The University of Melbourne


Research Project: COMP90055 (25pt)

# Effects of Data Sampling on Training of Machine Learning Models

Student Name: Jiankai JIN

Student Number: 956469

Supervisor: Olya Ohrimenko

June 19, 2020

**Abstract**

This research aims to study the effects of sampling methods on the training of machine learning models, and further validate the advantages of sampling methods for the training of machine learning models for certain settings. For starters, the time and space complexity of different sampling methods are investigated to prove the practicality of sampling methods for generating batches of data for training. Then, two sampling methods, SWO and Poisson, are implemented in Tensorflow, to investigate the effects of sampling methods on the training of machine learning models, in terms of accuracy and runtime. To further validate the advantage of sampling methods, the cumulative privacy loss, over the differentially private training process, of SWO, Poisson and shuffling are compared. Moreover, a dynamic sampling approach is also designed in this research to show the advantage of sampling methods in handling the multiclass imbalance problem.

# Contents

# 1 Introduction

In the training of machine learning models, we generally use an optimizer, such as stochastic gradient descent (SGD) or Adam, to optimize parameters of the model, and accordingly minimize the loss function. When applying an optimizer, we generally feed data to it one batch after another, to be computationally tractable and scalable [5].

Traditionally, we shuffle the dataset first and then partition it into multiple batches (Figure 1), as shuffling allows each batch has a representative data distribution of the overall dataset, and accordingly gradient calculated on each batch can be a good estimate of the "true" gradient calculated on the whole dataset [18].

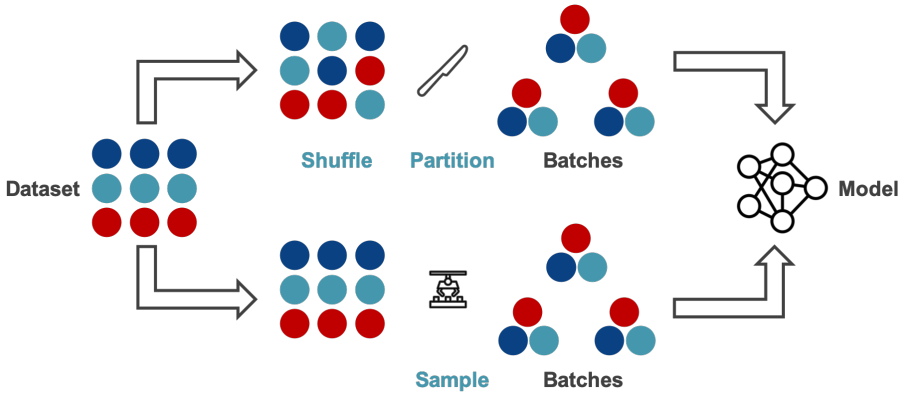


Figure 1: The Batch Generating Processes of Shuffling and Sampling

However, what has been underexplored is that sampling can also generate batches from a dataset (Figure 1).

Traditionally, sampling approaches are mainly used independently of the training processes of machine learning models. For example, bagging uses sampling with replacement to generate multiple sub-datasets for multiple sub-models of the ensemble [17]. In this way, sampling does influence the performance of the final machine learning model, but does not influence the training process directly.

Therefore, in this research, we will investigate the sampling approaches in the training of machine learning models, by mainly focusing on:

- *Whether sampling is an appropriate approach to feed data for the training of machine learning models?*

- *What advantages sampling approaches have over the traditional shuffling approach?*

## 2 Related Literature

It is shown in [6], when sampling is done independently and uniformly at random, batches will have similar data distribution of the overall dataset, just like batches generated by shuffling. Sasy and Ohrimenko also experimented and gained the result that training with shuffling and sampling basically have the same accuracy for neural network models [1].

In addition to those characteristics, studies have shown that sampling approaches have some advantages over shuffling.

Olken et al. have pointed out that sampling can be used for approximating results when performing the computation on the whole dataset is expensive or unnecessary, such as approximate answers to aggregate queries in the database management system [4]. Therefore, when the dataset is too large, we can use sampling in training to get an approximate predicting model.

Sampling can also help to decrease privacy loss. Studies have shown that sampling based approaches for general differential privacy mechanisms give an order of $O(\sqrt{m/n})$ smaller $\epsilon$ than shuffling based approaches [1, 2, 10, 14]. With smaller $\epsilon$, an adversary's inference from a dataset is more likely to be similar with or without the presence of an individual, and accordingly individual privacy is better protected [2].

Sampling can also deal with the multiclass imbalance problem, which means the trained model has bias against the minority classes, when the whole training dataset is multiclass imbalanced. Lin et al. have used a dynamic sampling approach over a 20-class imbalanced dataset in the training process, and the final model's performance over all classes is quite balanced, and accordingly it gains an overall good performance [3].

Different data in a dataset have different levels of importance for the training of machine learning models. Katharopoulos el al. proposed an importance sampling algorithm implemented in the training process, which takes advantage of the upper bound to the gradient norm to rank the importance of data, and sample data with higher importance to speed up the training of neural network models [7]. Kabkab et al. implemented a sampling strategy based on class balance, diversity, representativeness, and classifier uncertainty to generate batches in the training process, and at last trained a neural network model with less data, faster speed and no sacrifices in performance [8].

All those studies have proven the values of sampling methods in the training of machine learning models, in terms of training time, final model performance, etc.

In this research, we will investigate more on the effects of sampling methods on the training of machine learning models, such as effects on accuracy, privacy, and so on.

# 3 Research Method

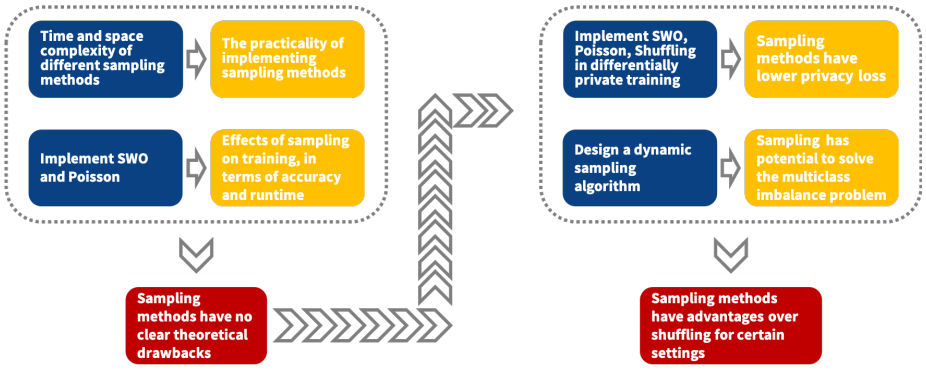


Figure 2: Research Method

The research method follows the steps in Figure 2, which shows how our main contributions, including four findings (yellow boxes) and two conclusions (red boxes), are gained.

First, we studied the time and space complexity of different sampling methods in the fourth section, and implemented SWO and Poisson sampling methods in the fifth section. Through those two studies, we proved the practicality of implementing sampling methods, and learned that sampling methods do compromise the accuracy of models a bit, but not much.

Then we implemented SWO, Poisson, and shuffling in differentially private training in the sixth section, and designed a dynamic sampling algorithm in the seventh section. Through those two further studies, we proved that sampling methods have lower privacy loss, and have the potential to alleviate the multiclass imbalance problem.

Based on the first two findings, we concluded that sampling methods have no clear theoretical drawbacks, compared with shuffling, in the training of machine learning models. Based on the last two findings, we concluded that sampling

Table 1: Time And Space Complexity Of Different Sampling Algorithm

| Complexity\Method | Shuffling | Oblivious SWO/Poisson | Dynamic Sampling | Importance Sampling |
|---|---|---|---|---|
| Time | $O(n)$ | $O(an)$ | $O(cn)$ | $O(n)$ |
| Space | $O(1)$ | $O(\sqrt[a]{n})$ | $O(cn)$ | $O(1)$ |

methods have advantages over shuffling for certain settings, in the training of machine learning models. Those two conclusions properly answered the two research questions we come up with in the beginning of the report.

# 4 Efficiency Investigation of Different Sampling Methods

Before implementing sampling methods and validating the advantages of sampling methods on the training of machine learning models, it's best to investigate their time and space complexity (per epoch) first, thus their efficiency at least will not be of any concerns. After all, without efficiency, implementing sampling methods in the training of machine learning models would be pointless, given the considerable large training datasets commonly used in practice.

To more conveniently discuss the efficiency of different sampling methods, several notations are introduced here:

- $c$: the number of classes in the training dataset.
- $k$: the number of batches.
- $a$: a small constant.
- $n$: the dataset size.
- $m$: the batch size.

For traditional shuffling algorithm, it just permutates the dataset, and then partitions the dataset into batches. As the permutation only takes linear time, the time complexity of shuffling is $O(n)$. Because permutation is done inplace, the space complexity of shuffling is $O(1)$, that is, only a constant number of working space is needed.

In [1], Sasy and Ohrimenko implements an oblivious sampling without replacement algorithm and an oblivious Poisson sampling algorithm, they use a clever grouping methodology to lower their time and space complexity to $O(an)$ and $O(\sqrt[a]{n})$ respectively.

In [3], a dynamic sampling method is implemented to alleviate the multiclass imbalance problem. This method first duplicates the examples of the minority classes, and extend the number of examples in the dataset to $cn$ at worst. Then, the method evaluates each example, and decides whether it should be used to update the model, each evaluation takes O(1), thus the method takes O($cn$) time at worst. The duplication that lets all classes have basically same number of examples makes the method take O($cn$) space at worst.

In [7], an importance sampling method is implemented to speed up the training process.

Importance sampling has two subsequent sampling in total. The first sampling returns a uniformly sampled large batch of $B$ examples, and takes O($n$) time. The second sampling uses the data distribution of the first batch to generate a smaller batch of $b$ examples, and uses this smaller batch to update the model, it takes O(1) time, depending on $B$. Therefore, the time complexity of importance sampling is O($n$).

Importance sampling undertakes a uniform sampling to generate a batch of size $B$, and a further sampling based on the distribution of the former batch to generate a smaller batch of size $b$. The final small batch of size $b$ is used to update the model. The largest working space taken is O($B$), which is irrelevant with dataset size $n$, thus the space complexity is O(1).

Dynamic sampling needs one forward pass through the network model for each sample during sampling to decide whether it should be sampled to update the model. Therefore, although dynamic sampling has linear time complexity with regard to the dataset size, the computation is still huge. Importance sampling uses two subsequent samplings to reduce the number of examples that needs importance evaluation, and accordingly controls the time and space complexity. Those two sampling methods only generate one batch each time.

Sampling methods generally have linear time complexity, which is the same as shuffling (Table 1). However, sampling methods generally have nearly linear space complexity, which is higher than shuffling (Table 1).

Although the space complexity of sampling methods is a bit higher than shuffling, it is still linear at most, and the time complexity of sampling methods is the same as shuffling, thus it is very practical to implement sampling methods in the training of machine learning models.

# 5 Sampling Algorithm

## 5.1 Design of Sampling Algorithm

In this section, two sampling algorithms are designed and implemented in TensorFlow. The first one is Sampling Without Replacement (SWO), the second one is Poisson Sampling (Figure 3).

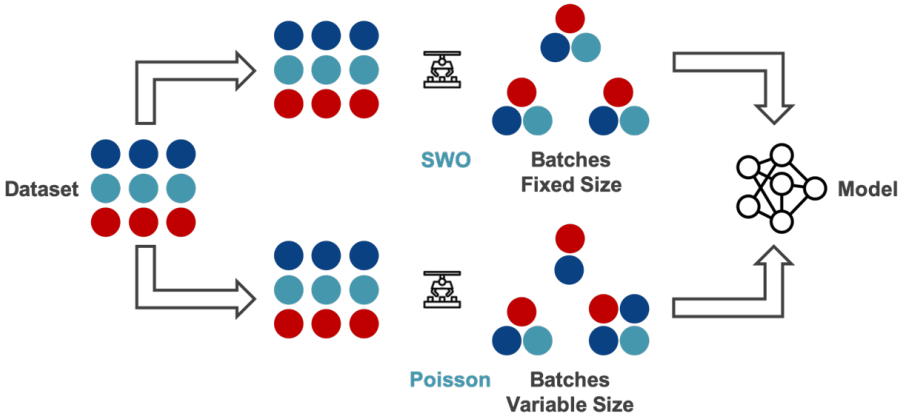


Figure 3: Illustration of SWO and Poisson

For SWO, each example is assigned an equal probability $q = batch_{size}/dataset_{size}$ to be sampled, and then the algorithm generates batches of equal size by randomly sampling from the dataset.

For Poisson sampling, each example is also assigned with an equal probability $q = b/dataset_{size}$ to be sampled. To decide whether an example should be sampled for a batch, a random number between 0 and 1 is generated, if this random number is smaller than the probability $q$, then this example is sampled for this batch. Thus, the size of each batch generated by Poisson is variable, but with expected size $dataset_{size} * q$.

All those two sampling methods are designed to generate multiple batches for each epoch of training. If those two sampling methods are implemented naively to loop through the dataset once to generate one batch, they will cost $O(n^2)$ time. This is too big, and accordingly a more elaborate design is implemented (Algorithm 1).

Basically, the algorithm is designed to loop through the dataset only once. The example indexes of each batch is generated first. Then the indexes of batches are grouped by the indexes of examples, that is, for each example, find all batches

containing it (Figure 4). During the one-time walkthrough of the dataset, each example is appended to all its corresponding batches.

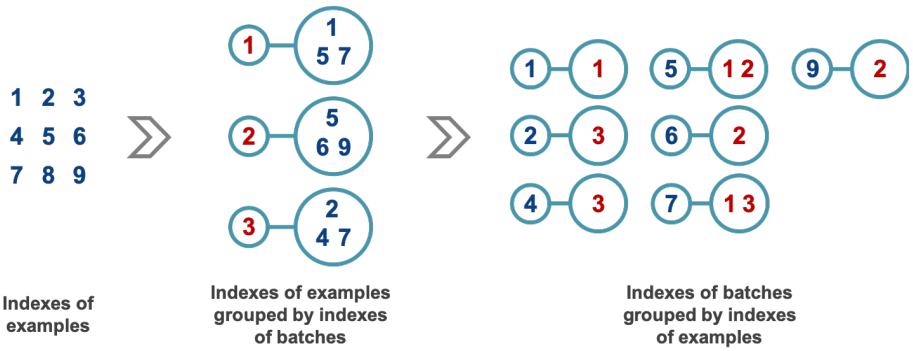


Figure 4: Sampling Algorithm Design

Algorithm 1 is the pseudocode of SWO. The only difference between SWO and Poisson is the way of generating indexes, so that there will not be another pseudocode for Poisson to avoid redundancy.

The sampling implementations of this research support tabular, image, and text data:

- For tabular data, sampling algorithms will directly sample from the dataset, and generate batches of data.
- For image data, images are resized to the fixed size 64*64*3, then sampling algorithms can sample from those fixed-size images and group them into batches.
- For text data, different texts have different length, they are padded with 0s to the same length, and then sampling algorithms can sample from those fixed-length texts and group them into batches.

## 5.2 Time and Space Complexity

There are two nested loops in the Algorithm 1.

In the first nested loops, $k * m = n$ indexes of examples will be assigned to different batches, thus the time complexity is O($n$).

In the the second nested loops of Algorithm 1, because there are exactly n

**Algorithm 1** Pseudocode Of SWO

$indexes \leftarrow UniformlySample(0, 1, 2, ..., dataset_{size} - 1)$
# generate indexes of examples of all batches, grouped by batch indexes
$indexes_{batch} \leftarrow \{\}$
$batches \leftarrow \{\}$
**for** $index_{batch} \in steps$ **do**
  **for** $index_{example} \in indexes[index_{batch}]$ **do**
    $indexes_{batch}[index_{example}].append(index_{batch})$
    # group the indexes of batches by indexes of examples
  **end for**
**end for**
**for** $example \in dataset$ **do**
  **for** $index_{batch} \in indexes_{batch}[index_{example}]$ **do**
    $batches[index_{batch}].append(example)$
    # append each example to all batches containing it
  **end for**
**end for**
return batches

Table 2: Accuracy Of Different Models Trained With Batches Generated By Different Sampling Methods And The Shuffling Method

| Method\Dataset | Cat_dog | Flowers | Malairia | Imdb | Yelp |
|---|---|---|---|---|---|
| Shuffling | 0.8262 | 0.6786 | 0.9556 | 0.8636 | 0.9275 |
| SWO | 0.8123 | 0.6457 | 0.9545 | 0.8546 | 0.9252 |
| Poisson | 0.8163 | 0.6229 | 0.9529 | 0.8242 | 0.9239 |

indexes of examples generated in the first nested loops, there are exactly n appending operations to do, thus the time complexity is O($n$).

For example, assuming there are 50,000 examples in the dataset, 500 examples in each batch, 100 batches in total. There will be 50,000 indexes of examples to be generated, and 50,000 examples to be appended to all batches in total. Therefore, there will be asymptotically O(50,000) operations.

Therefore, taking both the first and second nested loops into consideration, the total time complexity of SWO and Poisson is O($n$).

Because first nested loops generate $k * m = n$ indexes, and the second nested loops generate batches containing $k * m = n$ examples in total, the total space complexity of SWO and Poisson is also O($n$).

Table 3: Time Of Different Sampling Methods And The Shuffling Method On Different Datasets

| Method\Dataset | Cat_dog | Flowers | Malairia | Imdb | Yelp |
|---|---|---|---|---|---|
| Shuffling | 0.001 | 0.001 | 0.001 | 0.002 | 0.002 |
| SWO | 10.514 | 2.582 | 14.002 | 8.005 | 69.186 |
| Poisson | 15.386 | 2.614 | 16.633 | 7.836 | 94.863 |

## 5.3 Results

Performance tests of sampling methods have been done on 5 datasets, 3 of them are image datasets, 2 of them are text datasets. Image data are fed to the CNN model, text data are fed to the RNN model.

No matter the type of data (image or text), and the type of model (CNN or RNN), models trained with sampling methods have slightly lower accuracy, compared with models trained with shuffling (Table 2).

This slight compromise in accuracy is understandable, as the shuffling method uses the whole dataset to train the model, while sampling methods use only part of the whole dataset to train the model. After all, there are replacements between batches generated by sampling methods.

The percentage of data being sampled in the sampling process can be approximately calculated. Each batch has $m$ examples, the dataset has $n$ examples, and there are $\lfloor n/m \rfloor$ batches in total. Therefore, for an example, the probability of not being sampled is:

$$(1 - m/n)^{n/m}$$

For large n, this probability approaches $e^{-1}$. This is about 0.368. Therefore, on average there are only about 63.2% data used in the training process when sampling methods are used to generate batches.

However, sampling methods generally cost much more time to generate batches, compared with the shuffling method, although they all have linear time complexity (Table 3). With larger datasets, such as hundreds of thousands of examples, or text datasets, in which examples need to be padded to the same length, the efficiency of sampling methods gets worse.

This compromise in efficiency is because:

- Our implementations of sampling methods use a higher level API of TensorFlow, which cannot match the performance of lower level API of TensorFlow that is used by shuffling.
- Shuffling only permutates a dataset once, and then partitions it into

batches, but sampling methods are much more complex in the procedure of generating batches, thus sampling methods are inherently slower.

# 6 Implement Sampling in Differentially Private Training

The trained models should not expose the private information of the datasets which they are trained on. However, models trained in traditional ways might unintentionally expose part of the information of the datasets. For example, a model-inversion attack can recover images from a facial recognition system [9].

Therefore, in this section, we mainly focus on the differentially private training of machine learning models, which aims to train models with controlled and quantitative privacy loss. All implementations of this section are based on TensorFlow Privacy.

## 6.1 Models Trained With DP SGD

Differential privacy (DP) describes a promise, made by a data holder to a data subject: "You will not be affected, adversely or otherwise, by allowing your data to be used in any study or analysis, no matter what other studies, data sets, or information sources, are available. [2]"

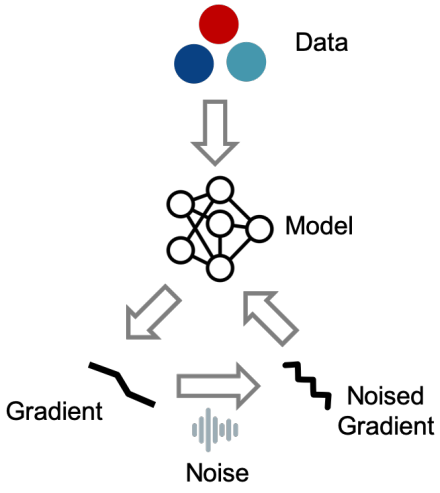


Figure 5: The Illustration of DP SGD

DP SGD is a differentially private stochastic gradient descent algorithm. It aims to train deep neural networks with non-convex objectives, under a modest privacy budget, and at a manageable cost in software complexity, training efficiency, and model quality [10].

Compared with traditional SGD, the biggest difference that DP SGD has is additive Gaussian noise. After clipping the gradients with respect to the norm bound C, DP SGD adds Gaussian noises to all gradients, and uses the average of all gradients with noises as the updating gradient (Figure 5). With those additive noises, privacy of data is better protected, and privacy loss is decreased.

The $\epsilon$ parameter is used to quantify the cumulative privacy loss over the training process. To more conveniently discuss the $\epsilon$ parameters of different sampling methods, several notations are introduced here:

- $E$: the number of epochs.
- $n$: the number of examples in a dataset.
- $m$: the number of examples in each batch.
- $k$: the number of batches per epoch.
- $T$: the number of samples, $En/m$.
- $\gamma$: the sampling ratio, $m/n$.
- $\delta$: the probability that plain $\epsilon$-differential privacy is broken.
- $\epsilon_{subsample}$: the $\epsilon$ parameter over a subsample.
- $\epsilon_{epoch}$: the $\epsilon$ parameter over an epoch.
- $\epsilon$': the $\epsilon$ parameter over a sequence of DP mechanisms.

Different batching methods lead to different privacy loss, and their $\epsilon$ parameters are calculated in different ways [16].

For Poisson, the $\epsilon_{subsample}$ parameter over a sample is calculated based on [14], which gives a much tighter bound on the RDP parameter for a subsampled mechanism. The $\epsilon_{subsample}$ is $log(1 + \gamma(e^{\epsilon} - 1))$. For a sequence of Poisson sampling, the cumulative $\epsilon$' parameter is calculated based on the moments accountant method [10], which gives a much tighter bound for the estimation of the privacy loss than strong composition.

For SWO, its $\epsilon_{subsample}$ parameter over a sample is also calculated based on [14]. The $\epsilon_{subsample}$ is $log(1 + m/n(e^{\epsilon} - 1))$. For a sequence of SWO, the cumulative $\epsilon$' parameter is calculated based on the strong composition theorem [2].

Table 4: The $\epsilon$ Parameters of Different Sampling Methods in Big-O Notation [1]

| Methods | ($\epsilon$', $\delta$') |
|---|---|
| Shuffling | $O(\epsilon\sqrt{Elog(1/\delta'')}, E\delta + \delta'')$ |
| SWO | $O(\epsilon\gamma\sqrt{Tlog(1/\delta'')}, T\gamma\delta + \delta'')$ |
| Poisson | $O(\gamma\epsilon\sqrt{T}), \delta$ |

Table 5: Accuracy and $\epsilon$ Values Of Models Trained With SGD and DP SGD on Different Datasets

| Method\Dataset | | Cat_versus_dog | Flower | Malaria |
|---|---|---|---|---|
| Shuffling | SGD acc | 0.8262 | 0.6786 | 0.9556 |
| | DP SGD acc | 0.7823 | 0.3885 | 0.8031 |
| | $\epsilon$' | 5687.38 | 5687.38 | 5687.38 |
| SWO | SGD acc | 0.8123 | 0.6471 | 0.9545 |
| | DP SGD acc | 0.7654 | 0.4974 | 0.8656 |
| | $\epsilon$' | 403.13 | 1419.99 | 353.24 |
| Poisson | SGD acc | 0.8163 | 0.6229 | 0.9529 |
| | DP SGD acc | 0.7589 | 0.4437 | 0.8634 |
| | $\epsilon$' | 1.20 | 2.78 | 1.12 |

For shuffling, in a single epoch, McSherry [15] states that running k algorithms in parallel on disjoint samples of the dataset has $\epsilon_{epoch} = max_{i\in[1,k]}\epsilon_i$, where $\epsilon_i$ is the parameter of the $i^{th}$ mechanism. For multiple epochs, the cumulative $\epsilon$' parameter is calculated based on the strong composition theorem [2].

The asymptotic estimation of the $\epsilon$' parameters over a sequence of DP mechanisms is summarized in Table 4. All those results are based on one prerequisite, the sample is hidden during the analysis, that is, the keys of the elements of a sample are hidden [1].

## 6.2 Results

Three datasets of images, Cat_Versus_Dog, Flower, and Malaria, are used to evaluate shuffling, SWO, and Poisson, when models are trained with DP SGD and SGD, in terms of accuracy and $\epsilon$'.

As can be seen from Table 5, models trained with different sampling methods have different $\epsilon$' values, the rank is: $Poisson < SWO < Shuffle$.

The different levels of privacy loss are caused by the different levels of uncertainty introduced by different sampling methods [16]. Poisson introduces most uncertainty in the process of generating batches, thus its privacy loss is the lowest. Shuffling introduces least uncertainty when generating batches, and

accordingly has the highest privacy loss.

In term of performance, accuracy of models trained with DP SGD is lower than accuracy of models trained with SGD. There are mainly two reasons that the accuracy of models trained with DP SGD is degraded:

1. gradients have to be clipped with respect to the norm bound C, thus the clipped gradients will bring less changes to the models than the original gradients.
2. noises are added into the gradients after clipping, this would cause the new gradients point to different direction than the original gradients.

There is also a subtlety that is worth attention. The accuracy difference between models trained with DP SGD and SGD is much larger for the Flower dataset. This phenomenon might be caused by the size of the dataset [10]. The Cat_Versus_Dog and Malaria training datasets have 18610 and 22046 examples respectively, in contrast, the Flower training dataset only has 2936 examples.

With larger dataset, the model can be updated with more gradients. Although each gradient is compromised a bit by the Gaussian noise, more gradients should neutralize those noises statistically, and accordingly the trained models can gain better performance.

Unfortunately, DP SGD of TensorFlow Privacy does not support RNN right now, so the "IMDB" and "YELP" datasets are not included in the results of this section. However, currently there are papers discussing differentially private recurrent language models [11], it is believable that TensorFlow Privacy will also support RNN in the near future.

# 7 Multiclass Imbalance Problem

Imbalanced dataset is the sort of dataset in which the number of examples in each class is quite uneven, such as 90% positive and 10% negative for a 2-class dataset.

In an imbalanced dataset, the class that has most examples is the majority class, all other class are called the minority classes.

When we try to do classification on this sort of dataset, we will have a multiclass imbalance problem, which means the model trained on this dataset will overfit to the majority class. This is because training in essence is using examples to generate gradients, and use gradients to update the model, thus the model is

trained (updated) more by examples of the majority class, and accordingly the model will overfit to the majority class.

However, the model we need is the one that can equally classify each class. In this section, sampling methods will be studied and implemented to try to train models that can more equally classify each class, and accordingly alleviate the multiclass imbalance problem.

Because this research mainly focuses on the effects of sampling methods on the training of machine learning models, other methods that can also alleviate multiclass imbalance problem, such as data synchronization and model-oriented methods which manipulate the loss of examples in the minority classes [12], will not be explored.

## 7.1 Random Oversampling

The baseline method to handle the multiclass imbalance problem is random oversampling (Algorithm 2), that is, duplicating the examples of the minority classes to make them have the same number of examples as the majority class.

Because of duplication, the model will be trained same times by the examples of the minority classes as the the majority class. Therefore, hopefully the model might learn better to classify the minority classes. However, this might not work well at last, after all, the number of different training examples is not changed, and thus the overall information provided for the minority classes is not changed as well.

**Algorithm 2** Pseudocode Of Random Oversampling

$dataset_{class} \leftarrow dataset$
# group examples in a dataset by their classes
$num_{class} \leftarrow dataset_{class}$
# count the number of examples in each class
$num_{max} \leftarrow max(num_{class})$
**for** $c \in classes$ **do**
  $dataset_{class}[c] \leftarrow Duplicate(dataset_{class}[c], num_{max}/num_{class}[c] - 1)$
  # duplicate examples in class c by $num_{max}/num_{class}[c] - 1$ times
**end for**
**for** $d \in dataset_{class}$ **do**
  $dataset_{final}.append(d)$
**end for**
$shuffle(dataset_{final})$
Return $dataset_{final}$

## 7.2 Dynamic Sampling

In fact, as a baseline method, random oversampling performs really badly. There are two reasons explaining its unsatisfactory performance:

1. It might cause severe overfitting problem. For example, for a credit card dataset, 99.9% of data might be negative (normal), only 0.1% of data are positive (abnormal). The duplication process will duplicate the abnormal data 998 times. This amount of duplicated data will definitely cause severe overfitting problem. That is, the model cannot generalize well on the examples of minority classes, can only perform well on examples of minority classes in the training dataset.

2. The random oversampling algorithm in essence simply blindly samples data from the minority classes more times. Some sampled data in the minority classes are already correctly classified by the model, yet some unsampled data in the majority class are still misclassified by the model. Therefore, maybe some examples in the minority classes should be ignored in the later training process, some examples in the majority class should be sampled more times.

To gain a better performance on the multiclass imbalanced dataset, a dynamic sampling method is designed in this research (Figure 6), its specific details can be seen in Algorithm 3.

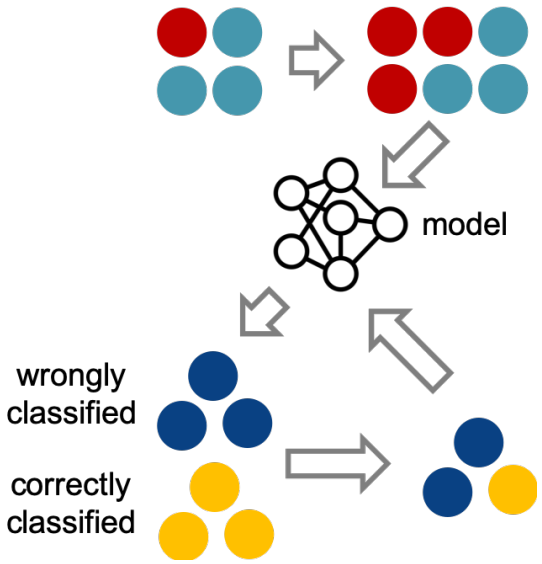


Figure 6: The Illustration of Dynamic Sampling

**Algorithm 3** Pseudocode Of Dynamic Sampling

$dataset_{class} \leftarrow dataset$
# group examples in a dataset by their classes
$num_{class} \leftarrow dataset_{class}$
# count the number of examples in each class
$num_{max} \leftarrow max(num_{class})$
$\alpha \leftarrow 0.7$
# the ratio of data in each batch to be sampled from wrongly classified data
**for** $c \in classes$ **do**
  $dataset_{class}[c] \leftarrow Duplicate(dataset_{class}[c], \lfloor\sqrt{num_{max}/num_{class}[c]}\rfloor - 1)$
  # duplicate examples in class c by $\lfloor\sqrt{num_{max}/num_{class}[c]}\rfloor - 1$ times
**end for**
**for** $d \in dataset_{class}$ **do**
  $dataset_{training}.append(d)$
**end for**
$shuffle(dataset_{training})$
**for** $epoch \in epochs$ **do**
  **if** $epoch = 1$ **then**
    $batches = sample(dataset_{training}, batch_{size}, batch_{num})$
  **else**
    $batches_{correct} = sample(dataset_{correct}, (1-\alpha) * batch_{size}, batch_{num})$
    # for each batch, sample 30% data from correctly classified data
    $batches_{wrong} = sample(dataset_{wrong}, \alpha * batch_{size}, batch_{num})$
    # for each batch, sample 70% data from wrongly classified data
    **for** $c \in batches_{correct}$, $m \in batches_{wrong}$ **do**
      batches.append(concatenate(c,m))
    **end for**
  **end if**
  $model.train(batches)$
  $labels_{predicted} \leftarrow model.predict(dataset_{training})$
  **for** $example \in dataset_{training}$ **do**
    **if** $labels_{predicted}[example] = labels_{real}[example]$ **then**
      $dataset_{correct}.append(example)$
    **else**
      $dataset_{wrong}.append(example)$
    **end if**
  **end for**
**end for**

To handle the severe overfitting problem, the duplication times of the minority classes has to be constrained. For example, for the aforementioned credit card dataset, we cannot simply copy the examples in positive class 998 times.

In dynamic sampling, the duplication times is set to be $\lfloor\sqrt{num_{max}/num_c}\rfloor - 1$, that is, the square root of the number of examples in the majority class divided by the number of examples in the minority class, and minus 1. For the aforementioned credit card dataset, examples in the minority class only need to be duplicated 30 times this time.

By setting this duplication times formula, if the ratio between the number of examples in the majority class and the minority class is small, such as 2, the duplication times is $\lfloor\sqrt{2}\rfloor - 1 = 0$, thus there is no need to duplicate. If the ratio is very large, such as 1000, the duplication times is $\lfloor\sqrt{1000}\rfloor - 1 = 30$, thus the duplication is done to have more training examples of the minority class, but not too many. Therefore, the model can have better generality and classification capability over the examples of the minority classes.

To let the model focus on the data that it performs poorly on, a boosting mechanism is integrated into the sampling process. After the first epoch, data of each batch are partly sampled from misclassified data and partly sampled from correctly classified data. The percentage of data of each batch sampled from misclassified data is set as the hyperparameter $\alpha$.

This mechanism is inspired by the GOSS algorithm of LighhtGBM, which uses gradient to evaluate the contribution of each example, and then partly sample examples with large contribution, and partly sample examples with small contribution [13].

From the second epoch, for each batch, $\alpha * batch_{size}$ examples will be sampled from data that are wrongly classified in the previous epoch, and $(1-\alpha)*batch_{size}$ examples will be sampled from data that are correctly classified in the previous epoch. Main formulas used include:

- $batches_{correct} = sample(dataset_{correct}, (1-\alpha) * batch_{size}, batch_{num})$
- $batches_{wrong} = sample(dataset_{wrong}, \alpha * batch_{size}, batch_{num})$
- $concatenate(c, m)\ for\ c \in batches_{correct}\ and\ m \in batches_{wrong}$

The $\alpha$ parameter is set as 0.7 in this research. Therefore, models will be trained more with the examples that they perform poorly on, and at the mean time will not forget what they have already learned by reviewing data they have correctly classified.

In essence, this sampling method is a biased sampling, which biases toward the examples the model performs poorly on, such as examples in the minority

Table 6: 2-Class Imbalanced Datasets

| Dataset | Size | Negative Class | Positive Class |
|---|---|---|---|
| Credit Card | 284807 | 99.83% | 0.17% |
| Thyroid | 7200 | 7.42% | 92.58% |
| Pulsar | 17898 | 90.84% | 9.16% |

Table 7: Accuracy and AUC of Different Sampling Methods on Different 2-Class Datasets

| Method\Dataset | | Credit Card | Thyroid | Plusar |
|---|---|---|---|---|
| Shuffling | acc | 0.9989 | 0.9245 | 0.96811 |
| | auc | 0.9376 | 0.7716 | 0.9400 |
| Random Oversampling | acc | 0.3457 | 0.8085 | 0.9452 |
| | auc | 0.9414 | 0.8424 | 0.9728 |
| Dynamic Sampling | acc | 0.9908 | 0.9291 | 0.9671 |
| | auc | 0.9744 | 0.8548 | 0.9718 |

classes, and examples that are wrongly classified in the previous epoch. It can also be seen as an importance sampling, that assigns examples the model performs poorly on with greater importance.

By integrating boost and oversampling into the sampling process, models are trained more with examples in the minority classes and examples that are currently hard to classify, and hopefully get better performance.

## 7.3 Results

### 7.3.1 Results on 2-Class Imbalanced Datasets

For starters, sampling methods are tested on the imbalanced datasets of 2 classes, including "credit card", "thyroid", and "pulsar" (Table 6). The reason to use imbalanced datasets of 2 classes as starters is that it is easy to find metrics to fairly evaluate the model's overall performance.

Two metrics, accuracy and AUC, are combined to evaluate the model's performance. Accuracy is a metric widely used in the evaluation of basically all models. However, it is a biased metric when used on imbalanced dataset. Therefore, AUC is also used in this section to try to fairly evaluate the model's classification capability on the imbalanced dataset, taking both majority and minority classes into consideration.

In term of accuracy (Table 7), the performance rank is:

$$dynamic\,sampling \approx shuffling > random\,oversampling$$

Table 8: Multi-Class Imbalanced Datasets

| Dataset | Size | Class Distribution |
|---|---|---|
| White Wine Quality | 4898 | [20,163,1457,2198,880,175,5] |
| Page | 5473 | [4913,329,28,88,115] |
| Thyroid | 2800 | [1632,91,275,31,771] |
| Adalone | 4124 | [57,115,259,391,568,689,634,487,267, 203,126,103,67,58,42,32,26] |

In term of AUC (Table 7), the performance rank is:

$$dynamic\,sampling \approx random\,oversampling > shuffling$$

Based on those two ranks, we can conclude that dynamic sampling gains the best performance on 2-class imbalanced datasets. Dynamic sampling retains the accuracy of the model, and at the meantime enhances the model's performance on the minority class. In contrast, although random oversampling has increased the model's performance on the minority class, it also sacrifices the model's accuracy, by compromising the model's performance on the majority class. As expected, shuffling cannot ensure the model's performance on the minority class, and accordingly gets lower values on AUC.

#### 7.3.2 Results on Multi-Class Imbalanced Datasets

In this section, sampling methods are tested on multiclass imbalanced datasets, including "white wine quality", "page", "thyroid", and "adalone" (Table 8).

The metrics used in this section to fairly evaluate the model's performance on multiclass imbalanced datasets include: accuracy, min recall, median recall, and max recall.

Min recall, median recall, and max recall are the minimum, median, maximum of recalls of all classes respectively. To get values of those different recalls, the recall of each class needs to be calculated, based on "one versus the rest" mechanism.

The min and median recalls reflect the model's performance on the minority classes, the max recall reflects the model's performance on the majority class. Accuracy is a metric that reflects the model's performance over all classes, but with bias towards the majority class, thus it is not a fair evaluation metric.

In term of accuracy (Table 9), the performance rank is:

$$shuffling \approx dynamic\,sampling > random\,oversampling$$

Table 9: Accuracy, Min Recall, Median Recall, Max Recall of Different Sampling Methods on Different Multi-Class Imbalanced Datasets

| Method\Dataset | | White Wine | Page | Thyroid | Adalone |
|---|---|---|---|---|---|
| Shuffling | acc | 0.5437 | 0.9370 | 0.6830 | 0.2441 |
| | min recall | 0.0000 | 0.0000 | 0.0000 | 0.0000 |
| | median recall | 0.0000 | 0.0069 | 0.0462 | 0.0000 |
| | max recall | 0.6700 | 0.9410 | 0.6639 | 0.43196 |
| Random Oversampling | acc | 0.2933 | 0.8804 | 0.5740 | 0.2015 |
| | min recall | 0.0000 | 0.2002 | 0.1284 | 0.0004 |
| | median recall | 0.1650 | 0.2238 | 0.2351 | 0.0439 |
| | max recall | 0.2656 | 0.3120 | 0.2481 | 0.2976 |
| Dynamic Sampling | acc | 0.5333 | 0.9527 | 0.6919 | 0.2353 |
| | min recall | 0.0000 | 0.0327 | 0.0077 | 0.0000 |
| | median recall | 0.0387 | 0.0803 | 0.0828 | 0.0295 |
| | max recall | 0.6651 | 0.6533 | 0.6341 | 0.3054 |

In term of min recall (Table 9), the performance rank is:

$$random\,oversampling > dynamic\,sampling > shuffling$$

In term of median recall (Table 9), the performance rank is:

$$random\,oversampling > dynamic\,sampling > shuffling$$

In term of max recall (Table 9), the performance rank is:

$$shuffling > dynamic\,sampling > random\,oversampling$$

As can be seen from the above four ranks, dynamic sampling retains the model's performance on the majority class, and at the meantime increases the model's performance on the minority classes.

In contrast, although random oversampling increases the model's performance on minority classes, it also greatly compromises the model's performance on the majority class, and accordingly compromises the model's overall accuracy. Shuffling, as usual, performs well on the majority class, but ignores the minority classes, thus its overall performance is unsatisfactory.

By focusing more on examples that are wrongly classified in the training process, and constraining the duplication times of the minority classes, dynamic sampling does help models to better learn how to do better overall classification on multiclass imbalanced dataset.

# 8 Discussion

This research is conducted to investigate the effects of sampling methods on the training of machine learning models, and then further prove the advantages of sampling methods on the training of machine learning models over the traditional shuffling method, for certain settings.

At first, the efficiency of different sampling methods is investigated to validate the practicality of implementing sampling methods to generate batches for the training of machine learning models. Results show that most sampling methods are linear in time complexity, have less than or equal to linear space complexity. Therefore, it is practical to implement sampling methods for the training of machine learning models.

Two sampling methods, SWO and Poisson, are then implemented and studied. Results show that the accuracy of models trained with sampling methods is a bit lower than the accuracy of models trained with shuffling. This is because sampling methods only feed models with about 63.2% data of the whole training dataset. Results also show that our sampling implementations take more time to generate batches, especially when the dataset is very large. This is partly because our programming implementation is not ideal, and partly because permutation used by shuffling is inherently faster than sampling.

Therefore, sampling methods have no clear theoretical drawbacks, compared with shuffling, in the training of machine learning models. With better programming implementation, although the accuracy is compromised a bit, sampling methods would still be potential choices for the training of machine learning models.

However, only not bad performance is not enough to convince us to implement sampling methods in the training of machine learning models. Thus, we conduct a further study on the privacy loss of sampling methods and shuffling. Results of models trained with DP SGD show that sampling methods (SWO and Poisson) have much lower $\epsilon$ values, and accordingly much lower privacy loss, but will cause little compromise in the accuracy of trained models, because of addictive Gaussian noise.

Moreover, sampling methods can also help to alleviate the multiclass imbalance problem. The results show that our dynamic sampling method is better than shuffling and random oversampling, by improving the model's capability on classifying examples in the minority classes, and at the mean time retaining model's capability on classifying examples in the majority class.

By showing that models trained with sampling methods have lower privacy loss, and alleviated multiclass imbalance problem, we have proven the advantages of sampling methods over shuffling for certain settings. For example, when the

training data is very private, or multiclass imbalanced.

With all those findings, there are still several problems that are currently unsolved in this research.

The first problem is the much slower running speeds of sampling methods (Table 3), compared with shuffling. Although permutation used by shuffling is faster than sampling, the running speed difference between shuffling and sampling should not be that large. This is actually caused by the high level API of TensorFlow used in our implementations of sampling methods. Specifically, the method "as_numpy_iterator()" used to loop through the whole dataset costs really a lot of time when the dataset is huge. In contrast, shuffling uses a low level API of TensorFlow that costs much less time. To make sampling methods run faster, and really practical for common use, they should also adopt the same low level API as shuffling.

The second problem is that there is no one standard for estimating the cumulative privacy loss over the training process. Moment accounts and strong composition provides different levels of tightness for the estimation of privacy loss, thus the $\epsilon$ parameters of shuffling, SWO, and Poisson, are compared under different precision levels. This might cause confusion.

The third problem is that, for dynamic sampling, whether an example should be sampled depends on whether it is correctly classified and whether it is in the minority classes. However, those two metrics might not be enough to give enough sampling bias towards the most contributing examples. Therefore, more metrics should be incorporated in the process of judging whether an example should be sampled, to more appropriately alleviate the multiclass imbalance problem. For example, the gradient of an example can also be used as a metric to judge whether this example should be sampled [13].

All those problems point out the future works could be done to let sampling methods play more important roles in the training of machine learning models.

# 9 Conclusion

Our results show that sampling methods generally have linear time complexity and less than or equal to linear space complexity, and sampling methods do compromise the accuracy of models a bit, compared with shuffling, but not much.

Moreover, under differentially private training, the privacy loss of sampling methods is much lower than the privacy loss of shuffling. When handling multiclass imbalanced datasets, sampling methods have advantages over shuffling by

feeding models with examples of greater importance.

Based on all the above findings, we concluded that sampling methods have no clear theoretical drawbacks compared with shuffling, and sampling methods are actually superior over the traditional shuffling method for certain settings. Therefore, sampling methods should be considered in the training of some models, and are worth to be studied more.

## Acknowledgements

I would like to express my special thanks of gratitude to my supervisor Doctor Olya Ohrimenko. Without her help and guidance, I would not be able to finish my research. She was very patient in answering my questions and clearing my confusions during the research. Wish her study go well, and gain steady progresses in the future.

I would also like to thank the University of Melbourne, where I spent about 2 years and learned a lot. By studying there, I gained a chance to be a recommendation algorithm engineer, and found my way for my future.

Thank you,

Jiankai JIN

# References


[1] S. Sasy and O. Ohrimenko, "Oblivious Sampling Algorithms for Private Data Analysis," in 33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada, 2019.

[2] C. Dwork and A. Roth, "The algorithmic foundations of differential privacy," Foundations and Trends in Theoretical Computer Science, vol.9, no.3, pp. 211–407, 2014.

[3] M. Lin, K. Tang and X. Yao, "Dynamic Sampling Approach to Training Neural Networks for Multiclass Imbalance Classification," IEEE Transactions on Neural Networks and Learning Systems, vol. 24, no. 4, pp. 647-660, April 2013.

[4] F. Olken, Random sampling from databases.

[5] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in Proc. 19th Int. Conf. Comput. Statist., 2010, pp. 177–186.

[6] O. Shamir, "Without-Replacement Sampling for Stochastic Gradient Methods," in 30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain, 2016.

[7] A. Katharopoulos and F. Fleuret, "Not all samples are created equal: Deep learning with importance sampling," in Proceedings of the 35th International Conference on Machine Learning, 2018.

[8] M. Kabkab, A. Alavi, and R. Chellappa. "DCNNs on a diet: Sampling strategies for reducing the training set size," CoRR, 2016.

[9] M. Fredrikson, S. Jha, and T. Ristenpart, "Model Inversion Attacks that Exploit Confidence Information and Basic Countermeasures," in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS 15, 2015.

[10] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang, "Deep learning with differential privacy," in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 308–318, 2016.

[11] H. B. McMahan, D. Ramage, K. Talwar, and L. Zhang, "Learning differentially private recurrent language models," in International Conference on Learning Representations (ICLR), 2018.

[12] L. Abdi and S. Hashemi, "To combat multi-class imbalanced problems by means of over-sampling and boosting techniques," Soft Computing, vol. 19, no. 12, pp. 3369-3385, 2014.

[13] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T. Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," Advances in Neural Information Processing Systems, pp. 3149–3157, 2017.

[14] Y. Wang, B. Balle, and S. Kasiviswanathan, "Subsampled Renyi Differential Privacy and Analytical Moments Accountant," in Artificial Intelligence and Statistics Conference (AISTATS), 2019.

[15] F. D. McSherry, "Privacy integrated queries: An extensible platform for privacy-preserving data analysis," in SIGMOD, 2009.

[16] L. Yu, L. Liu, C. Pu, M. Gursoy, and S. Truex, "Differentially private model publishing for deep learning," in IEEE Symposium on Security and Privacy, 2019.

[17] P. Bühlmann, "Bagging, Boosting and Ensemble Methods," Handbook of Computational Statistics, pp. 985–1022, 2011.

[18] M. Qi, W. Chen, Y. Wang, Z.-M. Ma, and T.-Y. Liu, "Convergence analysisof distributed stochastic gradient descent with shuffling," Neurocomputing, vol. 337, pp. 46–57, Apr. 2019.