, as can be seen in the table, NSGD/5MBS (MBS=mini-batch size) had 3 trials that did not converge after 2 and 4 epochs, 2 trials that did not converge after 8 epochs and 1 trial that did not converge after 16 and 29 epochs. The table's First Mean column is the mean of all epoch 1 trials. The Last Mean column is the mean of all epoch 30 trials. The Last Variance column is the variance of all epoch 30 trials.

We plot First Mean and Last Mean versus Silo Count in Figure 4.

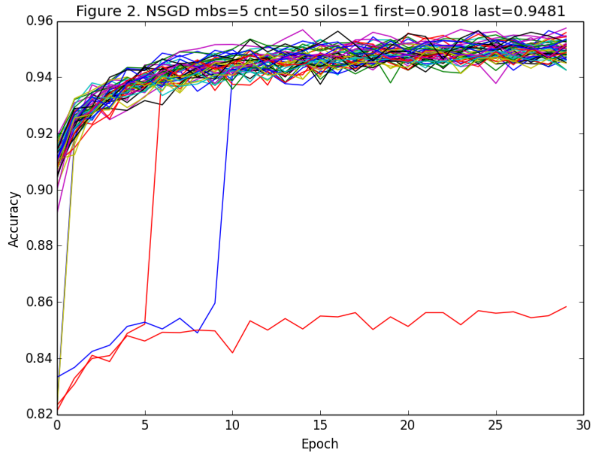
Figure 2. NSGD mbs=5 cnt=50 silos=1 first=0.9018 last=0.9481


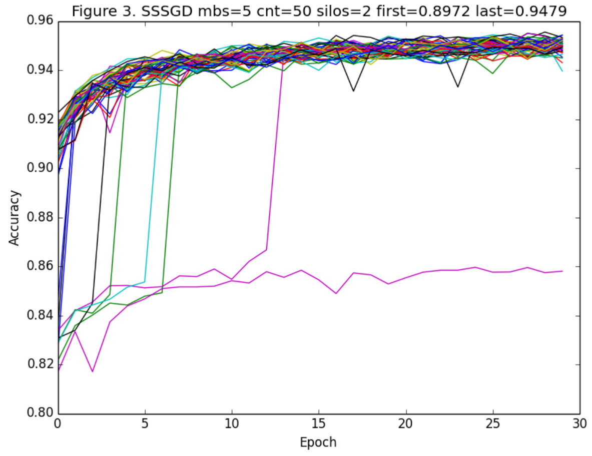
Figure 3. SSSGD mbs=5 cnt=50 silos=2 first=0.8972 last=0.9479


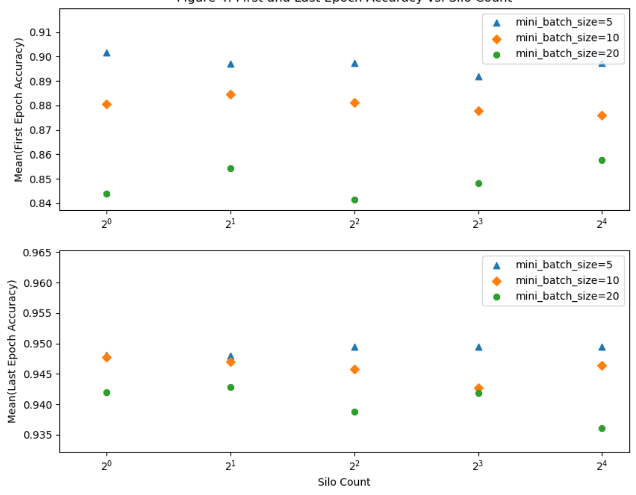
Figure 4: First and Last Epoch Accuracy vs. Silo Count


Table 1: NSGD and SSSGD Comparisons

Name	MBS	Cross overs 2,4,8,16,29	Silo Cnt	First Mean	Last Mean	Last Variance
NSGD	5	3, 3, 2, 1, 1	1	0.9018	0.9481	0.0003006
NSGD	10	6,6,5,3,1	1	0.8805	0.9477	0.0002901
NSGD	20	12,8,6,5,3	1	0.8439	0.9420	0.0007900
SSSGD	5	6, 4, 2, 1, 1	2	0.8972	0.9479	0.0002975
SSSGD	10	7, 4, 2, 1, 1	2	0.8846	0.9470	0.0003888
SSSGD	20	8, 7, 6, 1, 1	2	0.8544	0.9428	0.0011728
SSSGD	5	3, 2, 1, 1, 0	4	0.8973	0.9494	0.0000160
SSSGD	10	8, 5, 3, 2, 2	4	0.8812	0.9457	0.0005974
SSSGD	20	10,10,9,8,4	4	0.8416	0.9388	0.0010669
SSSGD	5	4, 4, 2, 1, 0	8	0.8919	0.9495	0.0000103
SSSGD	10	8, 5, 3, 3, 3	8	0.8777	0.9427	0.0009329
SSSGD	20	11,7,5,3,2	8	0.8480	0.9419	0.0007518
SSSGD	5	5, 4, 4, 1, 0	16	0.8974	0.9494	0.0000067
SSSGD	10	8, 7, 3, 2, 1	16	0.8759	0.9464	0.0002443
SSSGD	20	11, 8, 7, 5, 5	16	0.8575	0.9360	0.0014022

Figure 4 shows that as the mini-batch size increases, Accuracy falls and it takes longer to cross over the 90% threshold. This also accounts for the increased variance of the final epoch's accuracy. For mini_batches of 5, both "First Mean" and "Last Mean" seem largely unaffected by silo count.

3.5. Parallel Silo Randomization SGD

An alternate architecture would be to have a central server that sends the model to each silo. Silos would then return gradients to the central server. The server would sum the gradients together and evaluate the model against the test data. The server would continue the cycle until the desired level of accuracy was obtained or the epoch limit was reached. There are several positive tradeoffs to counter the added overhead disadvantage of central processing. One is that using a central server could be O(silo_count) faster, since it allows each silo to work in parallel. Another is that conceptually, any client could hold up processing in the serial silo scenario. A central server can more easily recover from the loss or processing delay of a single client.

4. CONCLUSIONS

Using "Federated Learning" to create "Deep Learning" models while preserving the privacy of their distributed, siloed datasets seems to work for the MNIST dataset across 16 silos. As such, it seems that our statistical assumptions (iid) are valid in this case.

However, this does raise the question, why wasn't forgetting encountered in prior work [20170217]? The answer, we believe, relates to the relatively large number of client nodes in their case. The authors actually ignore a subset of their clients since "experiments show diminishing returns for adding more clients beyond a certain point". So although, as they say, "any particular dataset will not be representative of the population distribution", random mini-batches *across collections of clients* will be. As such, statistically, their random mini-batches of clients probably correspond to our clients' random mini-batches.

5. FUTURE WORK

The next step is to use this paper to convince some silos to try it on real data. At that point, "plumbing" issues must be addressed. For example,

- **Meta Data** - In a peer-to-peer network, a client must know the order of processing so that it can send the data to the next client. What is the acknowledgement protocol? When does a "timeout" occur?, etc.
- **Standard Formats** - E.g. The interchange format contains layers of weights. What format and precision should be used to transmit the data? E.g. Is 100% transmitted as 100.000 or 1.0000?
- **Benchmark Data** - E.g. Published results in the literature generally lack benchmark data. This is necessary if we want to "calibrate" two different neural networks running in two different installations to see if they produce the "same" result.

In addition, the protocol must take into account the fact that data and silos will be added over time.

Future experiments might look into splitting test (validation) data as well as trial data.

We need to look into the possibility of adversarial attacks which could reverse compile patient data from the DL model [20170914].

Finally, in the real world, there are a variety of reasons that the institutions behind these silos do not co-operate (beyond HIPAA). They include inertia, "Not Invented Here" and competitive considerations. A future paper will describe a framework for addressing this.

Acknowledgments

The author wishes to thank Mike D'Amour and Felix Frayman for their comments and guidance.

References

[HIPAA] - https://www.hhs.gov/hipaa/for-individuals/guidance-materials-for-consumers/index.html

[MNIST] - http://yann.lecun.com/exdb/mnist/

[neuralnetworksanddeeplearning.com] Michael A. Nielsen, "Neural Networks and Deep Learning", Determination Press, 2015

[20170217] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson and Blaise Aguera y Arcas, Communication-efficient learning of deep networks from decentralized data, 20'th International Conference on Artificial Intelligence and Statistics (AISTATS), 2017.

[20170406] McMahan, Brendan and Ramage, Daniel, "Federated Learning: Collaborative Machine Learning without Centralized Training Data", Google AI Blog (April 6, 2017)

[20170914] B. Hitaj, G. Ateniese, and F. Pe ́rez-Cruz, "Deep Models Under the GAN: Information Leakage from Collaborative Deep Learning," in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017.

[20180424] Kaiser, Jocelyn, "A cancer drug tailored to your tumor? Experts trade barbs over 'precision oncology'", Science 364(6437) (Apr. 24, 2018) doi:10.1126/science.aat9794

[20181208] Vepakomma, P., Swedish, T., Raskar, R., Gupta, O., & Dubey, A. (2018). No Peek: A Survey of private distributed deep learning. CoRR, abs/1812.03288.

[20190311] Hao, Karen, "A little-known AI method can train on your health data without threatening your privacy", Technology Review (March 11, 2019)