Master's thesis

2021

Halvor Bakken Smedås

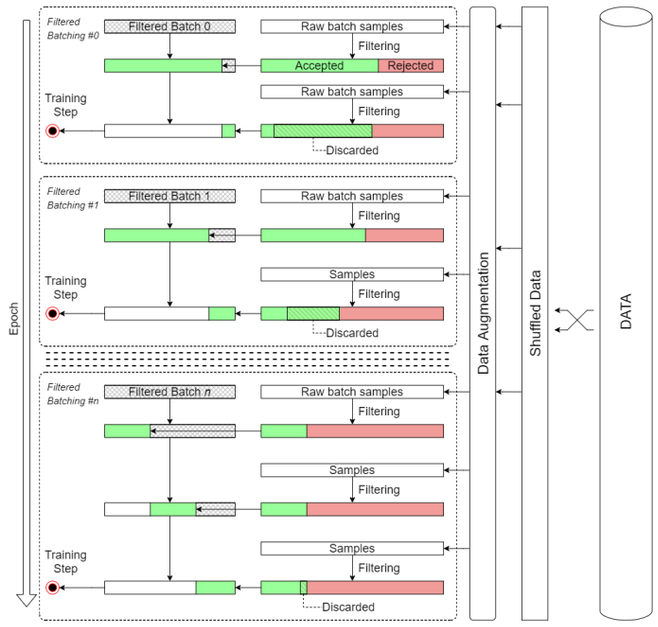
NTNU
Norwegian University of
Science and Technology
Faculty of Information Technology and Electrical
Engineering
Department of Computer Science

Halvor Bakken Smedås

ASSIST

Accuracy-driven Sampling Strategies for Improved Supervised Training

June 2021



NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

ASSIST

Accuracy-driven Sampling Strategies
for Improved Supervised Training

Halvor Bakken Smedås

Informatics
Submission date: June 2021
Supervisor: Zhirong Yang
Co-supervisor: John Reidar Mathiassen

Norwegian University of Science and Technology
Department of Computer Science

Halvor Bakken Smedås

ASSIST

Accuracy-driven Sampling Strategies for Improved Supervised Training

Master Thesis, 2020–2021
Master of Science in Informatics

Supervisors:
Zhirong Yang (IDI, NTNU)
John Reidar Mathiassen (SINTEF Ocean AS)

Department of Computer and Information Science
Faculty of Information Technology, Mathematics and Electrical Engineering

ABSTRACT

How we spend training time has become more important with neural network's evermore complex architectures. Recent research presents strategic data sampling methods as an alternative to mini-batch SGD, alleviating training of unimportant samples with little to no effect on training outcome. These methods are complex and rely on extra data processing.

We present a novel *filtering* mechanism to do strategic data sampling in image classification problems based solely on the boolean metric of sample classification accuracy and consider how it performs compared to the *de facto* standard of mini-batch SGD. We compare the two in terms of accuracy, mean loss, worst-case loss, quantile losses, and wall-clock time.

We employ large-scale structured experiments to evaluate performance across a large set of hyper-parameter combinations and find that our filtering approach fails to achieve trends seen in other strategic sampling mechanisms. Instead, we find our strategic sampler variant has its own merits, showing a tendency to reach similar losses between training and test datasets, indicating a generalising behaviour.

Keywords: Supervised learning, Strategic sampling, Importance sampling, Image classification,

Preface

The work and experiments presented in this thesis were conducted throughout 2020–2021. All experiments were carried out using Python 3.8.6[1], frameworks and libraries extensively used for the experiments include — but is not limited to — PyTorch[2] (v1.7.1) by FAIR and NumPy[3] (v1.19.2) by Travis Oliphant (and community contributors). Hence; any code, pseudo-code, figures and tables specific to technical details of our works may be contingent on said frameworks and libraries. The intent of these, and embodied meaning should however still be legible.

Professor Zhirong Yang of the Department of Computer Science at NTNU, and Senior Research Scientist John Reidar Mathiassen at SINTEF Ocean AS were my supervisors.

[1] `https://python.org/`
[2] `https://pytorch.org/`
[3] `https://numpy.org/`

Acknowledgements

I would like to first thank my thesis supervisors: Professor Zhirong Yang of the Department of Computer Science at NTNU, for accepting to supervise in an external, self-provided master thesis problem, allowing me to do the work conducted in this thesis. And my second supervisor: Senior Research Scientist John Reidar Mathiassen at SINTEF Ocean AS, for creating the thesis objective. This thesis would literally not exist without that. But I would also like to thank you for the countless Teams-meetings we've had, practically on a weekly basis, and often lasting several hours. Not only were these meetings incredibly rewarding as a way for me to bounce ideas back and forth, and to keep my thesis on track, but also as a way to gain some insight into the world of research and how everyday life is like in research.

I would like to thank the people of NTNU's High Performance Computing Group for allowing me to utilise their IDUN cluster [see Själander et al., 2019], without it, and the compute time on it, I might've never been able to do large-scale experiments, and, as such; never been able to build a conclusive understanding of the results of our experiments.

It has been a tough and weird year, so I would especially like to thank the people closest to me; my family, for comfort, advice and support during this thesis writing period. Finally, I would like to thank my friends and flatmates, for riding my back about working on this thesis instead of twiddling my thumbs for the last stretch of the year. Klara, especially, for freely giving hours and hours of your time to help me past the few last hurdles and get here on time (I know I'm pretty slow).

I am truly thankful to have so many people to push me like that while also being so supportive!

Thank you!

Halvor Bakken Smedås
Trondheim, June 14, 2021

Contents

Acronyms and Abbreviations

AI Artificial Intelligence. 1–3

ANN Artificial Neural Network. xii, 1–4, 10, 53, 54, 60, 61

BGD Batch Gradient Descent. xii, xiii, 14–16, 22, 23, 53, *see* Batch Gradient Descent

CNN Convolutional Neural Network. 10, 11, 51, 52

DNN Deep Neural Network. 9

MBGD Mini-Batch Gradient Descent. xiii, 15, 16, 22, 31, *see* Mini-Batch Gradient Descent

PRNG Pseudorandom number generator. 35, 38, 59, *see* Pseudorandom Number generator

SGD Threefold meaning, *see* Stochastic Gradient Descent. xiii, 8, 14–17, 22, 26, 52, 53

Glossaries

Novel Terms

Adaptive Sampler An adaptive sampler is a strategic sampler that uses adaptive strategies. Strategies that are adaptive to the training model's current state, and so make a different selection to what it would in another state of being. xi, 19, 22, *see* Strategic Sampler

Archetype The *archetype* refers to the de facto standard way of performing supervised machine learning, Using Mini-Batch Stochastic Gradient Descent. 19, 21, 22, 30, 33, 34, 36, 39, 51, 52, 54, 55, 58, 59

Strategic Sampler A data-sampler that makes a *designed selection* according to some strategy (other than stochastic). A strategy either being a sound analysis of current state-performance, or a more informal strategy, like an educated guess of an optimal sampling scheme. In this thesis strategic samplers can also generally be considered adaptive samplers. xi, 2, 4, 6, 33, *see* Adaptive Sampler

Trainer A trainer is an encapsulation of the training loop. The trainer's main functionality is to train the model. Additionally it supplies an interface to perform external operations upon starting & stopping the training loop, starting & ending an epoch, and starting & ending a batch.. 17

Artificial Intelligence Concepts

Batch Gradient Descent Gradient descent performed on gradients computed (typically as a mean) of all samples in a dataset. ix, xii, xiii, 14–16, 22, 23, 53

Classification Classification is the process of being able to label some data. An image classification for example, would involve attributing a class to an image based on its features.. 10

Cross Entropy Loss A popular loss function used in classification problems. Although it's definition is as follows:

$$\mathcal{C}(y, \hat{y}) = -\sum_{n=0}^{N} \hat{y}_n log(y_n)$$

it is more easily presented as $-log(y_i)$ where i is the index of the labelled class. 34, *see* Loss

Data Augmentation A process in which we transform the sample data to a new, valid variant of the same data. In image classification such an addition is often crucial to achieve high accuracy. Typical transformations include adding noise, random cropping, and the affine transformations: rotation, mirroring, scaling, and translating. 28, 29

Gradient Descent An iterative optimization algorithm prominently used as a foundation for modern machine learning. Adjusting parameters of the model according to their contribution in making the model erroneous - i.e. according to the gradients of the loss. xii, 1, 14–18, 22, 23

Loss The metric we're trying to optimize for when training a model. It is an evaluation metric for how the model is performing. Several loss metrics exists, where the main requirement for a loss function is that it is differentiable, allowing us to extract valid gradients that can be used for gradient descent. xii

Mini-Batch Gradient Descent Gradient descent performed on gradients computed (typically as a mean) of a mini-batch of samples of a dataset. Typically the mini-batches are constructed as subsections of a random permutation of the full dataset, referred to as mini-batch stochastic gradient descent. ix, xiii, 15, 16, 22, 31, *see* Mini-Batch Stochastic Gradient Descent

Mini-Batch Stochastic Gradient Descent An approximation of Batch Gradient Descent (BGD) using a significantly smaller, random selection of the entire dataset, organised into batches. The idea being that the samples within the dataset are representative of the whole concept, to allow the model to generalize and perform well without being presented with all the samples from the dataset. xii, xiii, 1, 3, 4, 9, 15, 16, 21, 33, 57, 58, 63, *see* Mini-Batch Gradient Descent & Stochastic Gradient Descent

Model An instance of an Artificial Neural Network (ANN). The subject of optimization in a machine learning problem (i.e. its *learnable parameters*: weights and biases). xi–xiii, 1, 4, 8–10, 22, 26, 29, 34, 61

One-Hot encoding A one-hot encoding is a binary representation of a class label consisting of a bit-mask where the only active bit is at the nth index for the nth class. It is a useful encoding as it can easily be used in vectorized operations, such as when calculating the loss. 13

Outlier An outlier is a sample that doesn't really fit in the dataset it's supposed to represent. An extremum which we generally do not want our classifiers to handle. 27

Overfitting A model is said to be overfitting when its performance in the training environment is better than its performance in its operational environment - i.e. the model has specialized in niche features of the training dataset, as opposed to be generalizing. 9, 10, 39

Parameter Learnable parameters are, generally speaking, the *weights* and the *biases* of an ANN, the values are over time tweaked during learning, using gradient descent. 18

Quantile Loss The loss of the qth quantile prediction of the model, ranking from 0 — being the best-case loss — to 1 — being the worst-case loss, at $q = 0.5$ you will find the median loss. In this thesis we tend to look at $q = 0.9$ because it gives an idea about how large the portion of samples the model is performing bad on is.. 53, 54

Sample A Datapoint from a dataset. In supervised machine learning a sample could be considered a tuple of some data, and a label for said data. Depending on the context, *the sample* might also refer to just the data within the sample. xii

Stochastic Gradient Descent A somewhat misused term, drifting from its original meaning. In this thesis it refers to either of these (though generally 1.):

1. A stochastic approximation of gradient descent. Performed on a randomly selected sample of a dataset.
2. The unification of MBGD and definition 1. — Mini-Batch Stochastic Gradient Descent.
3. The PyTorch optimizer, a generalized optimizer structure used for performing BGD, Stochastic Gradient Descent (SGD), Mini-Batch Gradient Descent (MBGD) with additional parameters for optimizations such as momentum and nesterov accelerated gradients.

. ix, xiii, 8, 14–17, 22, 26, 52, 53

Worst Loss The loss of the worst-case prediction of the model. xiii, 51, 53, 54, 58

Programming Concepts

Pseudorandom Number generator A pseudorandom number generator is typically the approach to obtain "random" values on computer systems. Generally speaking, a PRNG has an initial seed, and a current state. This grants us a way of setting up PRNGs that generate the same sequence of random values for differing instantiations of code. All use of PRNGs in this thesis take use of the Mersenne-Twister algorithm.. ix, 35, 38, 59

Tools and Datasets

CIFAR-10 The Canadian Institute For Advanced Research 10 dataset [**?**]. A dataset of real world objects in tiny color images:

- automobile
- bird
- cat
- dog
- frog
- airplane
- deer
- horse
- ship
- truck

There are 10 classes, each with 5000 training and 1000 test images associated with them. The samples are rgb-color images of size $3 \times 28 \times 28$, i.e. 24-bit precision, *true color* pixels.

`https://www.cs.toronto.edu/~kriz/cifar.html`. 25, 27, 35, 36, 38, 39, 42, 44, 46, 51, 55

CIFAR-100 The Canadian Institute For Advanced Research 100 dataset [**?**]. A dataset of real world objects in tiny color images. There are 100 classes, each with 500 training and 100 test images associated with them. The samples are rgb-color images of size $3 \times 28 \times 28$, i.e. 24-bit precision, *true color* pixels.
`https://www.cs.toronto.edu/~kriz/cifar.html`. 25, 27, 35, 36, 38, 39, 46, 48, 54, 55, 61

GitHub User Kuangliu's CIFAR-10 with PyTorch Implementation An implementation of a vast collection of high-performing neural network models, implemented in PyTorch [**?**]. The basis for most of the used models in the work for this thesis. `https://github.com/kuangliu/pytorch-cifar`. xiv, 38

MNIST The Modified National Institute of Standards and Technology dataset [LeCun et al., 1998]. A dataset of handwritten digits 0–9. There are 10 classes, each with approximately 6000 training and 1000 test images associated with them. The samples are grayscale images of size $1 \times 28 \times 28$, i.e. 8-bit precision pixels. For the purposes of this thesis, all samples are padded with zeros to be of size $1 \times 32 \times 32$, to appropriately fit our modified grayscale image-accepting versions of the models of *Kuangliu's CIFAR-10 with Pytorch*.
`http://yann.lecun.com/exdb/mnist/`. 22, 26, 27, 36, 38–40, 42

List of Figures

List of Tables

I

Introduction

State-of-the-art ANNs are predominantly trained on random subsets of data. However, are we naïve to think that learning on random data is the optimal way for an Artificial Intelligence (AI) to learn — that sampling data to train on in randomly arranged batches yields the best results?

Whenever we are training ANNs, we tend to opt for a stochastic approach, using random subsets of the available training data, hoping for the best possible outcome. Generally, the algorithm known as mini-batch SGD (and other subsequent derivations) is the go-to options, as they have proven advantageous in training time, memory footprint, accuracy and overall robustness to a wide range of problems. With it, randomly sampled data are iteratively presented to an ANN model, and through gradient descent, the model will be adjusted to perform better.

Is there no way for us to easily determine if some data might be more important than others? To tell which might be more useful to train on? Or maybe easier; which samples that are not useful to train on?

In this chapter we will present the underlying motivation for researching strategic sampling mechanisms, leading into the research questions we raise in this thesis and the goals we have when conducting this research, furthermore, it will present the contributory objectives this thesis will have to the field of AI and AI-research. Finally, this chapter will present the scope of the thesis and research project, alongside an overview of the structure of this thesis.

⁂

1.1 Motivation

The availability of data accessible to everyone has exploded. With that, gathering of data to build datasets for training ANNs is no longer a problem. As this is the case, we now instead ask ourselves, which of these data are actually relevant? Using all of the data will cause a slow training, as a large portion of the data may be entirely unimportant to find a good fit for the problem. The computational cost, and time of training on unimportant data when working on large scale neural network models is not negligible.

As ANNs grow more and more complex, the computational cost of training has led to a small surge of interest in the concept of importance sampling (or what we refer to it as in this thesis: *stategic selection*) in in the deep learning community. Katharopoulos and Fleuret [2018]; Song et al. [2020]; Jiang et al. [2019] all relatively recent presented results indicating that strategic sampling could lead to a significantly better accuracy in a fixed wall-clock time. This motivates us to find a novel approach to selection of data samples and look into strategic sampling as we see a potential to improve performance.

Katharopoulos and Fleuret [2018]; Jiang et al. [2019] both reiterate the idea that samples aren't equally important. That some samples may be more valuable to train on than others. They both state that some samples might as well be ignored. This is an idea that we latch onto, and eventually bring forward as the basis of our novel approach.

Chang et al. [2017]; Song et al. [2020] Find that by controlling sample selection they were able to reduce variance which then lead to higher accuracies. However in doing so they introduce a higher potential for overfitting. We take a note of this, and want to see if we can devise a novel approach that avoids overfitting.

1.2 Goals and Research Questions

Goal *Explore and implement strategic data sampling mechanisms for improved training of ANNs.*

The de facto standard in machine learning is some form of mini-batch stochastic gradient descent, fundamentally dependent on random sample selection. We want to take a side step, and look at non-random, or less random sampling procedures. We propose that there are ways in which a strategical selection can be made autonomously based on intermediate analysis of the trained model's own performance.

Research Question *How do strategic data sampling methods perform as an alternative to the de facto standard of mini-batch stochastic gradient descent?*

Given the information we have access to during training, rules and procedures can be established, creating the potential strategy-creation, and, as such; strategic samplers. We would like to find whether such a strategic sampler bring forward new dynamics within the training of an ANN. Could it compare to stochastic sampling? Could it lead to better training? It raises the questions:

Sub Research Question *How do strategic data sampling methods perform in terms of*

RQ1.1 *Accuracy*

RQ1.2 *Loss*

RQ1.3 *Wall-Clock Training Time*

There is a multitude of measures one could look at when comparing methods in computer science, and the deeper we delve into a topic, the more specific we can get on said measures. In AI, it falls natural to look at how the final performance of our models compares with differing hyperparameters — but equally, it falls natural to look at timing aspect of it; performance over time; how our models evolve. In either case, *performance* remains an umbrella term for several measures — be it mean-loss, training time, accuracy, or other, more niche measures specific to a certain type of setup.

We intend to delve into these measures, and find instances of data either supporting or opposing the idea of strategic samplers, and see if they too have a place in the field of AI.

1.3 Objectives of the Thesis

With this thesis, we intend to present contributions to the field of AI by questioning the *modi operandi* of data sampling for supervised learning. To follow up on this, We present novel strategies to data sampling.

Ultimately this thesis is intended to

1. Present the idea of strategic and adaptive sampling as a viable option to stochastic sampling.
2. Explore an area of optimization, currently being investigated by the deep learning community. i.e. the sampling algorithm.
3. Introduce a series of concepts that when employed in sampling, can aid the training process, and allow for more control of what the training focus will be.
4. Present empirical evidence under several different settings/conditions. And present recommendations based on what we see.

1.4 Research Method

To answer our research questions, we employ extensive experiments as our primary research methods, gathering quantitative data. We strive to explore the relation between a ANN's model and the presented data it learns from. Specifically, we intend to explore an alternative way of selecting which data the model should learn from at any point in time throughout training.

Through extensive parametrically configured experiments, we empirically demonstrate the capabilities of such an alternative selection strategy. By running the experiment using this wide set of parameters we intend to illustrate the limits of when our suggested approach is applicable and when it's not. We further identify benefits, drawbacks, and otherwise interesting findings, and highlight these when discussing further opportunities our novel approach enables. To this end, we identify that with extensive experiments, there is a need for extensive tracking of the various attributes a supervised learner holds and inheres too.

As this is already an issue addressed from early on in machine learning, there is a multitude of tools accessible to be used for tracking our models' performance. In preliminary work we used *TensorBoard*[1], a toolkit for machine learning metric-tracking and visualisation initially built as part of the Google Brain's *TensorFlow*. The somewhat limited features for data export in TensorBoard lead us to seek out an alternative for the work conducted in this thesis; hence we will be using *Weights & Biases*[2] (also known simply as *wandb*) by Weights & Biases for inspection instead in this thesis.

As we explore an alternative approach to data sampling, we naturally compare our novel approach to that of the de facto standard data sampling: mini-batch SGD. This allows us to project how our approach would work in the general use-case where an ANN is employed (for image classification), widening the applicability of the results found.

As such, when presenting the results of approach, we will always present it in relation to the equivalent run using standard mini-batch SGD. We present the results not only in respects to the overall accuracy of the trained model, but also in terms of losses — mean and worst loss, in addition to the quantile loss ($\mathfrak{q} = 0.99$)

1.5 Thesis Outline

The thesis introduction now concludes (Chapter 1). Following, we will extensively present the fundamental background theory which paves the way for us to introduce our idea. We keep this extensive and thorough, as we intended our work to be self-contained, allowing the everyman to read this thesis too, if they so desire. We will revisit the work of others, to see where our research brings novelty to the field. Through others work we will present our motivation to strive for strategic sampling as an approach (Chapter 2). Subsequently, our developed methods and mechanisms will be presented, and our preliminary work that paves the way for our final architecture used in the experiments (Chapter 3). The chapter after will present our setup for exhaustive experiments and the parameters used. We go over and cover all bases needed for reproduction of our results, as well as the results themselves (Chapter 4), before we conclude with an evaluation of our findings, a discussion about the research conducted, and sparks of ideas for potential future work (Chapter 5).

[1] `https://www.tensorflow.org/tensorboard/`
[2] `https://www.wandb.ai/`

II

Background Theory & Motivation

Before we delve into the depths of our new approach to strategic data sampling, we will first establish the state-of-the-art, and talk a little bit about the theory employed to get to this point, so that we can more easily pinpoint where our addition is, and how it plays a part of the bigger whole.

To that end, this chapter lays the foundation for where our contributory addition to the field of AI and the topic of sample selection is. We start of looking at literature central to our the work we will conduct, namely strategic sample selection. Wanting to ensure quality data to work from, we formalize a structured literature review protocol before we dig into the related works. From there we reiterate our motivation, grounded in the literature, before we move on to background theory. This we present thoroughly and in depth (experienced readers may freely choose to skip this, as it covers most of the needed material up to our new addition. We suggest however, to read on from section 2.4.4).

2.1 Structured Literature Review Protocol

Situating ourselves in the field, and finding related research involves a process of extensive searching through web archives, journals, books or otherwise. We have had to sift through material to identify similar research both in terms of research questions, and in terms of proposed solutions. When doing so we applied specific criteria to ensure we could quickly discern the works that were relevant, while also ensuring that the found research was up to standards, and could be trusted.

	Term: #1	#2	#3	#4
Group: ①	Sampling	Batching		
②	Stochastic	Uniform	Random	
③	Sample	Instance	Example	Data Point
④	Importance	Utility		
⑤	Importance ①	Strategic ①	Non-① ①	Non-② ①

Table 2.1: Literature inclusion and quality criteria. Using combinations of these, we were able to find primary sources of relevance to our work, and further fill the gaps by citation network traversal.

Search engines used to find related works were primarily Google Scholar[1], SemanticScholar[2], ResearchGate[3].

As is often the case in research, we found that others before us have been using differing terminology from what we initially were using. This ended up shifting our search frame drastically. For example, "adaptive" was considered a central keyword in our mind, we found that in fact many before us had not considered the use of adaptive about their approaches. A representation of the final search frame used can be seen in table 2.1.

One of the additional ways utilised to discover related works were citation network traversal. After performing initial searches using the keywords, we used tools such as ConnectedPapers[4] for quick expansion of found materials, traversing through the found works' related works-section, this yielded a more thorough overview of the works done within our research frame, and we could from this surmise that our work was novel.

From this point it was a matter of quick analysis, identifying whether or not the works were of import to our work, and so should be included, and secondly if the quality held up to certain criteria. The criteria used can be seen in table 2.2.

2.2 Related Works

In the last decade, several works have investigated methods for squeezing every bit of *usefulness* out of every training step in training. Some looking into the depths of the mathematical foundation we already have for training neural networks, namely the gradient descent algorithm, and how its components can be put together to form better batches. — Approaches that utilise the information we already have at hand during training: Losses and gradients. Others accumulate information over time to build auxiliary data that can be used to the same effect. As we found, there are 3 primary methods fundamentally achieving designed selections.

We note that Jiang et al. [2019] in fact is a preprint, and should be treated as such. Per protocol they should not really be included, as their status in peer review is unknown. However, upon seeing how similar some of their work is to ours, we make an exception.

[1] `https://scholar.google.com/`
[2] `https://www.semanticscholar.org/`
[3] `https://www.researchgate.net/`
[4] `https://www.connectedpapers.com/`
[5] if it is, use the one with the most recent results

Inclusion Criteria	①	The main topic covered is non-uniform sampling.
	②	Empirical evidence of improvement is presented, alongside a thorough discussion about the results.
	③	The work describes their approach/-es sufficiently.
Quality Criteria	①	The work is peer-reviewed or heavily used (and; as such, has stood the test of time)
	②	The work is not a duplicate of other work found[5].
	③	The presented approach/-es is/are compared to others' works.

Table 2.2: Literature inclusion and quality criteria.

Relevance	Category	Name
1	Importance Sampling	Accelerating Deep learning by focusing on the Biggest Losers [Jiang et al., 2019]
1	Importance Sampling	Not All Samples Are Created Equal: Deep Learning with Importance Sampling [Katharopoulos and Fleuret, 2018]
2	Importance Sampling	Training Deep Models Faster with Robust, Approximate Importance Sampling [Johnson and Guestrin, 2018]
2	Temporal & Uncertainty	Carpe Diem, Seize the Samples Uncertain "at the Moment" for Adaptive Batch Selection [Song et al., 2020]
3	Temporal & Uncertainty	Active bias: Training more accurate neural networks by emphasizing high variance samples [Chang et al., 2017]
4	NeuralNet	Autoassist: A framework to accelerate training of deep neural networks [Zhang et al., 2019]
4	NeuralNet	Mentornet: Learning data-driven curriculum for very deep neural networks on corrupted labels [Jiang et al., 2018]

Table 2.3: Relevancy overview

Importance Sampling Strategies Inferring utility of a sample in terms of their effect on the parameters of the neural network training on them will. The bigger the change a sample inflicts on the network's parameters, the more useful it can be deemed.[6]

Temporal & Uncertainty Strategies Inferring utility of a sample in terms of the certainty the model has and has had about the given samples over time.

NeuralNet Strategies Inferring utility by use of a secondary neural network.

2.2.1 Importance Sampling Strategies

Katharopoulos and Fleuret [2018] find preceding works to employ the losses or the gradient norms of samples to determine their utility. In their paper, they present two contributions, and results indicating an overall improvement to others given a time-budgeted training period. Their first contribution lies in a derivation of a tractable upper bound to sample gradient norms, and the second is an estimator of potential variance reduction achieved when using importance sampling. The latter of which helps to ensure that importance sampling only is used when it's deemed to give a training speedup.

However, the trend has lately seemed to shift slightly, and although the sample losses and gradients still remain the major part of the more recent works, many posterior works after Katharopoulos and Fleuret [2018] attempt to instead approximate ideal distributions, rather than to find the entire distribution.

Johnson and Guestrin [2018] is one of these works, in it Johnson and Guestrin [2018] provides an approach in which they first devise and term *Oracle-SGD*, an iteration upon standard SGD that samples non-uniformly to reduce variance by computing the gradient norms and additionally adjust the learning rate adaptively. Their iteration upon Oracle-SGD lies in the approximation they do to mitigate the issue of time-consuming gradient norm calculations.

On the other hand, there's still faith in the simpler approaches to be useful too. We found this in the more recent Jiang et al. [2019], which make their selection strategy tightly linked with sample losses. Their approach involves a process in which all samples are passed to the model, as one would with standard SGD. The difference is that instead of using said forward passes to calculate gradients and updating the model right away, they process the losses and build a probability-distribution for sampling said samples based on their loss.

2.2.2 Temporal & Uncertainty Strategies

Temporal, history-tracking strategies have had a resurgence in the last couple of years, with the most recent being [Song et al., 2020], in which they propose a sampling scheme based on a sliding window, i.e. a limited view on the past history. What they track is not the losses or gradients, but rather the past predicted labels. They suggest that doing so — using the model's uncertainties — will provide a useful sampling scheme to not only accelerate the training, but also reach higher accuracies.

Chang et al. [2017], as also referred to by Song et al. [2020], approaches the problem with the same idea in mind; sampling based on the uncertainty of a sample. They present a scheme where the history is kept of previous softmax activations, essentially tracking the confidence the model has had towards the correct label throughout training. Evaluating the variance over this history gives a sample's uncertainty.

2.2.3 NeuralNet Strategies

For completeness, we deem neuralnet strategies important to mention too, as they too seem to be an ever more prominent way of performing machine learning with good results. Sampling strategy is at the core of Zhang et al. [2019]. In their work, they propose a light-weight *assistant*

[6] Obviously, given that the label of said sample is correct.

network attached alongside, and jointly trained with the main model, where the light-weight assistant is used to pick determine a sample's importance. Their process from there is to filter out non-important samples from the sample pool, similar to how we too filter out samples from our sample pool during training.

Other recent approaches have also been suggested utilising an additional *attached network* alongside the *main network*. Albeit with slightly differing goals (focusing on performance of deep networks trained on corrupt training data), Jiang et al. [2018] propose to have a dual network structure; MentorNet and StudentNet, where the MentorNet learns a data-driven curriculum — a sample weighting scheme, for the StudentNet to learn from, indubitably showing the dual network structure to be viable in terms of generalisation.

2.3 Motivation

In modern machine learning we are essentially bound to chance, in the sense that we're relying heavily on stochastics in so many aspects when training. We see this as an intrinsic motivation to explore other alternatives. Furthermore, while several alternatives have been suggested, and a lot of them seem to work, we believe there is still room for novel approaches. Our motivation thus stems from findings of the reviewed prior works, findings we have categorised twofoldly:

Learning Time —— While stochastic sampling certainly has proven itself as a robust and strong contender in the game — one that will likely still be with us for years to come — we cannot help asking "is it truly not possible to do better than random?". Putting it as Katharopoulos and Fleuret does: *not all samples are created equal*; some samples are more useful than others at different points in time throughout training, and sampling only those (instead of noisily including samples that don't contribute as much) could potentially shave of not merely minutes, but potentially hours and days of training time. It is just a matter of finding which ones are more important. Common for [Katharopoulos and Fleuret, 2018; Jiang et al., 2019; Song et al., 2020] is that they all present having had improved training speeds in terms of wall-clock time reaching same-accuracy parameters compared to random-batch, i.e. mini-batch SGD.

Variance Reduction through Supervising Supervised Learning —— Supervised learning relies on the loss function to learn from the gradients. Others before us have further utilised the loss function to also do importance sampling, either by using the losses themselves, or the gradient norms [Katharopoulos and Fleuret, 2018]. Newer approaches also suggest that indirectly controlling; "supervise", supervised learning by controlling which samples to be presented reduces variance, and in so doing, increases accuracy [Song et al., 2020; Chang et al., 2017], albeit with the added danger of overfitting.

From these findings we establish our twofold motivation:

Motivator 1. Improved performance —— As both points above emphasise, several works point out an overall improvement in performance through designed selections. Hence, we see our potential in adding to the knowledge-pool, and investigating yet another new type of designed selection for improved performance (with respect to test accuracy, learning speed to reach fixed accuracy, and worst case loss).

Motivator 2. Limiting Overfitting —— Contrary to Motivatior 1, an emphasised problem of using designed selections is the increased risk of overfitting to the training data. Overfitting it self is a result of how the loss function is shaped. Minimising the losses over time will often cause the network's parameters to be too fine-tuned for the specific data of the dataset, as it is allowed to continue tuning its parameters to better fit data even though it's already capable of classify correctly. This will narrow the boundaries

of what data it will accept as part of the individual classes and cause the network to be more and more capable of handling the training data, but at the same time, in the worst case, less and less capable of handling data outside of the training data.

> *"You get what you ask for; not what you want."*

This is a phrase often used in relation to AI, and specifically about overfitting. Where *what you ask for* refers to the loss function, and *what you want* refers to what you actually want the model to learn.

We want to investigate what effect it might have on the learning if we actually manage to limit the training by *what we want* by not allowing the model to keep minimising on samples it already correctly classifies.

2.4 Background Theory

Let us now establish the fundamental background theory employed in our works. The experienced reader may skip this without missing out. Just be sure to catch the last piece from this section section 2.4.4, as it lays down some terminology core to the rest of our work.

⁂

Artificial Neural Networks (ANNs) are connectionist systems modelled to vaguely operate like humans' biological brain, being a cluster of interconnected neurons that through synapses bring about senses, thoughts, reasoning and behaviour. An ANN tries to achieve the same by sequencing nodes, a construct really only doing a small piece of math, simulating the inner works of synapses in the neural systems of humans and animals, organised into layers where each layers' nodes feed into each of the following layers' nodes.

The ANN as a construct is nothing more than a series of mathematical operations linked together in a way that allows it to over time encode some *meaning* into the various parameters and series of connections the network holds. As such, an ANN can typically be represented as a series of vectors, matrices and tensors.

This thesis will focus on one specific type of task these ANNs often are used for, namely *classification*; More specifically, we are going to be focusing on *image classification*, though we suggest that the general approach we take in the thesis should be equally applicable for other types of classification as well.

2.4.1 Image Classification

Image classification is all about determining what pictures' contents are. A common example is distinguishing whether the animal depicted in an image is a dog or a cat, but the more complex — and perhaps more useful — type of image classification deals with several hundreds of classes. A real-world example of where an image classifier is helpful is the classification of fish species either on still images or on a video feed.

The ANN structure is often considered the simplest of them all is the *fully connected network*. On its own, it is rarely enough to constitute as a good image classifier network by itself. It struggles to build an encoding that utilise the structural parts of the images we give it. In other words, its missing the ability to easily register and recognize the lines, corners, spaces, indeed the very structure the pixels together form.

Convolutional Neural Network

Enter the Convolutional Neural Network (CNN)! CNNs fundamentally build a way to operate on and learn structural features. It does this by splitting the classification problem into two parts. The first part is dedicated to finding features, and the second part classifies based on

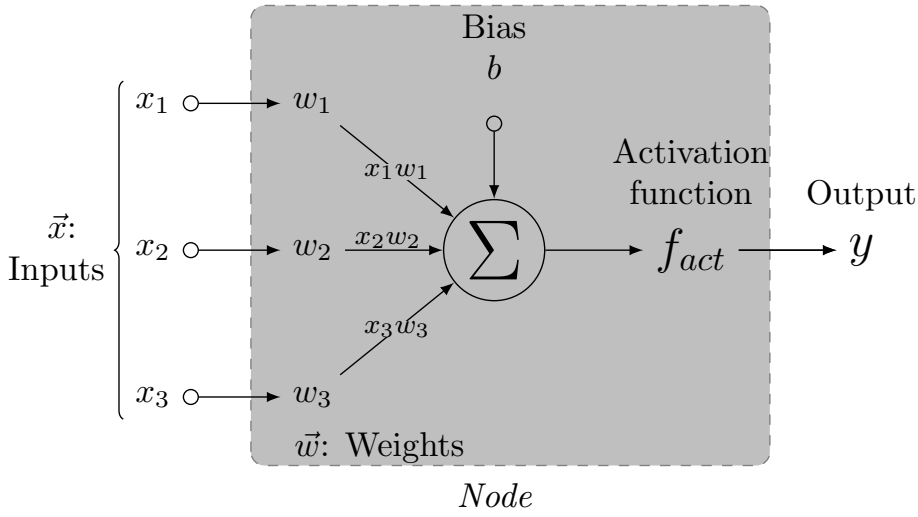


Figure 2.1: The fundamental node structure of an Artificial Neural Network

the collection of features found; as such, the CNN's fist couple of layers are a series of filters (for finding features), and the second part, a fully connected network structure.

In essence: Instead of using vectors of weights and multiplying them with the input before being summed (a fully connected layer, see fig. 2.1), the first couple of layers uses matrices of weights that it then uses to convolve the image. This in turn, gives a series of (typically) smaller images that could be considered *structural encoding-images*(see fig. 2.2), where bright regions represent a high presence of the feature described by the filter matrices. Doing convolutions like this in a sequence, with different filters for each layer, one ends up with a small image representing high-level structural features of the image.

The *encoding-images* are then passed on to the fully connected part, which considers and learns the constellation of high-level features, and classifies the original images on said features.

The Ins and Outs of The Image Classifier & The Categorical Cross Entropy Loss

If we step back for a second and consider the neural net component of an image classification process a *black-box*, an essential aspect of image classification through neural networks are the dimensions of the presented data.

Like all neural networks, we obviously send all data in, so we send the image, or even multiple images at once, as neural networks generally can process the data parallelly, as vectorised operations. The somewhat more interesting thing to see is that we get a vector out, as a result of passing the data through the network.

After all, isn't what were after a class identifier, not a vector of continuous values? — The neural network is in fact giving back a distribution of the class spectrum. It gives back a distribution of its beliefs or *confidences* as they often are dubbed. From this, we consider the index of the element in the distribution with the highest value as the predicted class (with a *confidence* of the element's value).

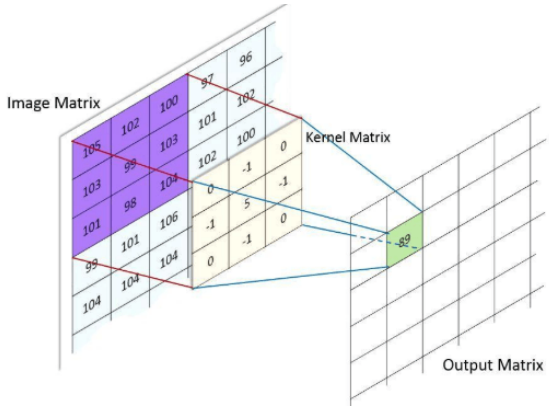


Figure 2.2: Convolution through a weight kernel of the CNN [Asthana, 2020]

⁂

There is a slight catch that needs to be mentioned here too, which is that the values passed out from our black-box, the neural network, is in fact slightly dependent on the framework the net is developed in, and slightly dependent on how the output is intended to be used. Oftentimes, the network's output hasn't yet been turned into a distribution (or probability mass function, as it doesn't sum to 1 yet, that typically happens as the output is passed through a softmax function, which can either occur in the network itself, or be incorporated in the loss function used for learning, and training the network. We present the above case as we do because we consider it to widely be considered the approach standard in image classification.

⁂

The next important step to cover when talking about image classification networks, is how the network can go on to actually learn anything from its current state, and from passing images through it. We will go more into details about that in section 2.4.2, but before we move on to it, we present the idea of *cross entropy loss*.

To know if we're doing well in a task, beside simply evaluating whether an image was classified correctly or not, we can also consider the previously mentioned *confidences* of the neural network; the distribution of beliefs about which class an image belongs to. It makes sense to draw the link between high confidence of a classification and the performance of the classifier (given, of course that the high confidence is placed in the appropriate class for the image). To put it another way; A classifier that classifies something correctly with high confidence is better than a classifier that classifies something correctly with low confidence. Equally, it applies to say that a classifier with low confidence in the incorrect class is better than a classifier with high confidence in the incorrect class.

These are the fundamental ideas of the *Categorical Cross Entropy Loss* function: Reward (low loss) high confidence in the correct classifications and, in effect, low confidence in incorrect classifications, and punish (high loss) misplaced confidence and low confidence in correct classifications.

$$\mathcal{C}(y, \hat{y}) = -\sum_{n=0}^{N} \hat{y}_n log(y_n) \tag{2.1}$$

where:

y	is the confidence vector, the output logits of the network softmaxed (to get a probability-value per class).
y_n	is the confidence score (or probability, if you will) of the sample being of the nth class.
$\hat{y}$	is the one-hot encoded label of the sample passed through the network.
N	is the number of classes in the problem.

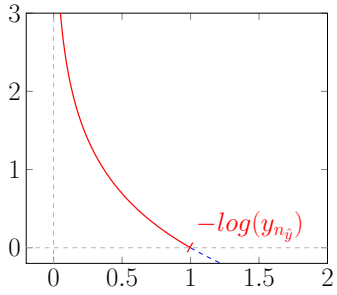


Figure 2.3: Cross Entropy Loss

In essence, eq. (2.1) boils down to the negative logarithm of the confidence in the correct class; $-log(y_{n_{\hat{y}}})$ — $n_{\hat{y}}$ being the *correct class index*. Because our y-vector holds values in the interval $(0, 1)$, the logarithm will only ever return negative values, meaning that our negation grants us a positive value. In turn, because only the $n_{\hat{y}}$th conficence value ends up being used in the logarithm, we have that if $y_{n_{\hat{y}}}$ is small, our loss is high, and that if it's big ($y_{n_{\hat{y}}} \to 1$) our loss is low.

If we now were to pass all the data of a dataset — one completely different from the training data, while still in the same domain (let's call it the *test dataset*) — through our model, we can take the mean value of the resulting losses, and in so doing, we have a metric representing the current performance of the model.

This is how, broadly speaking, modern classification networks are evaluated. Not only that; it is also the first step in how the models are trained.

2.4.2 Gradient Descent

$$\theta^* = \theta - \eta \frac{\partial}{\partial \theta} \mathcal{C}(\mathcal{X}, \mathcal{Y}; \theta) \tag{2.2}$$

where:

θ	is the network's parameters	$\mathcal{X}$	is all data from the dataset
η	is the learning rate	$\mathcal{Y}$	is all labels from the dataset
$\mathcal{C}$	is the loss function		

The loss function signature here is denoted slightly differently from before. This is to emphasise that the resulting loss is not only a result of the data and the labels passed in, it is also a result of the current parameter configuration. In the equation above, and the subsequent equations we will see this. $\mathcal{C}(\mathcal{X}, \mathcal{Y}; \theta)$ *is really just an alternative way of writing* $\mathcal{C}(\text{net.Forward}(\mathcal{X}), \mathcal{Y})$*, that shows the variable we are going to do partial derivatives with respect to.*

Equation (2.2) is the very essence of gradient descent; the parameter update equation — or *the learning rule* as it is often called — where each of the neural networks parameters (weights and biases) are tuned in accordance with the partial derivative of the loss function with respect to θ. Put simply, we tune parameters according to their contribution to the loss, essentially lessening their contribution to the overall deviation from the ground truth. In doing so, "teach" the neural network to model relations from the features of the samples to the appropriate labels.

2.4.3 Gradient Descent Sampling Schemes

Fundamentally, there are three sampling schemes that are established as variations of how we bring about gradient descent in machine learning, all with their benefits and drawbacks:

Batch Gradient Descent The most straightforward variation is what's simply known as gradient descent, or often as *batch gradient descent*. It considers the whole dataset as one single batch (hence the *"batch"* in the name) to perform a training step. Depending on the size of the dataset, this may already impose a problem for performing machine learning. Even modern computers simply cannot load the bigger datasets out there into memory as a single batch at a time, let alone do operations on it afterwards. Besides this, the algorithm is painfully slow, seeing as it's only updating the model parameters *once* using all available data. As a result of the relatively low update frequency of BGD, we end up needing to run our training for longer, over more epochs.

Stochastic Gradient Descent Stochastic gradient descent is the second variation of gradient descent. It could easily be considered the converse of BGD, in that it takes sampling to the opposite extreme. In stochastic gradient descent we don't train on batches — or rather; we train on single element-batches. For each epoch the dataset is shuffled, then we take sole samples out from the shuffled dataset and perform gradient descent on them individually, resulting in a large number of weight updates in one epoch.

The fundamental idea behind SGD is to by chance happen upon samples that hold features other samples too hold, and in so doing, do generalised training steps that will improve overall performance. The assumption is that in a dataset there is a large amount of redundancy — Many of the samples are similar to each other, meaning that one should be able to progress faster by training on randomly selected samples that to some degree can represent the general concept.

An analogous way of comparing BGD and SGD is to imagine we're walking down a valley where arrowed indicators showing *a way* (not necessarily the right way) are placed all over. Wanting to reach the bottom, we can choose to at every point of the way sum all the indicators together (BGD), knowing that the summed direction will be leading us to the bottom. Alternatively, we can save time by not trying to take all the arrows into consideration, and rather pick a random one to follow (SGD), knowing that generally, the arrows point towards the right direction, and so, eventually (and hopefully faster than by doing the arrow summation) the arrows will lead us to *a bottom* in the valley.

We highlight another benefit of SGD in the analogy: Because we're not not following summed arrow indicators, we're not walking the most straight way down into the valley. Instead we might end up wandering around a little, and actually discover a deeper valley, which we then descend into.

In such a case, the stochasticity of our sampling has essentially lead to exploring connection combinations in our network model, leading to an overall better fit — We've found a better local minimum (or even the global minimum) of our error surface.

The negative side-effect of performing SGD is already somewhat highlighted in that analogy: Although random sampling might be beneficial in terms of generalisable features being utilised better when training, the consequences of sampling wrong data might actually lead to a severe worsening of overall performance (when a sample's data is very different from other samples' data with the same label)

$$\theta^* = \theta - \eta \frac{\partial}{\partial \theta} \mathcal{C}(x, y; \theta) \tag{2.3}$$

The parameter update in SGD is exactly the same as in BGD, with the exception of it only using the one sample x and its label y instead of the whole dataset.

⁂

A significant problem of SGD is in its semantics. It is not something we will tackle in this thesis, but nevertheless worthy of mentioning, as it may lead to confusion otherwise: The term

SGD is severely misused to the extent that it nowadays generally mean a combination of SGD and the following MBGD, combined in what is really mini-batch SGD. Additionally, as we will see in the following section, MBGD can be seen as a generalisation, so SGD has grown into an umbrella term. This especially applies when looking at how the various frameworks name their optimisers. In PyTorch for example, the optimiser used for standard gradient descent is also dubbed SGD.

Mini-Batch Gradient Descent Lastly, we have mini-batch gradient descent (which almost always is equivalent to mini-batch SGD, depending on if it uses stochastic batches or not), which in more than one way can be seen as the compromise between the other two. Actually, it is very close to being a generalisation between the two. It strives for the best of both worlds, aiming to get the accuracies and stable descent seen in BGD, while also performing training faster, as in SGD. Actually, MBGD generally has greater throughput than SGD too, because modern machines and machine learning libraries support highly optimised tensor, matrix and vector operations.

Foundationally, the aim of MBGD is to split the dataset up into subsets — mini-batches — that themselves contain enough data to to some degree represent the entire dataset (which again only is a limited representation of the reality our model might face after training). By batching like this, MBGD hopes to generally smooth the descent, like in BGD, making more sound steps towards a local minimum than SGD would cause it to do, while not disregarding the speed benefits of doing multiple training steps in one epoch.

How the mini-batches are selected is a central topic we will come back to, as this essentially is our entry point, but for now it will suffice to say that generally, they come as a result of shuffling the dataset then taking `batch_size` samples out of it. Doing so results in what one might call mini-batch stochastic gradient descent, but the case is in fact that mini-batch sampling has proven it self so useful, that the term SGD is pivoting to mean mini-batch SGD, while SGD itself fades away as the less useful gradient descent variant.

Depending on the `batch_size` selected, specifically when it is set to 1, the mini-batch SGD is fundamentally the same as SGD. This brings us back to the first statement about MBGD being a generalisation of the two others: It can equally be said to be equivalent to BGD in the case where the sampling scheme used is not a stochastic one, but rather a sequential one. If the batch size is set to `dataset_size`, the MBGD will be equivalent in behaviour to that of BGD.

Having illustrated that MBGD can be considered a generalisation, and because the general tendency seems to be shifting towards MBGD being the de facto standard, MBGD will be the gradient descent variation we will focus our efforts into further improving.

$$\theta^* = \theta - \eta \frac{\partial}{\partial \theta} \mathcal{C}(\mathcal{X}_{(i:i+n)}, \mathcal{Y}_{(i:i+n)}; \theta) \tag{2.4}$$

The parameter update in MBGD is yet again close to being the same as its predecessors. Instead of training on the entire dataset, or on one lone sample, it's using a smaller subset of the entire dataset for each training step.[7]

Sampling Schemes — Complexity

There are certain limitations we always seem to encounter when writing algorithms in the realm of computer science. Namely complexity; time complexity and space complexity. Seeing as our aim is to find a sampling scheme that improves training, it should naturally also be compared to the approaches we already have in terms of complexity. Complexity however is very dependent on implementation, as such, we have to take another side-step here, and declare once again that the presented concepts are based on the PyTorch implementation.

[7] Although this depicts a sequential sampling of the data and labels, the more typical instance of MBGD uses stochastic batches

⁂

For further reference on what is happening in these sampling mechanisms, refer to Appendix fig. 3

Batch Gradient Descent In terms of the sampling itself BGD is among the top contenders. The time complexity, next to no operations is required to get the entire dataset loaded into memory. As we access it through the `Dataset`-structure in PyTorch we have to get the entries iteratively however, and then collate it all to make it into a batch. The same rules apply for the other approaches too, so timewise, or operationwise BGD is literally the baseline. Accessing all the data would involve generating the indices in the interval $[0, n]$ for doing the lookup of the dataset. Though, seeing as all subsequent variants also have to do this step, we can quite simply ignore it.

In terms of space BGD is hardly advantageous in any respect. It is arguably *the* disadvantage of BGD. For larger datasets it might not even be an option, considering how much memory is required to hold the data, and offloading the memory into *swap* on the hard drive of the machine running the training would likely cause it to practically halt.

Stochastic Gradient Descent With SGD we generate a list of indices, again in the interval $[0, n]$. We go on to shuffle the list. From this point it's just a matter of using the indices stored in the list iteratively.

Mini-Batch Stochastic Gradient Descent With mini-batch SGD we generally do the same as in SGD; create an initial index-list, shuffle it, and then upon every batch start, we request k samples, instead of 1 as in SGD.

Variation	Ops. per Epoch (+ batching)	Samples in Memory per Batch
BGD	`1 + 1`	`N`
SGD	`3 + N`	`1`
MBSGD	`3 + num_batches`	`N / num_batches`

Table 2.4: Complexity through training of the commonplace gradient descent sampling schemes. `N` is the number of samples in the whole dataset

Variation	Sampled per Training Step	Training Steps per Epoch
BGD	Entire dataset	`1`
SGD	One random entry	`N`
MBGD	One random subset of the dataset	`N / batch_size`

Table 2.5: The three commonplace variations of gradient descent

Comparing the three variants, we see that the number of training steps done heavily rely on the way we sample our data. One way of thinking of the three variants is that they ultimately prefer either step quality, or number of steps, with MBGD/mini-batch SGD attempting to compromise between the two. A key point to make about the three variants is that they all are ambivalent to the actual content of the samples they do training on. The de facto rule is

that the training loop should go through the dataset in some fashion, be it sequentially, one by one, or stochastically in a batchwise fashion; the whole dataset must be consumed through an epoch. *Epoch* as a term is in fact often defined as one pass through all samples of the dataset. This has some implications for our approach, seeing as we are filtering samples, and so, we're not training on all data. We will come back to this issue in section 3.3.

2.4.4 Connecting it All — The Archetypal Learning Process

When writing machine learning algorithms there are certain components that regularly show up. Depending on where we are developing said algorithms, some components may stick out more clearly as separate entities of the larger system. Additionally, these entities may vary a little between the different programming frameworks, though at the time of writing *Tensorflow*[8] by Google Brain and *PyTorch*[9] by FAIR (Facebook AI Research) are the two most prevalent frameworks in the industry, and both encapsulate the following structures into relatively separable entities, so we consider them applicable for all intents and purposes of this thesis.

The training medium
: Either raw data or some generative process producing data.

The neural network
: Typically a composition of matrices, vectors, or tensors for high computational throughput, representing the parameters of the neural network; the weights and biases.

The optimizer
: An entity that performs *gradient descent* (typically SGD) computations. It computes and assigns updated parameters to the neural network model (ideally) making it perform better.

The loss function
: An equation describing the distance between the model prediction and the labelled ground truth of a sample.

With these commonplace components of a machine learning algorithm established, it's relatively easy to extract yet another encapsulation; what we've come to call a *trainer*. Trainers are instantiations of the previously mentioned components coupled together, making the trainer be the interface for the training loop.

With this, the most primitive version of an archetypal trainer emerges, encapsulating a training loop most AI enthusiasts and researchers would agree is the *standard* — The *archetype*. Coming up with new approaches on how we can improve machine learning further, it makes sense to look to the standard, both as a template where we can easily pinpoint which part we are trying to improve, but also as a way to demonstrate the improvements a novel approach brings; by comparing it to the archetype.

[8] `https://www.tensorflow.org/`
[9] `https://pytorch.org/`

Algorithm 1
Archetype Training Loop (see appx. fig. 7)

Require: A Dataset; $\boldsymbol{\mathcal{S}}$.
Require: A Loss Function; $\boldsymbol{\mathcal{C}}(\mathcal{Y}, \hat{\mathcal{Y}})$.
Require: An Optimiser; **Op**.
procedure TRAIN(net, $epochs$)
  for $e \leftarrow 0$ **to** $epochs$ **do**
    for all $(\mathcal{MB}_{data}, \mathcal{MB}_{labels})$ **in** $StochasticBatch(\boldsymbol{\mathcal{S}}, size_{\mathcal{MB}})$ **do**
      $predictions \leftarrow net.Forward(\mathcal{MB}_{data})$
      $losses \leftarrow \boldsymbol{\mathcal{C}}(predictions, \mathcal{MB}_{labels})$
      $gradients \leftarrow net.Backward(losses)$
      Op$.step(net, gradients)$ ▷ Adjust *net* parameters
    end for
  end for
end procedure

An essential detail to point out here is that obviously there are a wide range of ways one might structure the training loop. However, the encapsulation we have done when defining the archetype trainer, is based on the bare minimum for training a neural network model. By extension, the result comparisons we later present that uses the archetype as a baseline, should in general be extrapolatable for other types of training loops too.

Accuracy & Error

We've already touched upon how neural networks learn, using gradient descent through calculating losses of predictions, and then adjusting the contributing parameters to minimize said losses. But the losses themselves are not a proper representation of how well the model is performing. A correct classification may still have a relatively large loss if the classifier network's prediction is a relatively uniform distribution. As an example, given the CIFAR-100 dataset, with a relatively uniform distribution being the prediction of the network, even a correct classification could have a loss of $-ln(\frac{1}{100}) \approx 4.6$ given that we use the categorical cross entropy.

Our proposed method relies not on the losses or the strength of the predictions (confidences) our network makes, but rather on the *accuracy*. The boolean, non-differentiable value of whether or not the sample has been classified correctly. Hence we formally define *accuracy* to be precisely that, while also dubbing the opposite — when a sample is inaccurately classified — an *error*.

III

Architecture

We will now introduce our adaptive sampling strategy, and how it compares to the archetype's sampling in structure, explicitise what we are changing, and what we keep as is. We then move on to talk about the technical aspects using an adaptive sampling strategy. Before we move on to the, however: What does "adaptive" mean in the context of data sampling for supervised learning?

3.1 Adaptive Sampling

> ADAPT:
> *Having an ability to change to suit changing conditions.*
> — Cambridge Dictionary

With a formal definition of *adapt* readily at hand, we now establish what *adaptive* means in this thesis, and in the context of data sampling for supervised learning:

We identify that in our context the *changing conditions* are the state of the learning process. In other words; how the current fit of the model is — how well it performs. From that, we can further assert that in order to make data sampling adaptive to the learning, it has to consider the current performance of the model. Not only should it consider it, but it should also use it to further improve the learning outcome — *to suit the changing conditions* — essentially meaning that the current performance has to be utilized for picking out the most relevant samples for optimal learning.

Summarised, our approach to adaptive sampling incorporates the current model performance to pick out samples that have the highest possible utility, in terms of further learning.

The Utility of a Sample

Looking at the problem of finding the most relevant data points for learning presents another issue, namely; knowing what is relevant in the first place. The naïve, and infeasible way would be to simply try each and every permutation of sample batches possible, then feed them through the network, update the model, and evaluate which batch that resulted in the best update step being made — a brute-force approach.

Such an approach is not only infeasible, due to the inconceivably large number of possible permutations, but also totally defeats the purpose of making a designed selection in the first place. Indeed, the possible results of such an approach would be that the gradient descent is fast in terms of needed update steps to reach a local minimum, but the sheer time needed to even make one update step is so long that all other presented approaches would be faster.

In preliminary work, we entertained the idea that a sample's *utility*, or *importance* is tightly coupled with the performance of our model when operating on said sample. Upon performing literature review we also find that others before us also consider this to be the case (see Katharopoulos and Fleuret [2018]; Jiang et al. [2019]). The idea presented is that the utility of a sample can be inferred from performance measures like *loss*, *gradient norm*, or *rank* of either of the two.

$$U(x) \sim P(x)$$

x	: data sample(s)
U	: Utility
P	: Performance

Where P is either of the aforementioned performance measures or their inversion (high-loss samples have higher importance than low-loss samples for example). We suggest that in classification tasks, there is additionally the binary *accuracy/error* value (correctly or incorrectly classified). Accuracy and error, however, being binary values, cannot directly be used to select *the most important*.

Another, perhaps more likely modelling of the relation, although more up for interpretation is the following:

$$U = f \circ P \qquad \left| \begin{array}{ll} f & : \text{an unknown function} \\ U & : \text{Utility} \\ P & : \text{Performance} \end{array} \right.$$

Suggesting that there is a more complex relation in play - that *utility* presents itself as a compound function, perhaps dependent on a multitude of properties of the sample and how the model performs when presented with said sample.

Luckily for us, we're not trying to find a sampling strategy that only selects the samples of high utility. What we are trying, is to find a sampling strategy that select more samples of utility that the standard mini-batch SGD archetype approach does. And this task is comparatively much easier! We mentioned earlier that selecting the *most important samples* is not possible directly through the metric of accuracy. However, we can assume — considering the relation described above — that inaccurately classified samples generally are of higher utility than accurately classified samples.

A potential benefit of selecting such a non-specialised metric to decide which samples to include in training, is that we might circumvent the overfitting Song et al. [2020] present to often be a result of training on designed selections over time, seeing as our data selection then has the potential to be more varied than that of a loss- or gradient-focused selection.

Our suggested approach is thus to only sample inaccurately classified samples, as they are assumed to have higher utility than that of a stochastically sampled samples.

3.2 Overthrowing the Archetype

From the archetype we identify that the region we want to modify to introduce adaptive sampling is the how data is fed to the neural net. As our selection mechanism is dependent on the current state of the model, we first need to feed all samples to it to determine which to accept, and which to reject. Then, when all accepted samples (i.e. the inaccurately classified ones) are found, they are passed through the model yet again, this time to calculate the losses and gradients. Finally, we use the gradients in the gradient descent update rule (see eq. (2.2)), and move on to the next batch.

3.2.1 Preliminary Work — The Search for Selectors

In the preliminary phase of the thesis, we explored a wide range of selectors, many of which were novel in their own right, or novel variations of existing methods. Before we get to our final architecture and approach we use in our experiments, we present some of our preliminary work that we did to explore the field and gather ideas, ultimately leading us to where we ar now. Many of which we see potential in as further discussed in section 5.4 Future Work.

The fundamental idea we started out with in our exploratory phase for this thesis was the idea of adaptive selections; using the model's own properties to determine what to train on.

We were exploring opportunities primarily using MNIST, leading us to working with the baseline basically being BGD, as everything could fit easily in one batch on GPU. This lead to some variants being difficult to apply in later iterations as larger datasets generally need to be split into smaller mini-batches, and worked on using either SGD or MBGD.

Algorithm 2
Training Loop in Preliminary Work. ▷ This is essentially BGD

Require: A Dataset; $\boldsymbol{\mathcal{S}}$.
Require: A Loss Function; $\boldsymbol{\mathcal{C}}(\mathcal{Y}, \hat{\mathcal{Y}})$.
Require: An Optimiser; **Op**.
1: **procedure** TRAIN(net, $epochs$)
2: **for** $e \leftarrow 1$ **to** $epochs$ **do**
3: $\mathcal{B}_{data}, \mathcal{B}_{labels} \leftarrow Select(\boldsymbol{\mathcal{S}}; net)$ ▷ In BGD, *Select* returns $\boldsymbol{\mathcal{S}}$
4: $predictions \leftarrow net.Forward(\mathcal{B}_{data})$
5: $losses \leftarrow \boldsymbol{\mathcal{C}}(predictions, \mathcal{B}_{labels})$
6: $gradients \leftarrow net.Backward(losses)$
7: **Op**$.step(net, gradients)$ ▷ Adjust *net* parameters
8: **end for**
9: **end procedure**

This search for a new method of sampling quickly went to *losses*, as they are readily accessible as part of the training loop already. Initially, the work revolved around the idea that the loss could be used directly for an informed selection.

Top-k Selector Our first iteration of a selection mechanism was the top-k selector. The essence of it to simply sort the samples on their loss value, then pick the k samples of highest loss.

Algorithm 3
Select (Loss Sorted Top-k)

Require: A neural net model; *net*.
Require: A Loss Function; $\boldsymbol{\mathcal{C}}(\mathcal{Y}, \hat{\mathcal{Y}})$.
function SELECT($\boldsymbol{\mathcal{S}}$)
  $\mathcal{X}, \hat{\mathcal{Y}} \leftarrow \boldsymbol{\mathcal{S}}$
  $predictions \leftarrow net.Forward(\mathcal{X})$
  $losses \leftarrow \boldsymbol{\mathcal{C}}(predictions, \hat{\mathcal{Y}})$
  $idx_{desc} \leftarrow \text{argsort}(losses)$ ▷ Sort indices on losses, descending
  $\mathcal{S}_{\#} \leftarrow \boldsymbol{\mathcal{S}}[idx_{desc}[0:k]]$ ▷ Select k top elements of $\boldsymbol{\mathcal{S}}$
  return $\mathcal{S}_{\#}$
end function

Top-q Selector Other variants of this were quantile-based selectors, with q being of the interval $(0, 1)$ (normally set to values like 0.5), which selected all samples above said quantile. Using it meant giving up control of number of samples for a more flexible selector taking the relative sample losses into account. This would sometimes lead to noisy learning, as the non-static size of batch size effectively made learning rate have less control of the step size taken in gradient descent.

Loss Weighted Random Selector The loss weighted sampler bears resemblance to *top-k*, and shares the same basic structure as algorithm 3. With the fundamental difference that it uses a weighted random selection instead of the top k elements of the sorted loss. The sorted loss is instead utilised as the to make a probability density function for weighting the selection. This in turn, is similar to the proposed selection mechanism proposed by Jiang et al. [2019], with the primary difference being that their approach was defined to accumulate batches, while we were still performing selection on the whole dataset, putting us closer to a filtered variant of BGD.

Loss q-Split Random Selector This loss-based sample selector again is similar to the previous selectors, with it, we aim to divide a cumulative loss-power into equally sized portions, and from that take equally many samples from each portion. Doing so with a high power **p** would result in making a **q**th part of the selection from a small number of high-loss samples. Doing so with a low power **p** would result in a more equal spread of selected samples in terms of losses.

An example case of using this: If we've set $\mathbf{q} = 2$ and $\mathbf{p} = 3$, the cumulative losses would in the last couple of elements raise very quickly, seing as that's where the high-loss samples reside after sorting. Having $\mathbf{q} = 2$ in this case would make half of the selection among these last couple of samples, while the other half would be spread equally among the low losses and middle losses.

Algorithm 4
Select (Loss q-Split Random Selector)

Require: A neural net model; net.
Require: A Loss Function; $\boldsymbol{\mathcal{C}}(\mathcal{Y}, \hat{\mathcal{Y}})$.
Require: A power factor; $\mathbf{p} \in (0, \infty)$. ▷ typically 2 or higher
Require: A number of buckets; $\mathbf{q} \in [2..\infty)$. ▷ typically 2
Require: A number of samples; $\mathbf{k} \in [2..\infty)$. ▷ typically 2

function SELECT($\boldsymbol{\mathcal{S}}$)
  $\mathcal{X}, \hat{\mathcal{Y}} \leftarrow \boldsymbol{\mathcal{S}}$
  $predictions \leftarrow net.Forward(\mathcal{X})$
  $losses \leftarrow \boldsymbol{\mathcal{C}}(predictions, \hat{\mathcal{Y}})$
  $idx_{asc} \leftarrow \text{argsort}(losses, -1)$ ▷ Sort indices on losses, ascending
  $cuml \leftarrow \text{cumsum}(losses[idx_{asc}]^p)$ ▷ cumulative losses
  $buckets \leftarrow [[] * \mathbf{q}]$ ▷ make q empty buckets
  $bs \leftarrow \text{last}(cuml) \,/\, \mathbf{q}$ ▷ bucket size
  $b_i \leftarrow 0$
  for all i **in** idx_{asc} **do** ▷ Assign indices to their buckets
    $buckets[b_i].append(i)$
    if $cuml[i] \geq bs * (b_i + 1)$ **then**
      $b_i \leftarrow b_i + 1$
    end if
  end for
  $idx_{select} \leftarrow []$ ▷ Fill by equally many samples from each bucket
  for all j **in** $[0..q]$ **do**
    $idx_{select}.append(SelectRandom(buckets[j], k \,/\, \mathbf{q}))$
  end for
  $\mathcal{D}_{\#} \leftarrow \boldsymbol{\mathcal{S}}[idx_{select}]$
  return $\mathcal{D}_{\#}$
end function

An interesting side-note to add about **p** is that it could hypothetically be decaying over time to reduce selection pressure, similar to what Song et al. [2020] do in order to circumvent potential overfitting issues related to non-uniform sampling.
From here we saw that although the different solutions seemed to work, they were getting ever more convoluted and introduced new hyperparameters. So we went back to the simplest variant, the top-k selector, and rather tried to make changes to that. This turned out to be the birth of the filtering-concept

Although most of the selectors also can be seen as filtering mechanisms, they themselves rely on a dataset. And there is nothing stopping us from doing some simple filtering on said dataset before we apply the selectors. Hence, we came up with some filtering schemes too:

Hard Accuracy Filtering The simplest of them all, and closest to the main experimental focus of this thesis is the *hard accuracy filter*, where we first filter out samples the network is already able to handle, and then prioritise which samples to train on based on a k.

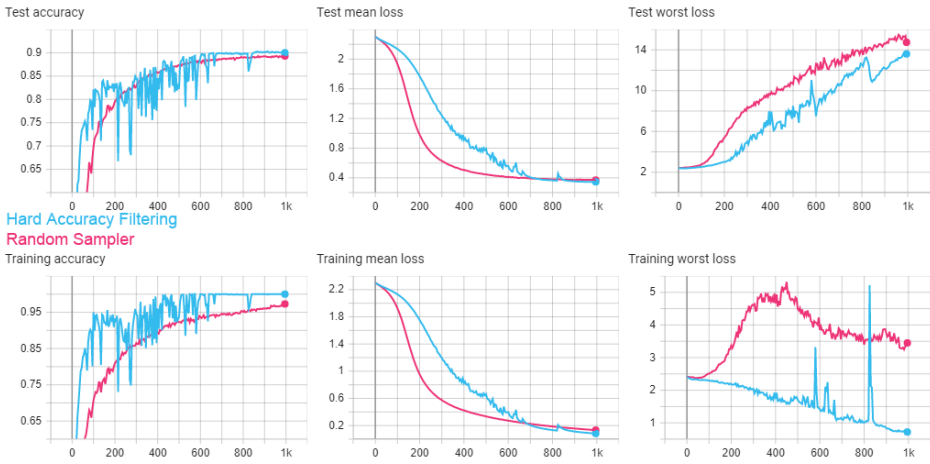


Figure 3.1: An example run where Hard Accuracy Filtering (algorithm 5) was utilised. Showing promising results on MNIST with a simple 3-layer fully connected network.

Algorithm 5
Select (Hard Accuracy + top-k)

Require: A neural net model; *net*.
Require: A Loss Function; $\boldsymbol{\mathcal{C}}(\mathcal{Y}, \hat{\mathcal{Y}})$.
function SELECT($\boldsymbol{\mathcal{D}}$)
  $\mathcal{X}, \hat{\mathcal{Y}} \leftarrow \boldsymbol{\mathcal{D}}$
  $predictions \leftarrow net.Forward(\mathcal{X})$
  $filter \leftarrow \text{argmax}(predictions) \neq \hat{\mathcal{Y}}$
  $predictions_{\#} \leftarrow predictions$ **where** $filter$
  $\mathcal{D}_{\#} \leftarrow \boldsymbol{\mathcal{D}}$ **where** $filter$
  $\hat{\mathcal{Y}}_{\#} \leftarrow \mathcal{D}_{\#}[1]$ ▷ Select filtered labels
  $losses \leftarrow \boldsymbol{\mathcal{C}}(predictions_{\#}, \hat{\mathcal{Y}}_{\#})$
  $idx_{desc} \leftarrow \text{argsort}(losses)$ ▷ Sort indices on losses, descending
  $\mathcal{D}_{\#} \leftarrow \mathcal{D}_{\#}[idx_{desc}[0:k]]$ ▷ Select top k from the filtered subset of $\mathcal{D}_{\#}$
  return $\mathcal{D}_{\#}$
end function

Obviously, one catch with this approach is the potential problem of not having enough samples to take from after having filtered. Depending on k this might be a problem or not. In larger datasets like CIFAR-10 and CIFAR-100. k needs to be somewhat large to encounter such a problem. Nonetheless, it is a real problem, and it is a problem that is actually nice to encounter, seeing as that would mean the network has learnt most of the samples of the dataset.

It should be considered however, before employing such a solution. Our go-to option in cases where we encountered this problem (which, given that we were using MNIST was somewhat more often) was to simply select random samples from the dataset, essentially defaulting back to SGD.

Soft Accuracy Filtering An iteration of the filter-idea from there was the soft accuracy filter. The grounds for this idea came from a concern that hard accuracy filtering potentially ends up having samples oscillating out and in of the set of samples the model can handle.

Soft accuracy utilises the prediction of the neural network to determine not only if a sample is correctly classified, but also that it's well within some margin of the class compared to the other classes. Essentially, it strives push its decision boundary a bit further than the bare minimum for a correct classification on the training dataset. Doing so would achieve similar effects to that of [Lin et al., 2017], in which an additional term is added to the cross-entropy loss, reducing the loss on samples that are well within the decision boundary. (see appx. fig. 8)

Algorithm 6
Select (Soft Accuracy + top-k) ▷ Changes from Hard Accuracy highlighted

Require: A neural net model; net.
Require: A Loss Function; $\boldsymbol{\mathcal{C}}(\mathcal{Y}, \hat{\mathcal{Y}})$.
Require: A margin; $\boldsymbol{\alpha} \in (0, 1)$
function SELECT($\boldsymbol{\mathcal{D}}$)
  $\mathcal{X}, \hat{\mathcal{Y}} \leftarrow \boldsymbol{\mathcal{D}}$
  $predictions \leftarrow net.Forward(\mathcal{X})$
  $filter \leftarrow \text{argmax}(predictions) \neq \hat{\mathcal{Y}}$
  $cdx \leftarrow \text{softmax}(predictions)$ ▷ *Belief* distributions per sample
  $idx \leftarrow [0..len(\boldsymbol{\mathcal{D}})]$ **where not** $filter$ ▷ Get indices of accurately classified
  $marginal \leftarrow []$
  for all i **in** idx **do**
    $p, q \leftarrow \text{sort}(cdx[i])[0:2]$ ▷ Get 2 highest *beliefs* of sample
    if $p - q \leq \boldsymbol{\alpha}$ **then**
      $marginal.append(\boldsymbol{\mathcal{D}}[i])$
    end if
  end for
  $predictions_{\#} \leftarrow predictions$ **where** $filter$
  $\mathcal{D}_{\#} \leftarrow \boldsymbol{\mathcal{D}}$ **where** $filter$
  $\hat{\mathcal{Y}}_{\#} \leftarrow \mathcal{D}_{\#}[1]$ ▷ Select filtered labels
  $losses \leftarrow \boldsymbol{\mathcal{C}}(predictions_{\#}, \hat{\mathcal{Y}}_{\#})$
  $idx_{desc} \leftarrow \text{argsort}(losses)$ ▷ Sort indices on losses, descending
  $\mathcal{D}_{\#} \leftarrow \mathcal{D}_{\#}[idx_{desc}[0:k]]$ ▷ Select top k from the filtered subset of $\mathcal{D}_{\#}$
  $\mathcal{D}_{\#}.appendAll(marginal)$ ▷ Add all soft filtered samples
  return $\mathcal{D}_{\#}$
end function

3.2.2 *Keep It Simple Stupid* — Our Main Approach to Selection

As mentioned in the previous section, we got to a point in our preliminary work where we found our selectors to be too convoluted, and too complex. They were ideas combining concepts we yet had to reason about on the individual level, considering what each concept brought to the selection. Hence; we take a step backwards in complexity, and follow the *KISS (Keep It Simple Stupid)* principle, when we now pose the idea of reducing the complexity of our selection.

We move from having introduced a filtering mechanism as an addition to our loss-oriented sample selectors, to instead discard the loss from our selection process entirely. Our motivation for doing so lies in the aforementioned uncertainty about what accuracy as a metric gives us; we want to discover it contribution, while also suggesting that the accuracy metric might yet prove sufficient on its own as a means to perform sample selection, as it is not as susceptible to high-loss outliers as a loss-oriented approach may be (meaning that it doesn't prefer the high-loss erroneous samples over the low-loss erroneous samples, which a loss-oriented one might).

Previous methods have already indicated a benefit of non-stochastic selection of samples utilising losses or gradients [Katharopoulos and Fleuret, 2018; Jiang et al., 2019], in addition to our preliminary exploratory experiments. Hence; we can also motivate our step back to the simpler filtering mechanism as it will be the newcomer to the field.

From this point, we look to larger datasets (CIFAR-10 & CIFAR-100), in addition to MNIST, as a selector algorithm ideally should scale to larger machine learning problems too. Not just the relatively simple problem of image classification on MNIST.

Accuracy Filtering The accuracy filtering scheme is very similar to that of algorithm 5, with the exception that we don't need to calculate losses at all.

Algorithm 7
Filter (Accuracy/Error)

Require: A neural net model; *net*.
1: **function** FILTER($\boldsymbol{\mathcal{D}}$)
2: $\mathcal{X}, \hat{\mathcal{Y}} \leftarrow \boldsymbol{\mathcal{D}}$
3: $predictions \leftarrow net.Forward(\mathcal{X})$
4: $\mathcal{D}_{\#} \leftarrow \boldsymbol{\mathcal{D}}$ **where** $\text{argmax}(predictions) \neq \hat{\mathcal{Y}}$
5: **return** $\mathcal{D}_{\#}$
6: **end function**

We insert this into a training loop similar to the Archetype Training Loop (Algorithm 1), modified to perform a filtering operation on the dataset every time a new epoch starts. This will from

3.3 The Problem of Filtering Large Data

A somewhat convoluted problem arise when using the epochwise-filtered training loop (Algorithm 8). Namely; memory requirements. This applies especially when applying the algorithm to larger datasets, as they cannot all be held in memory at once. Strictly speaking, this is not the case with the datasets we used, as they are relatively small in the grand scheme of things, but it's nonetheless something we considered when implementing the algorithm.

With some datasets consisting of samples of a vastly larger size than that of CIFAR-10, such as ImageNet which (admittedly has varying image sizes, but) has a rough estimate of images'

Algorithm 8
Epochwise Filtering Training Loop. ▷ Changes from `Archetype (Algorithm 1)` highlighted

Require: A Dataset; $\boldsymbol{\mathcal{S}}$.
Require: A Loss Function; $\boldsymbol{\mathcal{C}}(\mathcal{Y}, \hat{\mathcal{Y}})$.
Require: An Optimiser; **Op**.
procedure TRAIN(net, $epochs$)
  for $e \leftarrow 1$ **to** $epochs$ **do**
    $\boldsymbol{\mathcal{D}} \leftarrow StochasticBatch(\boldsymbol{\mathcal{S}}, len(\boldsymbol{\mathcal{S}}))$ ▷ Shuffled dataset
    $\boldsymbol{\mathcal{D}}_{\#} \leftarrow Filter(\boldsymbol{\mathcal{D}}; net)$ ▷ Select samples of high utility
    for all $\mathcal{MB}_{data}$, $\mathcal{MB}_{labels}$ **in** $Batch(\boldsymbol{\mathcal{D}}_{\#}, size_{\mathcal{MB}})$ **do**
      $predictions \leftarrow net.Forward(\mathcal{MB}_{data})$
      $losses \leftarrow \boldsymbol{\mathcal{C}}(predictions, \mathcal{MB}_{labels})$
      $gradients \leftarrow net.Backward(losses)$
      Op.$step(net, gradients)$ ▷ Adjust net parameters
    end for
  end for
end procedure

average size being 400×350, 24-bit (3_B) *true color* color pixels. Oftentimes, the images are pre-processed and re-scaled to a size such as 256×256, meaning we get a total of $3_B \times 256 \times 256 = 192_{KiB}$ per sample. Assuming we were to work on a portion of the whole dataset; let's say $1\,000\,000$ training samples, we would be looking at $192_{KiB} \times 1\,000\,000 \approx 183_{GiB}$, which is beyond any normal amount of RAM, and especially an normal amount of VRAM accessible for doing training on GPU.

If we try to do epochwise filtering on such a large set of large samples, we would end up overstepping our available memory. Not only do we need to load all the samples in the first place — albeit, we can do that batchwise — but we would need to store the *filtered data* too, before we perform a training step on it which, again would be a massive amount of data.

Having considered this problem, we naturally looked at possible solutions too, of which we found two:

3.3.1 Storing Indices of Filtered Data

One of the easiest ways to alleviate the needed memory when storing filtered data, is to instead store the indices of said data. Then, upon fetching a mini-batch of filtered data, one would fetch `batch_size` stored indices, and then simply perform a lookup from the dataset directly.

This solution is good, and works well to mitigate the memory issues, as an index is typically either 4_B or 8_B in size, compared to the 192_{KiB} described above, and we see how much more scalable this solution is. However, a problem quickly arise with this approach too.

Data Augmentation Modern approaches for image classification, and classification in general often rely heavily on *data augmentation*, a process in which we transform the sample data (in image classification problems being the image) to an randomly augmented version of the same data, essentially creating more training data than we have accessible through the dataset. Obviously this could simply be seen as an extension of the dataset, which would still work fine if we were storing aside the data itself for training.

Once we store aside the index instead however, we will augment the sample data differently from the data we filtered on, which, depending on the severity of the augmentation will cause the training loop to ignore augmented samples it may still be unable to classify correctly.

Furthermore, seeing as the data we filter on and the data augmented post-filtering is different when storing the index, we end up in a scenario where the index is marked as an unlearnt sample, because the augmentation done before filtering resulted in a sample our model was unable to classify correctly, while the augmentation done upon training may result in a sample our model is capable of handling.

The problem is framework specific, so this may not apply for all implementations, and might yet be circumvented by manually doing augmentation in the training loop. It is however an issue worth mentioning, as our implementation of epochwise filtering don't take augmentation into account, precisely because it doesn't store aside the data, but rather the indices.

3.3.2 Not Filtering Epochwise; Batchwise Filtering

While epochwise filtering might be beneficial in the sense that the added overhead is minimal, it's also a solution that doesn't explore the potential of filtering to its full extent. — Why not filter for all batches instead? That way we can always ensure that the samples the model is training on is (according to our selection scheme) the samples of most utility.

Using a batchwise filtering mechanism instead alleviates us from all of the resource-related issues, as we now can keep smaller batches in memory instead. With that, it also circumvents the issues related to data augmentation we saw in epochwise filtering. We simply keep the data for when we're actually training the model, post-filtering, and as such; we can always be sure that the data presented to the model is data controlled by us through our sampling scheme.

There are two ways in which we could implement a batchwise filtering, both of which have their own catch to them:

Algorithm 9
Adaptive Sampler Training Loop - Batchwise (Unstable batch size)

Require: A Dataset; $\boldsymbol{\mathcal{S}}$.
Require: A Loss Function; $\boldsymbol{\mathcal{C}}(\hat{Y}, Y)$.
Require: An Optimiser; **Op**.

procedure TRAIN(net, $epochs$)
  for $e \leftarrow 1$ **to** $epochs$ **do**
    for all $\mathcal{MB}$ **in** $StochasticBatch(\boldsymbol{\mathcal{S}}, size_{\mathcal{MB}})$
      $\mathcal{MB}_{\#} \leftarrow Filter(\mathcal{MB}; net)$
      $\mathcal{MB}_{data}$, $\mathcal{MB}_{labels} \leftarrow \mathcal{MB}_{\#}$
      $predictions \leftarrow net.Forward(\mathcal{MB}_{data})$
      $losses \leftarrow \boldsymbol{\mathcal{C}}(predictions, \mathcal{MB}_{labels})$
      $gradients \leftarrow net.Backward(losses)$
      Op$.step(net, gradients)$ ▷ Adjust *net* parameters
    end for
  end for
end procedure

The First one is quite simply receiving a batch of n size from the dataset, performing the filtering operation, and then using this subset of filtered data as its batch.

The catch with this one is that the resultant filtered batch has a varying size. Throughout training the effective batch size would end up becoming smaller and smaller, leaving the few samples left in the batch to have a larger impact on the gradient than any one sample normally would have when a batch size is set to be stable, effectively overriding the set learning rate. We imagine this could lead to unstable progression, and instead end up opting for option two:

Algorithm 10
Adaptive Sampler Training Loop - Batchwise

Require: A Dataset; $\boldsymbol{\mathcal{S}}$.
Require: A Loss Function; $\boldsymbol{\mathcal{C}}(\mathcal{Y}, \hat{\mathcal{Y}})$.
Require: An Optimiser; **Op**.
Require: A raw batch sampling size; $\boldsymbol{size_{\mathcal{RB}}}$.
Require: An ideal filtered batch size; $\boldsymbol{size_{\mathcal{FB}}}$.

procedure TRAIN(net, $epochs$)
  $\mathcal{FB} \leftarrow []$ ▷ Initialise empty filtered samples mini-batch
  for $e \leftarrow 1$ **to** $epochs$ **do**
    for all $\mathcal{RB}$ **in** $StochasticBatch(\boldsymbol{\mathcal{S}}, \boldsymbol{size_{\mathcal{RB}}})$ **do**
      $\mathcal{RB}_{\#} \leftarrow Filter(\mathcal{RB}; net)$
      $\mathcal{FB}.append(\mathcal{FB}_{\#})$
      if $len(\mathcal{FB}) \geq \boldsymbol{size_{\mathcal{FB}}}$ **then**
        $\mathcal{X}, \hat{\mathcal{Y}} \leftarrow \mathcal{FB}[0 : \boldsymbol{size_{\mathcal{FB}}}]$ ▷ Select $\boldsymbol{size_{\mathcal{FB}}}$ samples
        $predictions \leftarrow net.Forward(\mathcal{X})$
        $losses \leftarrow \boldsymbol{\mathcal{C}}(predictions, \hat{\mathcal{Y}})$
        $gradients \leftarrow net.Backward(losses)$
        Op$.step(net, gradients)$ ▷ Adjust net parameters
        $\mathcal{FB} \leftarrow []$ ▷ Empty filtered samples for next batch
      end if
    end for
  end for
end procedure

The second option is to accumulate filtered batches over a number of passes getting raw data from the dataset. This would look a lot like an archetypal training loop, with the exception that we put filtered the raw batches (i.e. the unfiltered data) instead of training on them. We only do the forward pass (see eq. (2.2)), and from there evaluate the accuracies to accept or reject samples into the filtered batch. Once the batch is filled, we perform an update step, and reset the accumulated batch.

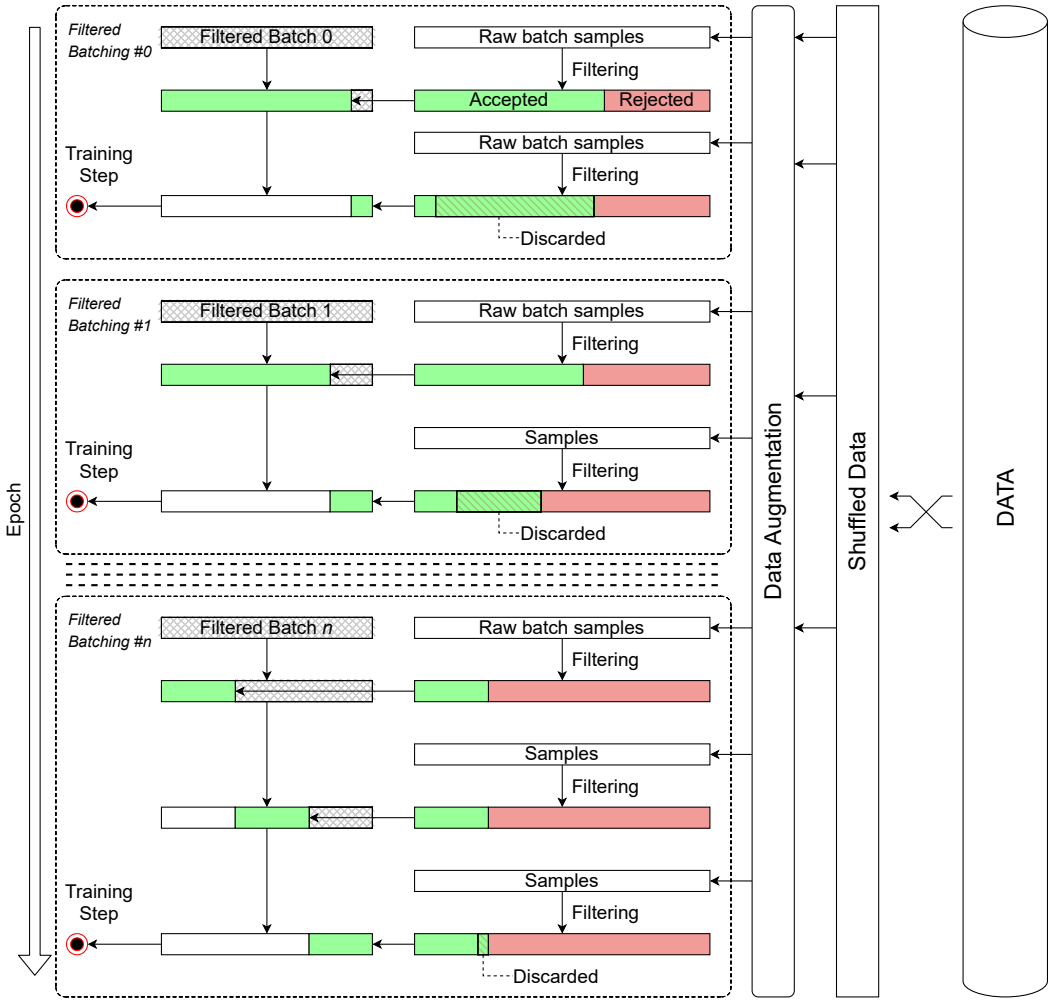


Figure 3.2: Accumulative Batching of Filtered Data

Batch Accumulation is what we have dubbed this second option. We want our trainer algorithm to serve data in batches, and perform MBGD on them. We want these batches to be carefully selected. When doing this careful selection (i.e. filtering) of which samples to include in a batch, we end up rejecting some samples from being included in the batch. To mitigate the issue of variable batch size, we adapt an accumulative way of building our batches. Inspired by Jiang et al.'s [2019] accumulation of samples before performing updates, we too iterate over the dataset and filter out samples that should not be part of the current batch.

For computation speed, it is ideal if we emplace as much as possible from the dataset into

one batch. Doing so will allow us to fully utilize data broadcasting and vectorized operations, which most modern computers — and certainly those designed to be machine learning capable — generally do, and draw great performance-boosts from. However, emplacing more data into one raw batch may lead to a large batch overflow, which ultimately ends up being work that's just discarded.

Filtered Sample Overflow is an issue our current implementation of the accumulator does not take into account. By that we mean that unfortunately, the sampling process ends up doing more work than what is ultimately used.

Raw data that endure filtration is not guaranteed to be included in the filtered batch. When a filtered batch is completed there may yet be samples that were filtered, but that didn't fit into the batch. Seeing as we once again want our batches to be of a stable size, we simply discard the overflown accepted samples. This fortunately is a diminishing drawback; as the acceptance ratio decreases continually through training, the overflow grows smaller and smaller (This is illustrated in appx. fig. 6b).

The Semantic Problem of *"Epochs"* is one that arise when we introduce filtering to the selection process. Specifically, the term *epoch* as is, is defined as a cycle over all the samples of the training dataset. How are we to define when our filtering reaches the end of an epoch, when we practically never cycle through all the data of the training dataset?

In the epochwise filtering scheme, this isn't all that big of an issue, as we could look at the filtered dataset to be the one we cycle over. Equally, when filtering batchwise with an unstable batch size (Algorithm 9), we can easily consider a cycle as a having filtered all the samples from the training dataset.

However, when we use the stable batch size implementation (Algorithm 10), we end up in a situation in which we might have varying number of training steps being performed per epoch. Additionally, as it is most efficient for building up our filtered batch, a filtered batch may contain samples from several different epochs, and several epochs may pass without a training step being performed. If the batch size is set to a large number, and the filter's acceptance ratio is growing low, this can easily become the case. (All of these effects can be seen illustrated in appx. fig. 6)

3.4 Final Architecture

Our final architecture, and the architecture we refer to from this point on, will primarily be batchwise filtering with stable batch size (i.e. unstable number of training steps per epoch). And Epochwise filtering.

For both we have the option of using data augmentation. For batchwise filtering we can guarantee a "hard filtering", as the augmented samples we accept through the filtering also will be the same sample we train on. For epochwise filtering however, turning data augmentation on results in a "soft filtering" in which an augmented sample accepted through the filter will end up being augmented differently when sampling it again for the training.

IV
Experiments & Assessment

We will now go over the experiments we ran in detail, identifying their purpose, as well as the procedure of how we conducted them. We will present the metrics of the experiments, identifying which data are measured and tracked, as well as a description of the variables of the experiments.

We will present the hardware and software used to conduct and analyse the experiments and their results, as well as initial parameters.

Finally, we move on to present the results and evaluate them.

4.1 Experiment Plan

We have built a novel way of selecting samples for mini-batch SGD. Our initial motivation for making a new selection procedure was to reveal whether it could improve performance and limit overfitting. As such, we focus our experimentation efforts to assess a large set of configuration combinations for our novel approach. Not only that, but we also ensure that for every one configuration using our approach, we have an equivalent archetype training loop running alongside using the same set of parameters.

It has to be emphasised again that although we are comparing our accuracy-filtering approach to that of an archetypal mini-batch SGD, we are not focusing our effort on exceeding it in terms of accuracy in any way. We are not looking to outperform the state-of-the-art, but rather to discover niches, benefits, drawbacks; indeed, overall behaviour in a multitude of hyperparameter configurations. Our ultimate goal, after all, is to find if our novel approach in any respect is valuable and if using a designed selection like this can have useful effects.

With that being said, it should also be noted that we also have runs using state-of-the-art configurations. Yet again to see if we too benefit from said configurations or not. These runs can be observed in appendix II.

To a great extent, our experiments are conducted exhaustively, attempting to isolate the

effects of our additions and open for the eventuality that our additions combined with other optimisations may prove favourable or unfavourable.

Concretely, we are tracking the following metrics and poll their values once per epoch:

Wall-Clock Time *How much time does the training loop use to get through all epochs.* — A lot of the related works found that non-stochastic sampling can lead to reaching higher accuracies faster in a set time frame [Katharopoulos and Fleuret, 2018; Song et al., 2020; Jiang et al., 2019]; hence, we look for this behaviour when using our sampling method too.

Accuracy *How many of the samples given has the model correctly classified per epoch.* — Naturally, if we were to outperform the archetype, it would be safe to say that something interesting is happening. It's a no-brainer to keep track of how the accuracy of models subdued to our new sampler.

Mean Loss *The mean of all samples' classification losses, defined by cross entropy in our experiments.* — As are looking at the effects of our approach in terms of overfitting, one of the crucial metrics to track is mean loss, as it is perhaps the best metric to identify overfitting with.

Worst Loss *The loss of the one sample our model performs worst on.* — The mean loss is not the only loss metric of interest however! The worst loss-metric will help us understand the generalisation ability of the model.

Quantile $\mathbf{q = 0.9}$ Loss *The loss of quantile samples.* — In addition to the mean loss and worst loss, a metric we track is the loss of quantile $q = 0.9$. Doing so might again help us understand the generalisation capabilities, but will also allow us to make rough estimates of how our losses in general are.

In addition, we track the following metrics which have been left out in plots and tables for brevity, and because they don't add much of interest:

Best Loss	**Quantile $\mathbf{q = 0.5}$ Loss**
Quantile $\mathbf{q = 0.99}$ Loss	**Process GPU Temp**
Process GPU Power Usage	**Process GPU Memory Allocated**
Process GPU Temp	**Process GPU Utilization**
Process CPU Threads In Use	**Process Memory Available**
Process Memory In Use	**CPU Utilization**

All of the runs of our experiments are set up to adhere to the same set of hyperparameters. Unless explicitly stated otherwise, they follow the setup illustrated in table 4.1

Configurable Parameter	Value	Parameter Purpose
`model`	`ResNet18`	The type of model are we training.
`optimizer`	`SGD`	Which optimizer to use for the run.
`batch_size`	`1024`	The size of the mini-batch used for training.
`learning_late`	`0.1`	The initial learning rate the `optimizer` should use.
`data_augmentation`[1]	`True`	Should the run use augmentation on sampled data from the dataset.
`rng_seed`	`0`	The random seed for the Pseudorandom number generator (PRNG) used in the run.
`weight_decay`	`0.0005`	At what rate should the weight parameters of the model be decaying. (`0` ⇒ no decay).
`epoch_filtering`	`False`	Make an accuracy-filtered dataset selection at the start of epochs.
`batch_filtering`	`True`	Should the run Create accuracy-filtered batches.
`lr_scheduler`	`None`	Learning rate scheduler (available choices are `exponential`[2], `multistep`[3] and `None`).
`accumulation_batch_size`[4]	`None`	The size of raw batches used in batch accumulation (`None` ⇒ `batch_size`).
`momentum`[5]	`0.9`	Add this proportion of the last update vector to the current update vector.
`nesterov`[5]	`False`	Should the run use nesterov accelerated gradients.
`amsgrad`[6]	`False`	Should the run use amsgrad, maximizing the squared gradients term used in the Adam update equation.[Ruder, 2016]
`gamma`[7]	`0.1`	Decay factor deciding how much the learning rate should change (Defaults to `0.1` using `multistep`, `0.995` using `exponential`).

Table 4.1: Baseline parameters used in all experiments.

Our experiments can be seen as one big experiment with separate trials for various parameter configurations. However, within the parameter combination exploration, some sets of combinations can easily be considered distinct to the rest of the runs. To make it easier to process and present the data, we pick out a selection of these runs with the intent to present any findings of trends seen in these, and findings specific to those runs. For a full overview of the experiments we ran, see Appendix II.

[1] Only applies for runs using CIFAR-10 or CIFAR-100
[2] `https://pytorch.org/docs/stable/optim.html#torch.optim.lr_scheduler.ExponentialLR`
[3] `https://pytorch.org/docs/stable/optim.html#torch.optim.lr_scheduler.MultiStepLR`
[4] Only applies to runs using `batch_filtering`.
[5] Only applies to runs using the `SGD` optimizer.
[6] Only applies to runs using the `Adam` optimizer.
[7] Only applies runs using a `lr_scheduler`.

The experiments are highly exploratory, which could be considered a problem, as it is difficult to know where to start and where to go, but due to the novelty of our approach, we see it as the only option to lay the grounds. By running these exploratory experiments, we intend to find the unknown relations, if any, between the sample filtering and the various metrics listed earlier, and to that extent present data to discuss so we can further answer our research questions.

The selection of experiments we pick out are the following:

MNIST Fully Connected Networks A scenario in which a simple fully connected network often is sufficient to get high scores — how will a model trained using our approach fare?

Modified Parameters For This Experiment	
Parameter	Values
`model`	`[FCNet100, FCNet1000, FCNet3000]`
`data_augmentation`	`False`
`batch_filtering`	`[True, False]`

MNIST Epochwise Filtering Explore how the two implemented filtering modes compare to each other, how they both compare to an archetypal trainer, and how they compare when combined.

Modified Parameters For This Experiment	
Parameter	Values
`model`	`[LeNet, ResNet18, DLA]`
`data_augmentation`	`False`
`epoch_filtering`	`[True, False]`
`batch_filtering`	`[True, False]`

CIFAR-10 Fully Connected Networks CIFAR-10 is a *slightly* more challenging scenario for a fully connected network. Will our addition help in that regards?

Modified Parameters For This Experiment	
Parameter	Values
`model`	`[FCNet100, FCNet1000, FCNet3000]`
`data_augmentation`	`True`
`batch_filtering`	`[True, False]`

CIFAR-10 Batch Size Variations What happens when we adjust batch size, and how will it affect accuracy, not to mention losses?

Modified Parameters For This Experiment	
Parameter	Values
`batch_size`	`[16, 128, 256, 512, 1024]`
`data_augmentation`	`True`
`batch_filtering`	`[True, False]`

CIFAR-100 Fully Connected Networks CIFAR-100 on a fully connected network is practically unheard of, but maybe our filtering might help?

Modified Parameters For This Experiment	
Parameter	Values
`model`	`[FCNet100, FCNet1000, FCNet3000]`
`data_augmentation`	`True`
`batch_filtering`	`[True, False]`

CIFAR-100 Batch Size Variations The same as with CIFAR-10 and batch size variations, but with more classes, and fewer examples per class. How will we fare with less data per class?

Modified Parameters For This Experiment	
Parameter	Values
batch_size	[16, 128, 512, 1024]
data_augmentation	True
batch_filtering	[True, False]

4.2 Experiment Setup

The experiment runs have all had a somewhat involved setup to get going, through iterations however, we have reached this relatively condensed list of information relevant for replicating the works done in this thesis. This will now be covered, when we first look to hardware and software details, how we ran experiment runs on a cluster computer, how the initialisation of variables are set, and which datasets we are employing.

4.2.1 Hardware & Software

Processor	**Cores**	**Memory**	**GPU**
Intel Xeon Gold 6132	28	768GB	NVIDIA Tesla V100 16GB

Table 4.2: Hardware Specifics of the Cluster Node Used.

To run our experiments we used the IDUN-cluster of the HPC-group of NTNU [Själander et al., 2019]. That gave us the control of hardware and software parameters, as well as the needed GPU-capabilities for performing all the experiments we have.

The cluster uses the Slurm Workload Manager to allocate resources and to its users and manage a queue of jobs to be run. Our experiments' Slurm-jobs follows this general format:

1. Request resources:
   - 16 processing cores (for optimal throughput using 4 workers per DataLoader in PyTorch).
   - 32GB Memory (for loading and handling datasets in larger bulks).
   - ∼4 Tasks at a time (for all array-jobs).
2. Load module of needed software and libraries:
   - Python[8] 3.8.6
   - PyTorch[9] 1.7.1
   - torchvision[10] 0.8.1
   - numpy[11] 1.19.2
   - wandb[12] 0.10.26
3. Run python-script with parameters determined by job array-index and predefined parameter list.

[8] https://www.python.org/
[9] https://pytorch.org/
[10] https://pytorch.org/
[11] https://numpy.org/
[12] https://wandb.ai/

4.2.2 Implementation

The majority of our runs are based in batchwise filtering, and both that and the less interesting epochwise filtering should both be reconstructible from the algorithms laid out in algorithms 7 and 10. Initialisation of all runs of an experiment happens upon startup of the run, when it receives the arguments of the run (or defaults if omitted). The initial of all runs have been set up in such a way as to use the same initial PRNG-state, ensuring that although the divergence of random values is inevitable (considering that we have two vastly differing sampling mechanisms), it will always start out the same between runs within an experiment

As a general rule, our experiments run for at least 300 epochs, as that is the region whence we've found most interesting behavioural indicators. That being said, some of the experiments we let run longer, both 500 and 1000 epochs are reasonable for a run to execute within.

Most runs we run on the ResNet18 [He et al., 2015] architecture, modified for use in 32×32-sized images as opposed to the 224×224 of the original implementation. We also have a selection of runs and experiments running on LeNet [LeCun et al., 1998], and state-of-the-art architectures like DLA [Yu et al., 2018]. At the time of writing all implementations of the various networks used can be found online; see *Kuangliu's CIFAR-10 with Pytorch*[13] (except the *FCNet* models, which are all simple one-layer networks of the $in \to n \to out$ format).

4.2.3 Datasets

As stepping stones to show a general tendency in training behaviour when using our custom sampling scheme, we conduct experiments on several datasets of increasing difficulty. Doing so should give us a good idea of the boundaries of our new approach, indicating what works and what doesn't, and in which circumstances.

Dataset	Classes	Training set	Per class	Test set	Per class	Dimensions
MNIST	10	60 000	∼6 000	10 000	∼1 000	28×28
CIFAR-10	10	50 000	5 000	10 000	1 000	32×32×3
CIFAR-100	100	50 000	500	10 000	100	32×32×3

Table 4.3: Overview of datasets used in experiments.

MNIST: Ever since LeCun et al. [1998] first introduced their MNIST dataset, it has paved the way for many a machine learning enthusiasts and researchers. For many, training a classifier on the MNIST dataset is their first serious machine learning project. The dataset consists of images of 10 classes; hand-written digits from 0–9. It comes pre-separated in a training set and a test set — respectively 60 000 and 10 000 samples. The images are 28×28 pixels of 8-bit precision (grayscale). MNIST is relatively popular to use as a stepping stone to explore classification concepts, and put new algorithms to the test, as it is suitably small in terms of complexity, and in terms of required memory, while also being large in terms of samples per class (approximately 6 000 training samples for each of the ten classes), giving a lot of data control to the users of the dataset.[14]

CIFAR-10: From the world of grayscale to the world of colour, that is perhaps the most striking difference between CIFAR-10 and MNIST. Along with the fact that CIFAR-10's classes are images of vastly different objects. Where MNIST keeps to hand-written digits; a relatively small domain, CIFAR-10 expands on this massively. Although CIFAR-10 also only has 10

[13] `https://github.com/kuangliu/pytorch-cifar/`

[14] The dataset is available at `http://yann.lecun.com/exdb/mnist/`

classes, the domain we're working with within CIFAR-10 is massive in comparison. The objects depicted in the different images of the dateset (i.e. the classes) are airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships and trucks. All of which are three dimensional objects, meaning the images (which can represent the objects in two dimensions) can only show part of the objects at a time. Any neural network hoping to appropriately classify these can no longer be of the simple kind, or at least that's the consensus of current state simple fully connected models. The internals of the neural net will need to be able to compress said features and generalize across them.[15]

CIFAR-100: From one relatively complex problem to another, CIFAR-100 takes the difficult parts from CIFAR-10, and ramps them up yet another notch. Where CIFAR-10 has a total of 5 000 samples that can be used to model a class, CIFAR-100 has only 500. In essence, this should mean that a CIFAR-100 classifier needs to be even better in terms of generalization to score high in performance.[15]

Dataset Transforms

As mentioned in section 3.4, we allow both batchwise and epochwise filtering to have batch accumulation be a part of their training loops. We only set up data augmentation for the CIFAR-experiments, as MNIST is trivial to solve with a very high accuracy, even without it. The configuration of our augmentation is as follows:

CIFAR-10 & CIFAR-100

```
RandomHorizontalFlip(),
RandomAffine(degrees=180, translate=(0.1, 0.1), scale=(0.9, 1.1)),
RandomCrop(32, padding=4),
ColorJitter(brightness=0.5, contrast=0.5, saturation0.5, hue=0.05),
```

4.3 Results

We will now present the results from the experiments as listed in section 4.1. For all of the experiments, we will present 9 plots showing the main metrics (covered in section 4.1) of 1-3 runs of a single experiment, in addition to the baseline archetype and its metrics. Further than that, we will also supplement the plots with a condensed-form table, summarising the final value for each of the metrics, and the relevant ratios of m_{test}/m_{train} where m represent the loss and accuracy metrics. This should aid us when looking for signs of overfitting.

Outside the plots and tables we present in this section is also the extensive list of all experiments ran. It can be found in Appendix II.

4.3.1 MNIST Experiments

[15] The dataset is available at `https://www.cs.toronto.edu/~kriz/cifar.html`

Filtering Modes and Network Variants with MNIST

MNIST		Accuracy			Mean Loss			Q=0.9 Loss			Worst Loss		
Model	Filtering	Train	Test	Ratio	Train	Test	Ratio	Train	Test	Ratio	Train	Test	Ratio
LeNet[16]	None	0.999	0.990	0.991	0.005	0.031	6.468	0.001	0.002	1.023	3.272	12.640	3.864
	Epochwise	0.968	0.964	0.995	0.140	0.156	1.113	0.008	0.007	0.890	27.821	30.966	1.113
	Batchwise	0.998	0.988	0.991	0.072	0.086	1.197	0.194	0.195	1.003	2.664	4.684	1.759
	Epoch + Batch	0.953	0.952	0.999	0.213	0.213	1.004	0.547	0.551	1.007	13.197	9.984	0.757
ResNet18[17]	None	0.996	0.988	0.993	0.016	0.044	2.761	0.010	0.010	1.067	8.224	10.766	1.309
	Epochwise	0.979	0.980	1.000	0.070	0.063	0.906	0.029	0.025	0.868	13.521	7.334	0.542
	Batchwise	0.998	0.994	0.995	0.084	0.091	1.087	0.191	0.189	0.987	1.860	3.371	1.813
	Epoch + Batch	0.964	0.967	1.003	0.258	0.255	0.987	0.538	0.526	0.978	6.954	5.182	0.745
DLA[18]	None	1.000	0.996	0.996	0.001	0.016	10.796	0.002	0.002	1.035	0.039	7.763	200.017
	Epochwise	0.989	0.986	0.998	0.039	0.040	1.042	0.003	0.002	0.899	17.521	10.427	0.595
	Batchwise	0.994	0.988	0.994	0.096	0.108	1.128	0.215	0.227	1.054	3.978	3.945	0.992
	Epoch + Batch	0.977	0.977	1.000	0.155	0.154	0.995	0.343	0.338	0.986	8.153	5.790	0.710

Table 4.4: Batchwise and Epochwise Filtering on MNIST with LeNet. In this experiment we vary which network type we use, and explore how the two filtering mechanisms we have presented, namely *batchwise* and *epochwise* filtering perform. Figure 4.1 shows the plots of the 4 emphasised LeNet-runs. We note that independently from the networks used we have the same order from worst to best performer among the different filtering mechanisms, *no filtering* being the top contender. Additionally, we see a trend in mean loss and Q-loss:

$$\texttt{No filtering} < \texttt{Batchwise} \ \& \ \texttt{Epochwise} < \texttt{Epoch + Batch}$$

From this point we more or less discarded usage of Epochwise filtering as an alternative. With this, and several other runs (see appendix II) we found it to be severely worse in practically all respects, and rather prioritized our efforts on experiments using batchwise filtering. For now we take a last note of this experiments: the $loss_{test}/loss_{train}$ ratios are generally close to 1 with batchwise filtering enabled.

Figure 4.1: Batchwise and Epochwise Filtering on MNIST with LeNet, using the four combinations of filtering.

Fully Connected Network on MNIST

MNIST		Accuracy			Mean Loss			Q=0.9 Loss			Worst Loss		
Model	Filtering	Train	Test	Ratio	Train	Test	Ratio	Train	Test	Ratio	Train	Test	Ratio
FCNet3000	None	0.996	0.982	0.986	0.031	0.061	1.959	0.060	0.069	1.142	8.322	6.629	0.797
	Batchwise	1.000	0.968	0.968	0.621	0.623	1.004	0.997	0.998	1.000	1.902	3.135	1.648
FCNet1000	None	0.996	0.982	0.987	0.032	0.062	1.951	0.060	0.068	1.135	8.666	6.462	0.746
	Batchwise	1.000	0.968	0.968	0.605	0.608	1.005	0.984	0.986	1.002	1.932	3.208	1.660
FCNet100	None	0.994	0.978	0.985	0.037	0.069	1.899	0.066	0.076	1.144	9.663	7.282	0.754
	Batchwise	0.999	0.965	0.967	0.483	0.487	1.008	0.878	0.873	0.995	1.885	3.758	1.993

Table 4.5: Fully Connected Networks on MNIST, 3000 nodes hidden layer. Here we see something that at first looked like overfitting when applying our filtering, we clearly seem to have an edge over no filtering in the training data. But again the batchwise sampler fails to similarly when used on the test dataset. We again take note of the better loss ratios in mean and q=0.9-losses.

4.3.2 CIFAR-10 Experiments

[16] LeCun et al. [1998]
[17] He et al. [2015]
[18] Yu et al. [2018]

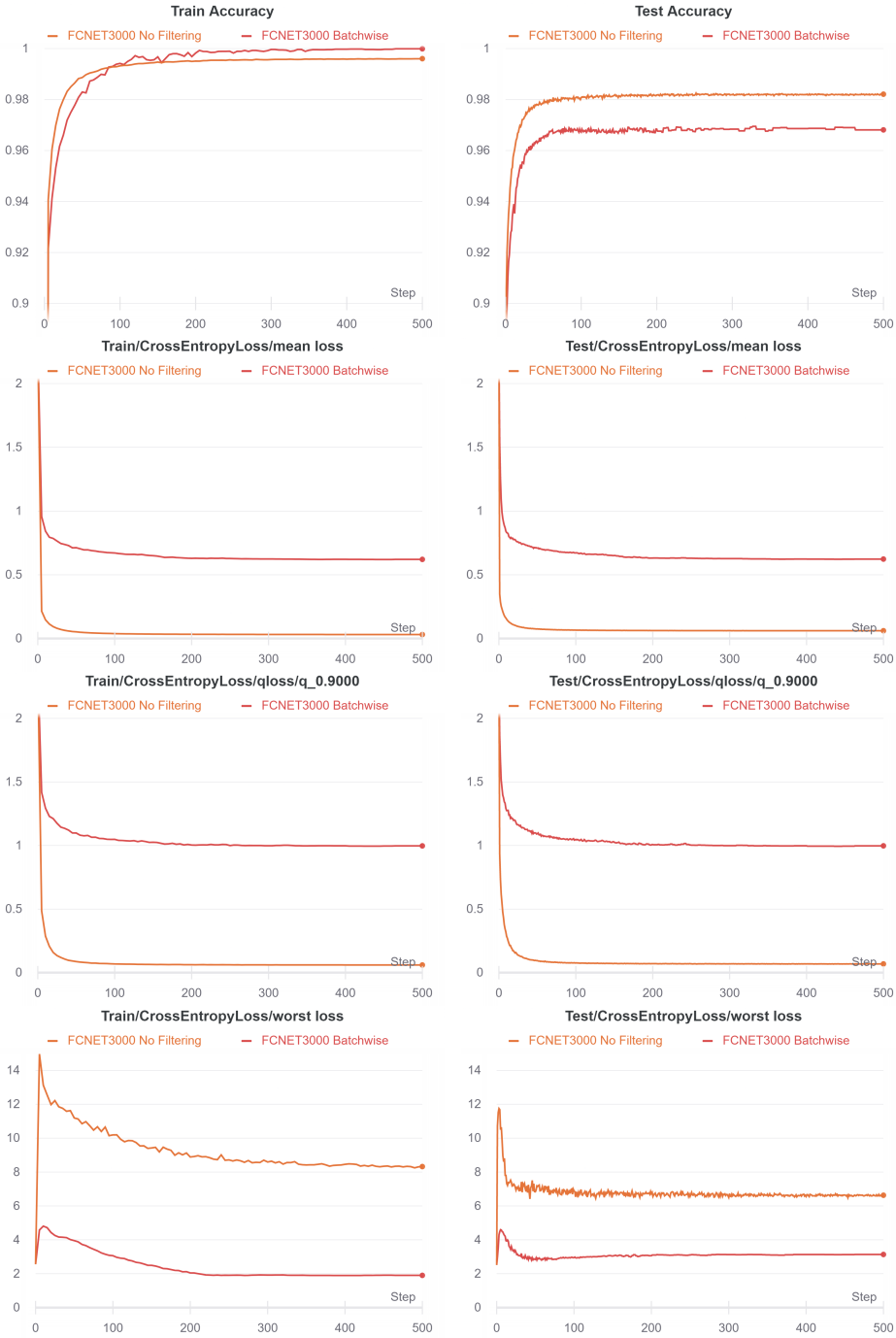
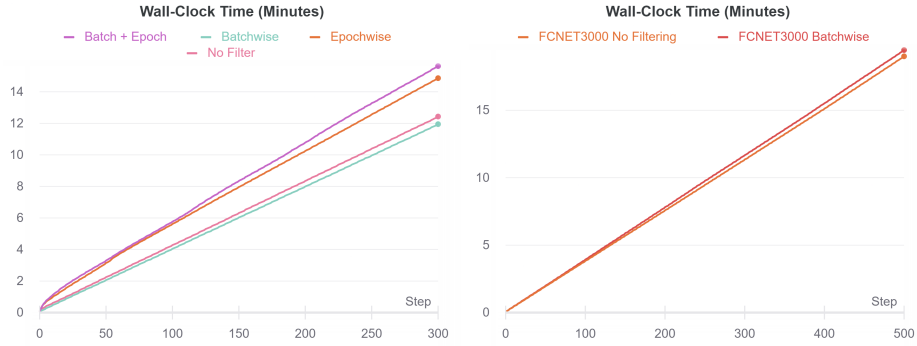


Figure 4.2: MNIST with Fully Connected Network, 3000 nodes in hidden layer.



(a) Batchwise and Epochwise Filtering on MNIST with LeNet. Wall-Clock Time.

(b) MNIST with Fully Connected Network, 3000 nodes in hidden layer. Wall-Clock Time

Batch Size Variations

CIFAR-10		Accuracy			Mean Loss			Q=0.9 Loss			Worst Loss		
Batch Size	Filtering	Train	Test	Ratio	Train	Test	Ratio	Train	Test	Ratio	Train	Test	Ratio
1024	None	0.972	0.849	0.874	0.079	0.582	7.346	0.084	1.993	23.720	11.435	15.933	1.393
	Batchwise	0.885	0.787	0.889	0.579	0.783	1.353	0.875	1.214	1.387	7.834	12.946	1.653
512	None	0.953	0.862	0.905	0.133	0.495	3.712	0.228	1.599	7.007	9.704	16.415	1.691
	Batchwise	0.903	0.812	0.899	0.510	0.711	1.395	0.821	1.140	1.388	5.756	12.914	2.243
256	None	0.917	0.860	0.937	0.232	0.428	1.846	0.655	1.392	2.124	11.727	12.338	1.052
	Batchwise	0.910	0.824	0.906	0.487	0.654	1.344	0.825	1.100	1.334	5.140	12.086	2.351
128	None	0.863	0.831	0.963	0.396	0.513	1.296	1.298	1.736	1.337	11.215	9.460	0.844
	Batchwise	0.832	0.783	0.941	0.671	0.777	1.159	1.057	1.213	1.148	6.556	10.651	1.625
16	None	0.590	0.578	0.979	1.202	1.225	1.019	3.039	3.134	1.031	13.549	11.017	0.813
	Batchwise	0.530	0.521	0.983	1.264	1.278	1.011	2.100	2.139	1.019	9.423	8.714	0.925

Table 4.6: ResNet18 Batch Size Variations with CIFAR-10. Once again we take a note of the trend in losses: In mean loss, our approach is performing slightly worse. In $q = 0.9$-loss we see the same apply for batch sizes from 256 and up, while the test $q = 0.9$-loss is better across all batch sizes. This in turn we see in the train-test loss ratio too for mean and sub-$q = 0.9$ losses. The contrary is the case when it comes to worst-case losses, where our filtering results in a lower overall worst-case loss, but a larger gap between worst-case losses of the test and training sets.

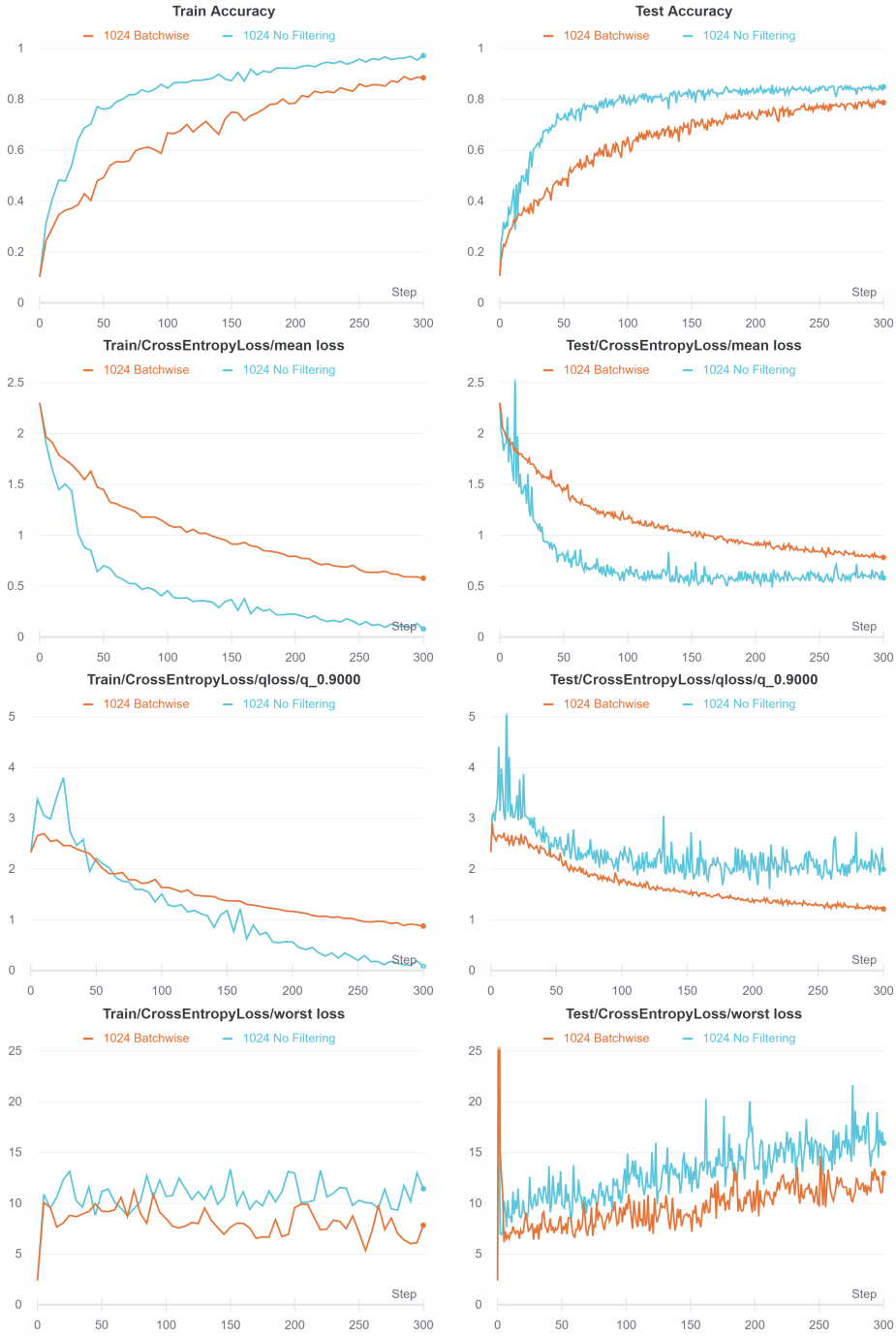


Figure 4.4: ResNet18 Batch Size 1024 on CIFAR-10

Fully Connected Network

CIFAR-10		Accuracy			Mean Loss			Q=0.9 Loss			Worst Loss		
Model	Filtering	Train	Test	Ratio	Train	Test	Ratio	Train	Test	Ratio	Train	Test	Ratio
FCNet3000	None	0.327	0.331	1.012	2.007	2.023	1.008	3.703	3.745	1.011	11.853	13.368	1.128
	Batchwise	0.306	0.305	0.998	1.960	1.957	0.998	2.839	2.845	1.002	7.714	9.661	1.252
FCNet1000	None	0.323	0.324	1.004	1.978	1.984	1.003	3.543	3.591	1.014	12.994	13.454	1.035
	Batchwise	0.304	0.306	1.008	1.956	1.949	0.997	2.803	2.793	0.996	8.276	10.779	1.302
FCNet100	None	0.287	0.291	1.012	2.003	1.997	0.997	3.344	3.329	0.995	19.261	9.056	0.470
	Batchwise	0.277	0.279	1.006	1.975	1.966	0.995	2.697	2.679	0.993	12.644	6.851	0.542

Table 4.7: Fully Connected Network on CIFAR-10, 3000 nodes in hidden layer. For this problem, we dont see such a strong tendency towards the trend we otherwise have seen with mean losses and q=0.9-losses. It still applies, but is for some reason not as strongly present when running on CIFAR-10 with a fully connected network.

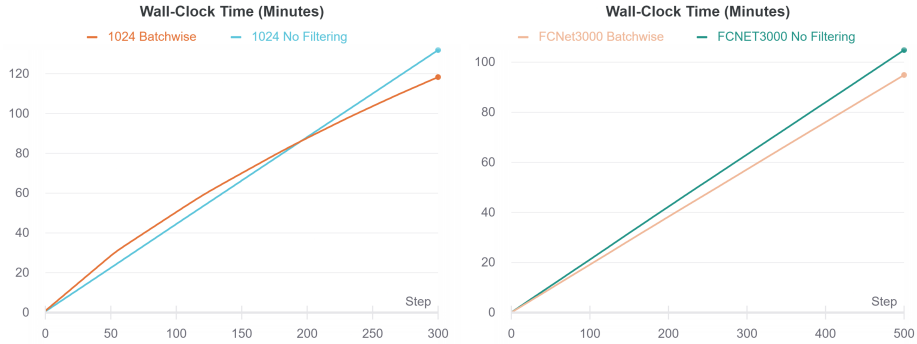
4.3.3 CIFAR-100 Experiments

Batch Size Variations

CIFAR-100		Accuracy			Mean Loss			Q=0.9 Loss			Worst Loss		
Batch Size	Filtering	Train	Test	Ratio	Train	Test	Ratio	Train	Test	Ratio	Train	Test	Ratio
1024	None	0.790	0.512	0.648	0.704	2.371	3.365	2.407	6.920	2.875	14.541	19.328	1.329
	Batchwise	0.924	0.560	0.606	0.388	1.659	4.281	0.957	4.484	4.683	9.300	15.359	1.652
512	None	0.631	0.480	0.760	1.338	2.339	1.748	4.079	6.437	1.578	15.471	15.910	1.028
	Batchwise	0.885	0.565	0.638	0.478	1.647	3.447	1.190	4.413	3.709	9.125	14.732	1.615
128	None	0.462	0.425	0.918	2.011	2.185	1.087	4.645	4.947	1.065	12.182	13.316	1.093
	Batchwise	0.539	0.470	0.871	1.635	1.932	1.182	3.455	4.116	1.191	10.297	10.369	1.007
16	None	0.144	0.145	1.004	3.590	3.602	1.003	5.474	5.490	1.003	15.282	12.813	0.838
	Batchwise	0.145	0.146	1.002	3.583	3.585	1.000	5.270	5.272	1.000	11.979	12.108	1.011

Table 4.8: ResNet18 Batch Size Variations with CIFAR-100. This is perhaps the most interesting experiment found, as it differs a lot from the results we've seen previously. The trend we've seen in loss ratios being better is gone, and the model trained with filtering is outperforming the archetypal, no-filtering approach.

Figure 4.5: CIFAR-10 on a Fully Connected Network with one layer of 3000 nodes.



(a) CIFAR-10 on ResNet18 using a batch size of 1024. Wall-Clock Time.

(b) CIFAR-10 with Fully Connected Network, 3000 nodes in hidden layer. Wall-Clock Time.

Fully Connected Network

CIFAR-100		Accuracy			Mean Loss			Q=0.9 Loss			Worst Loss		
Model	Filtering	Train	Test	Ratio	Train	Test	Ratio	Train	Test	Ratio	Train	Test	Ratio
FCNet3000	None	0.143	0.138	0.962	3.854	3.931	1.020	6.050	6.185	1.022	23.568	20.028	0.850
	Batchwise	0.119	0.115	0.965	3.848	3.904	1.015	5.636	5.705	1.012	19.525	17.375	0.890
FCNet1000	None	0.137	0.134	0.976	3.871	3.940	1.018	6.048	6.162	1.019	23.154	18.758	0.810
	Batchwise	0.114	0.109	0.959	3.880	3.935	1.014	5.670	5.760	1.016	18.734	18.176	0.970
FCNet100	None	0.113	0.107	0.954	3.969	4.022	1.013	5.923	6.018	1.016	16.711	16.392	0.981
	Batchwise	0.100	0.094	0.949	3.944	3.995	1.013	5.567	5.641	1.013	17.227	16.989	0.986

Table 4.9: Fully Connected Network on CIFAR-100, 3000 nodes in hidden layer. Very similar results to that of CIFAR-10, loss ratios are better in all but the worst loss, but in this case, like with CIFAR-10, thhe difference is only marginal.

Figure 4.7: Batch Size Variations on CIFAR-100

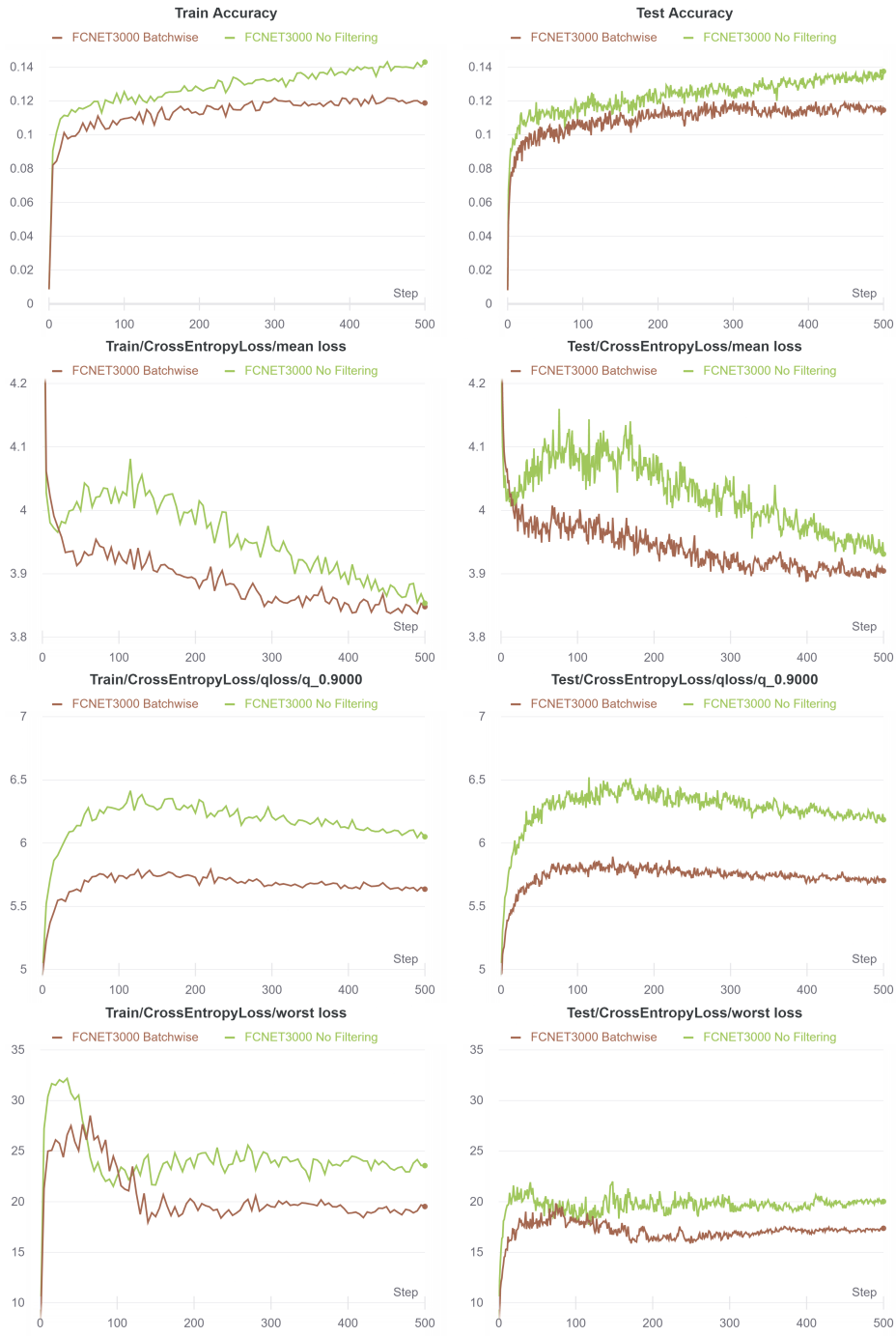
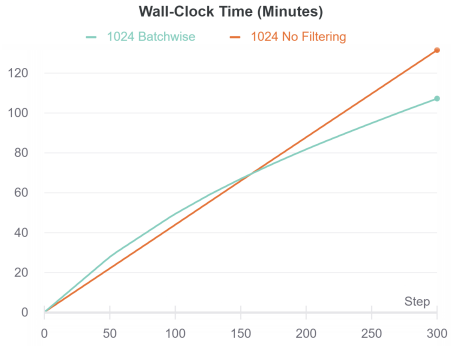
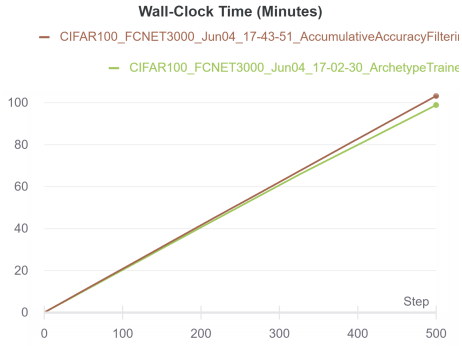


Figure 4.8: CIFAR-100 with Fully Connected Network, 3000 nodes in hidden layer.



(a) CIFAR-100 on ResNet18 using a batch size of 1024. Wall-Clock Time.



(b) CIFAR-100 with Fully Connected Network, 3000 nodes in hidden layer. Wall-Clock Time.

4.4 Assessment

Before we dig into the details of the highlighted experiments, lat us look the experiments on the higher level. We have pointed out that some of the experiments we have taken a note of. Some of which indicated these trends across several runs.

- Different CNNs don't seem to act differently between various modes of filtering.
- Epochwise and batchwise filtering cause the model to perform worse than without filtering in terms of accuracy.
- Epochwise filtering seems worse than both batchwise filtering and no filtering at all in terms of accuracy and worst loss, but is often good in terms of overall losses.
- The smaller the batch size is, the smaller the difference between the archetypal and our approach is.
- An overall improvement in epochs per Wall-Clock Time.
- Batchwise filtering grants more similar mean losses and $q < 0.9$-losses between the training and test sets, the ratio is often close to 1. Contrary to this, batchwise filtering also often grants a worse worst loss-ratio than what the archetype grants us.

Additionally, there's the surprising moment in one of our latest experiments, in which our approach not only surpasses the archetype, but it does so on the most difficult problem of the ones we've put it through, CIFAR-10. Before we go on to talk about the results, what they mean and why we get the results we do, we will address our approach to analysis:

Analysis Tools

We have run a vast number of experiments, and the results are plenty, so we needed a structured way to process it all. To pinpoint areas of interest in our data, and to help in analysis we have utilised some of the toolset *Weights & Biases*[19] provides. Specifically, we used it to continually track experiments while they were running, as well as to evaluate parameter importance in respects to the various tracked metrics. Where the latter was used mostly as a guide for

[19] `https://wandb.ai/`

discovery, hinting at which parameters could have had a legitimate effect when combined with filtering.

In addition to that, to get an overview of all of our runs, we used the python library *pandas*[20]. With it we were able to construct dense tables to easily look at the how our runs performed relative to their most similar configurations in terms of initial parameters. Additionally, it allowed us to easily pinpoint our focus on the different hyperparameters, and look at all runs grouped respectively by their parameters, as a way to both discover trends, and as a way to contradict trends by observing non-trend-following training progressions.

4.4.1 CNNs Seem Unhindered by our Accuracy Filtering

This isn't necessarily all that surprising, but one could imagine a scenario in which sampling only the unhandled samples would lead to a worse initial set of convolution kernels, which again would make it harder for the model to classify samples due to the kernels being ineffective.

We believe however, that the initial few iterations — in which even our sampler strategy is close to stochastic — will allow the model to get kick-started and exposed to enough of the various features of the domain, which will allow it to quickly make differing filters.

One of the potential problems when applying epochwise filtering combined with CNNs might be that the epochwise filtering limits the sample selection too much, and *miss out* on opportunities to learn useful generalisable features, and instead develops niche features of the samples that were accepted during filtering.

Batchwise filtering, on the other hand, we can't see a reason for it to suffer any problem as it doesn't *lock down* the selection for the next couple of iterations as the epochwise one would. Instead, it is doing the exact same as SGD; and uses the selected sample *once* before moving on.

Overall, we don't find much reason to believe that accuracy filtering has an all too serious impact on how well a CNN might train and ultimately perform.

4.4.2 Filtering Causes Worse Accuracy

We see in the majority of our experiments that filtering actually has a negative effect on accuracy. This is of course an unwanted effect, and one we didn't expect to see.

Our hypothesis for why this is is twofold, as we believe the effects of batchwise and epochwise are not of the same causal root.

For epochwise filtering the hypothesis is that the selected samples don't represent enough of the problem in any given batch. The model might improve a lot in just the first update step of an epoch. Depending on the likeness of the samples selected for the batch, all of them might easily be handled by the model after that first epoch too. So instead of helping the model to train faster, we instead end up limiting what data it can learn from, causing a slower and worse training.

For batchwise filtering the hypothesis is that the samples selected are *barely learnt*. The *belief distributions* the model ends up producing when fed a sample end up being more flat for practically all samples we feed it (Our figure for soft accuracy filtering shows this effect intuitively; Appx. Figure 8).

This is supported by the fact that our mean loss and $q = 0.9$-loss ratios are so much closer to 1 than what the archetype-trained network is, but should obviously have been specifically observed directly too.

[20] `https://pandas.pydata.org/`

4.4.3 Epochwise Filtering is the Big Loser?

Among the four variants used throughout our experiments, we early on discarded the epochwise filtering as ineffective. But in analysis the results of the runs we made using epochwise filtering still show various interesting features

First note is that although it has a lower accuracy than no filtering and batchwise filtering, the accuracy isn't all that bad, though this is likely due to the relatively uncomplex problems we let it face (mainly MNIST), and the very good ANN models we used. Besides that, it also seems to end up with very similar training and test set accuracies.

Second note we take is that it often results in a model that has better overall losses than that of our model trained using batchwise filtering. We see this clearly indicated by the quantile loss which from $q = 0.9$ is lower than that of a model being trained using batchwise filtering. Why this is the case isn't really clear to us, though one hypothesis is that because we *lock* our selection down to a certain set of samples, it might be able to better optimise on said samples over a longer time, allowing it to reach such low losses.

Third note we take is that it generally ends up as the loser among the four in terms of worst loss, where again batchwise filtering is the winner. An interesting effect of using both filtering mechanisms is that we end up with a better worst loss than when only using epochwise. This seems like a strong indicator that batchwise filtering in general pushes worst loss down.

Last note we make ourselves when looking at how the epochwise sampler had performed, is how the central filtering function will differ from datasets where data augmentation is required and applied. This is as mentioned in section 3.3.1 not precisely filtering the samples, so we cannot at any point be sure about if the samples presented are samples the model do not already handle.

One hypothesis as to what could improve epochwise filtering, based on the experiences gained in our preliminary work, is that larger batch sizes would benefit the training when using it, with the fundamental idea that the closer to a full BGD we get, the better the resultant gradient will be. One imagined scheme that looks a lot like our Algorithm 5 is essentially the case where one runs BGD with filtering. Effectively, however, this would be the same as batchwise filtering when using the dataset's size as its batch size. Which, as we talked about in section 3.3, often is unfeasible or impossible due to the resource requirements of doing so.

4.4.4 Diminishing Effects of Batchwise Filtering on Small Batches

We take a note that the runs using smaller batch size usually tend to perform similarly to the non-filtering training loop, both in terms of accuracy and in losses. Having said that, we also take a note of a slight edge our approach has in losses. Seemingly the filtering is still affecting the losses towards being more similar in between training and test set.

This was expected behaviour, as a smaller batch size, also with filtering enabled, will leave out a vast number of important samples, similar to that of SGD.

4.4.5 Larger Number of Epochs Passed per Wall-Clock Time

We take note of how our sampling mechanism seemed to eventually cause faster and faster epochs, until it again reached a seemingly stable and linear progression. Initially, this was something that piqued our interest, as it seemed our filtering somehow caused a faster training, not in terms of accuracy per training step, but literally in terms of time.

This was of course not the case, as our algorithm itself was doing more work behind the scenes than a standard stochastic sampler would be doing. An hypothesis rose as to why this was happening, and we confirmed it by tracking the dataflow of our samples: The model had

gotten to the point where it was already capable of classifying a few batches worth of samples, meaning that the pool of samples to fill a batch of was shrinking (We investigated this, and tracked it, see appx. fig. 6a).

This again meant that over time our epochs would contain fewer and fewer training steps. This is why we instead have chosen to present the experiment plots in epoch time steps, rather than wall-clock time, but supplement with the Wall-Clock Time plots to give the full picture. Arguably, considering this, the ideal way we should have tracked and presented our data is per training step. Alas, this comes as an afterthought upon analysing the results.

It is nonetheless worth mentioning, as it brings to light a possibility that our sample selection might yet be considered more beneficial in terms of progress per training steps. We can draw this conclusion from the fact that we are doing fewer and fewer training steps per epoch, but still reach comparable results to that of the archetype.

4.4.6 Generalisation and What our Loss-Ratios Mean

One of the clearest trends we see in our gathered data is that accuracy filtering does have an effect. Specifically, batchwise accuracy filtering is consistently more self-similar in its mean losses and $q = 0.9$-losses between training data and test data. This we see as an indicator that our sampling mechanism to some degree limits overfitting, and allows for more generalisation.

This as a result makes a lot of sense considering we don't keep training on samples the model already handles. Indeed, it is the intended behaviour we, in retrospect, should have expected to see. These results, per our interpretation, means that the ANN subjected to accuracy-filtering (especially batchwise) is tuned in such a way as to barely keep the samples within its decision boundary.

An unfortunate side-effect of not minimising the losses further on all samples that are already modelled and classifies correctly, is that our classifier won't further build up *confidence* about the classified sample. The slightest difference in a presented sample might make a model trained with filtering classify wrongly. but because of the distribution of beliefs, the loss ends up being relatively low; after all, the classifier wouldn't be that far of from classifying correctly if the beliefs it had were pretty much a similar to a uniform distribution (at least when dealing with few classes).

We sum this up to the idea that our approach to training leads to a generalising model that ironically also is too *indecisive* to get high accuracies.

Another interesting observation made in terms of losses is the overall higher train/test-ratio we see in worst loss our approach leads to. This yet again is an effect which although initially is surprising, makes sense when we go to the root of what our approach is doing. The worst-case samples; these are the samples that consistently ends up in the pool of samples that our model actually trains on in the end. With that, these samples also end up being the few samples the training loop gets to use for training the model. It poses the question of whether our approach ultimately ends up overfitting the worst-case samples.

4.4.7 Outperforming the Archetype on CIFAR-100?

When getting to the results of CIFAR-100, the most complex problem of the ones we have been using throughout the thesis, it was surprising to see the trends of batchwise filtering, having been the *runner-up* in all the other experiments, all of a sudden surpass the archetype, and beat it by a fair margin too, with an improvement of 0.1–8.5% (measured in test accuracy difference) depending on the batch size used (seen in table 4.8).

Trying to explain this behaviour with the previous hypotheses in mind is tricky. It is however also the kind of behaviour we sought out to discover, where a change of parameter shows a clear difference in training progression. In this case the "parameter" tweaked is the actual problem we're exposing the model to.

To reason about where this change in behavior occurs, we reiterate the differences between CIFAR-100 and CIFAR-10:

- The number of classes.
- The number of samples representing each class.
- The content of the images.

Specifically, we believe that the second point is of significance when applying filtering. With CIFAR-100 we have a tenth of the available data per class to build convolution kernels out of. Seemingly, there is a dynamic in which we select more important samples to construct good kernels to classify more samples correctly than what the archetype does using the same set of parameters.

One hypothesis is that because we now are dealing with a larger number of classes and fewer samples, we are utilising the batch size we have better than the stochastic sampler does, which may choose samples the training model already knows. But ultimately this experiment has left us baffled.

There is a catch to it all however; as this behaviour arises in a scenario in which the archetype is not using the state-of-the-art additions found to significantly improve accuracy in CIFAR-10 and CIFAR-100:

A fairly typical way to fine tune a classifier model at the end of its training is to reduce the learning rate. As it is fairly typical to do so, we also ran experiments in which we apply learning rate decay (using the exponential learning rate scheduler listed in table 4.1), or apply a reduction in learning rate per nth epoch (using the multistep learning rate scheduler listed in table 4.1). As expected, we see an improvement in both training set accuracy, and test set accuracy for the archetype, but the same does not apply when using our sampling strategy, which instead stagnates at an test accuracy $\sim 5.3\%$ of absolute test accuracy lower than the archetype which plateaus at around 75.5% test accuracy. (see appx. fig. 5)

V

Discussion

In this chapter we will discuss findings and reason about what they mean in relation to our posed research questions. From there we will discuss and present shortcomings of our sample selection processes, and the research conducted, before we re-iterate on what our work ultimately has contributed to, and what the road from here looks like before we conclude.

5.1 Findings

We set out on this journey with the intent of exploring and implementing strategic sampling mechanisms for improving training of neural network models. We did so knowing that we at least had a couple of options we could look into to use in guiding our selection process, where losses were one of the options, while the second was a novel idea that instead would base itself upon the accuracy as a metric for guiding selection. This lead us to posing our primary research question. We ask *How do strategic data sampling methods perform as an alternative to the de facto standard of mini-batch SGD?*.

To set us of we look into others' prior works related to the issue of non-stochastic sampling, and find that indeed several findings have been made on various strategic sampling schemes, where several indicate an improvement in performance when compared against mini-batch SGD.

To further enable us to answer our research question we pose three supplementary sub-questions regarding in which aspects we look for performance gains.

RQ.1.1 — Accuracy

The first sub-question we look to is RQ1.1 — how strategic sampling methods perform in terms of accuracy.

From related works, a clear conclusion to this question is that it often performs well, especially when considered in terms of a fixed wall-clock time span, strategic samplers gain higher accuracies, and as such have proven to be beneficial. The literature also presents the

idea of having the sampler be adaptive to some tracked or computed metric determining whether or not it is deemed to be beneficial to design a selection, or if stochastic sampling will be the better option in a given step of training [Katharopoulos and Fleuret, 2018; Song et al., 2020].

Our own findings were somewhat different to that of the literature, in the sense that we generally weren't able to outperform the archetype, but then again our approach to strategic sampling is also very different to that of the literature, so a different outcome was considerably more interesting, as that brings nuances to when, and for what we might say that strategic sampling works, and how it performs.

From this we see that indeed there are cases in which strategic sampling perform better than the archetypal, in the literature, it specifically seems to benefit the start of the training as it can cause high accuracies in a short amount of time. With that said, the archetype may yet surpass these high accuracies over longer wall-clock time spans by applying well-known fine-tuning optimisations such as decreasing learning rate. This was also the case we saw in our one example where accuracy-filtering proved to reach higher accuracies faster than the archetype (see appx. fig. 5).

RQ.1.2 — Loss

The second sub-question we look to is RQ1.2 — how strategic sampling methods perform in terms of loss.

In the literature, we found there were several findings about losses and overfitting issues related to strategic sampling. Conclusively, all have their merits and limitations in terms of losses, and in terms of overfitting. Some are better suited for low-complexity problems, while others are more robust. Indicating that the state-of-the-art is still paving the way for strategic sampling being a viable or better alternative to archetypal mini-batch SGD.

Throughout this thesis we have considered several loss-metrics to get into the depths of how losses are affected by using our two proposed filtering sampling schemes. In our efforts we particularly paid attention to the ratios between training losses and test losses, as they together with the accuracy indicate how well we are generalising, and how our model might be overfitting. Empirically we found an overall improvement in the training loss to test loss ratio, indicating that the sampling mechanism should lead to a generalising model.

However, we also found that due to the filtering hard boundary cutoff (i.e. classification accuracy), training fails to achieve optimal training accuracies, meaning of course that the test accuracies also remain worse than to that of an archetype-trained model.

Additionally, we found evidence of what we dub *overfitting of worst case samples*, where, in some cases the worst case samples continually are included in a batch, causing the model to adjust particularly to these samples. Using a filtering mechanism that is based on accuracy also typically results in worst loss-samples being included. As such, the filtering causes training to focus on these samples from the training dataset, making the overall fit adjusted specifically to minimise the losses of these worst-case losses. This we consider an overfitting-like side-effect our filtering sampling unfortunately inhabits.

RQ.1.3 — Wall-Clock Training Time

The third, and final sub-question we look to is RQ1.3 — how strategic sampling methods perform in terms of wall-clock training time.

In the literature, this is emphasised as one of the advantages of applying strategic sampling methods. It is shown empirically that various strategic samplers achieve a better accuracy in a shorter amount of time, and that the overhead added to the sampling procedure itself, while substantial compared to the simplicity of stochastic sampling, is negligible because of the gained value of each training step.

This is pointed out in terms of a fixed wall-clock time however, and in general the value of applying strategic sampling is diminishing the further into training we get.

Although the improved wall-clock training time is not a trend see in our results when employing accuracy filtering as the strategic sampling method, we can see indications of training steps of higher value when using it. This is indicated when our accuracy filtering cause fewer and fewer training steps to be taken per epoch, while we still reach comparable results to that of the archetype in comparable time.

5.2 Limitations of our Work

5.2.1 Data Augmentation

As highlighted in section 3.3.1, data augmentation is a common practice to get more use out of the data provided in the training set. In section 3.4, we emphasise that in our current implementation of epochwise filtering, we achieve a sub-optimal filtering in which the data filtered upon may be a vastly different augmentation of the same sample from the data

This sub-optimal filtering was left as is as we saw it as a potential to explore how such a "soft" filtering mechanism would behave. After all, the underlying images (pre-augmentation) are the same, so one way to look at it is that one augmented sample represents other augmentations of the same sample too, which of course is what we ideally want, and what we to some degree might have in the later stages of training. But early on in the training this "soft" filtering is not ideal, as it effectively causes selection of samples of unknown difficulty instead of samples the model cannot handle.

Ultimately however, it served to be a source of confusion, which contributed to us practically discarding it as a useful sampling mechanism. We did perform some runs explicitly attempting to explore how it behaved (see appendix II; experiments using epochwise filtering with and without data augmentation), where in general it seemed to still be working similarly to how it would perform when not using data augmentation, albeit with a higher accuracy, which we attribute to the data augmentation rather than to our filtering.

Our initial idea on how one might mitigate the issues related to data augmentation to achieve true filtering on epochwise filters was to keep track of the PRNG-state and fork the state right before filtering is done, then join the state afterwards, ensuring that the randomly generated augmentation-transformations are happens in the same way upon training. This would work, but adds a fair bit of complexity to the sampler.

5.2.2 Batch Accumulation

An issue we brought up in section 3.3 when talking about how we accumulate batches (see algorithm 10 and subsequent fig. 3.2), is the problem of how we best fill our filtered batch with as little overhead as possible. Filtering may involve several batch look-ups of the raw data, before our filtered batch is fully formed. This is just the nature of how accumulation of batches work.

It makes the approach sub-optimal as there is an added overhead where we gradually end up having to do more and more look-ups of raw data in order to accumulate enough filtered samples to create a filtered batch. This is shown in appx. fig. 6a, where the overhead gradually increases as number of required raw batches to produce one filtered batch increases.

Our suggestion to reduce this overhead is tightly coupled to what we see in the figures. Seeing as we can track how many batches we needed to produce one batch the previous steps, we can employ simple extrapolation and easily estimate how many batches are needed based on the previous required steps. Then we simply adjust how many samples we take out of the dataset as the raw batch. Essentially, we would be estimating how much of the raw batch would be accepted, and in so doing alleviate us of some overhead. Some overhead is still to be expected however, as we're sure to miss the target number of accepted samples every now

and then too. And at some point, the acceptance ratio might lead to us having to fetch huge batches of raw data, which again leads us into the problem of memory.

Varying Batch Size or Number of Training Steps per Epoch

This problem is slightly linked with the previous one, and is again emphasised in section 3.3.2. With filtering employed, we end up having to break one of two *norms* in supervised machine learning:

Varying Batch Size We either end up having to break the norm of using a static batch size, which could have negative side-effects on training, as the size of a batch may vastly change how big the resultant parameter adjustments are.

Varying Number of Training Steps per Epoch Alternatively, we end up with what we have, an epoch that is just a construct vaguely hinting at how far we have gotten in our training. It's mainly problematic in terms of comparison with other approaches, as they cannot really be considered to have gone through the same number of training steps, making it a somewhat poor grounds for comparison.

How we get around this problem is really not clear, as it seems rooted of the supervised machine learning-training loop.

Not Getting Through all Experiments

There is a rather serious point to be made about our work in terms of the experiments we ran, as not all ran to completion. Ideally, we'd run experiments using all (sensible) combinations of our adjustable parameters (seen in table 4.1). Due to time constraints, queuing-times (on the cluster used to run experiments [Själander et al., 2019]) and crashes of runs and experiments, we regrettably weren't quite able to do so.

The experiments we did run (appendix II) are nonetheless satisfiable, and we deem them thorough enough to draw the conclusions we have drawn. Yet, we see it as a lost opportunity that we weren't able to test more combinations, especially involving more optimizer variants and settings, as we'd like to have covered a larger basis to get a better overview of where our approach specifically succeeds or fails.

5.3 Contributions of our Work

Summarising the merits of our work in this thesis, we will now reiterate, and present what we believe to be our biggest contributions to the AI-field.

We demonstrate that accuracy filtering on its own is not sufficient Within the relatively large set of differing configurations we have put our approaches to the test, we demonstrate that accuracy filtering on its own is insufficient as the sole mechanism for selection of samples.

We suggest that a combination of filtering and loss-oriented selection might be beneficial Presenting a set of algortithms that utilise both loss and accuracy as grounds for the selection, as well as results from our exploratory preliminary experiments indicating, we suggest that the two combined may yet prove a good contestant for improved supervised learning.

We observe that accuracy filtering results in a lower worst-case loss and better generalisation ratio Begging the question if this is something we can further dig into, and potentially find new methods to improve generalising behaviour of ANNs without it affecting the overall accuracy as much as it has done in our case.

5.4 Future Work

It is bittersweet to see our accuracy filtering work as poorly as it does. We had high hopes that it would prove effective. It has however caused a great surge in inspiration, and has been at the root of many a new idea throughout its development.

We see it evident that the concept of accuracy filtering has some potential yet, although we also acknowledge that it on its own is far from sufficient in aiding the training of an image classifier.

We have seen several works bring forward an improved performance using importance sampling or otherwise non-stochastic sampling. With our work we also present results suggesting that filtering yet may prove useful to ensure generalisation capabilities of an ANN model.

In our preliminary work, we worked out several algorithms combining the two concepts, and through ad-hoc experimentation found several to be promising.

Seeing as we with this thesis instead focused our efforts on what we considered to be the foundation of said algorithms, the filtering, the various presented additions are still open and viable options for where to keep going from here.

Concretely, we suggest that combining accuracy filtering and strategic sampling may yet prove to be a beneficial approach, both in terms of runtime, generalisation-capabilities and, of course, in terms of accuracy. Based in the preliminary work, we already have a great deal of ideas for where to take this next:

We suggest combining one of the following selectors, with one of the following filtering mechanisms:

Selectors	
Top-k Selector	Selecting only the k samples with the worst loss
Top-q Selector	Selecting only the sample from quantile q and above
Loss Weighted Random Selector	Using the sorted losses to make a probability density function for weighting the selection of the respective samples.
Loss q-Split Random Selector	Bucketing samples based on their losses, and selecting equally many samples from each bucket.
Filters	
Hard Accuracy Filter	Any sample that is already handled by the model is rejected from the sampling pool.
Soft Accuracy Filter	Any sample that is already handled by the model is rejected from the sampling pool, unless its close to a decision boundary (i.e. the *belief* distribution has more than one clear contender for which class the model should decide for).

One of the last results we found was of CIFAR-100 being surpasses by our implementation. We see this as a basis upon which we new research can start, investigating how accuracy filtering works on larger and larger datasets (in terms of number of classes).

One way to start of such a project would be to take control of the dataset and split it up manually. That way one could run experiments of differing number of classes, and of differing number of samples per class, and see when this behaviour occurs.

VI

Conclusion

We have investigated the topic of non-uniform, strategic data sampling for improved supervised learning and considered how it performs compared to the commonplace standard of Mini-Batch Stochastic Gradient Descent. We find that recent works have made headway in this respect and that their findings indicate greater performance in a fixed wall-clock time in terms of accuracy, and at times in terms of losses and how generalising the trained models ultimately are.

Furthermore, we have developed a novel *filtering* mechanism to perform strategic data sampling to aid supervised learning in image classification problems based on the fundamental metric of classification accuracy. We do this to investigate whether the binary accuracy is a sufficient metric to base selection on to get closer to an answer about whether such a strategic sampling mechanism also may be a viable option to mini-batch SGD.

To investigate the effects of our *filtering* mechanism, we employed extensive experimentation looking at various combinations of features and parameters used in addition to our filtering, with the hopes of finding trends unique to our approach. To that extent, we parallelly ran experiments with the same configurations using mini-batch SGD, and, in respects to our research questions discovered that: accuracy is not sufficient on its own, filtering causes losses that indicate high generalisation capabilities in the model, and finally that the selection made through filtering does not yield a better wall-clock training time, but tends to have a comparable improvement in fewer training steps.

We conclude with a discussion on how strategic sampling compares in performance to that of mini-batch SGD, in which we highlight that while our own approach fails to reach similar performance trends as other before us looking into strategic sampling methods, our strategic sampler mechanism has its own merits, such as a ratio close to one between training and test losses, indicating an inherit generalising behaviour.

Our findings present new opportunities of which we introduce, and talk about, illustrating our thoughts on where to take the road from here.

We have in this thesis done body of work that has lead us ever closer to answer the research questions posed in the beginning, and we ultimately consider our efforts a foot in the door, pushing towards more research not focusing inwards, but outwards, to the consideration of a sample's importance in supervised learning.

Bibliography

Asthana, M. (2020). Layers of a convolutional neural network.

Chang, H.-S., Learned-Miller, E., and McCallum, A. (2017). Active bias: Training more accurate neural networks by emphasizing high variance samples. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition. *CoRR*, abs/1512.03385.

Jiang, A. H., Wong, D. L.-K., Zhou, G., Andersen, D., Dean, J., Ganger, G., Joshi, G., Kaminsky, M., Kozuch, M. A., Lipton, Z. C., and Pillai, P. (2019). Accelerating deep learning by focusing on the biggest losers. *ArXiv*, abs/1910.00762.

Jiang, L., Zhou, Z., Leung, T., Li, L., and Fei-Fei, L. (2018). Mentornet: Learning data-driven curriculum for very deep neural networks on corrupted labels. In *ICML*.

Johnson, T. B. and Guestrin, C. (2018). Training deep models faster with robust, approximate importance sampling. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.

Katharopoulos, A. and Fleuret, F. (2018). Not all samples are created equal: Deep learning with importance sampling. In Dy, J. and Krause, A., editors, *Proceedings of the 35th International*

Conference on Machine Learning, volume 80 of *Proceedings of Machine Learning Research*, pages 2525–2534. PMLR.

Krizhevsky, A. (2012). Learning multiple layers of features from tiny images. *University of Toronto*.

kuangliu (2016). pytorch-cifar. `https://github.com/kuangliu/pytorch-cifar`.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

Lin, T., Goyal, P., Girshick, R., He, K., and Dollár, P. (2017). Focal loss for dense object detection. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*.

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *ArXiv*, abs/1609.04747.

Själander, M., Jahre, M., Tufte, G., and Reissmann, N. (2019). EPIC: an energy-efficient, high-performance GPGPU computing research infrastructure. *CoRR*, abs/1912.05848.

Song, H., Kim, M., Kim, S., and Lee, J.-G. (2020). Carpe diem, seize the samples uncertain "at the moment" for adaptive batch selection. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, CIKM '20, pages 1385–1394, New York, NY, USA. Association for Computing Machinery.

Yu, F., Wang, D., Shelhamer, E., and Darrell, T. (2018). Deep layer aggregation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Zhang, J., Yu, H.-F., and Dhillon, I. S. (2019). Autoassist: A framework to accelerate training of deep neural networks. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Appendices

I Sampling

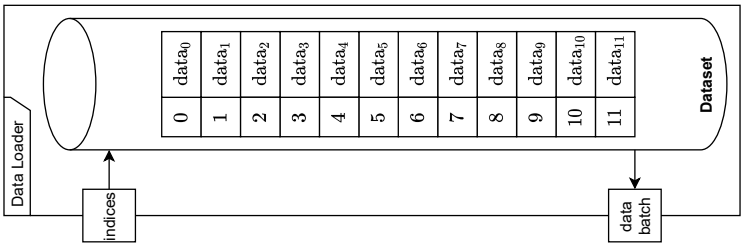


Figure 1: Data Loader

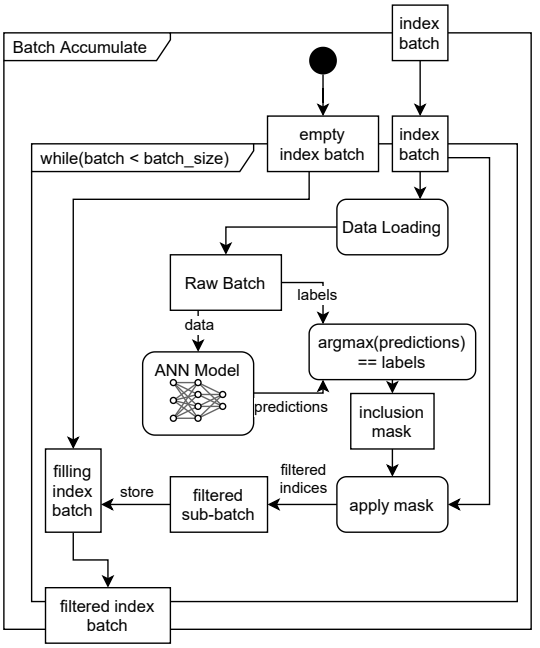
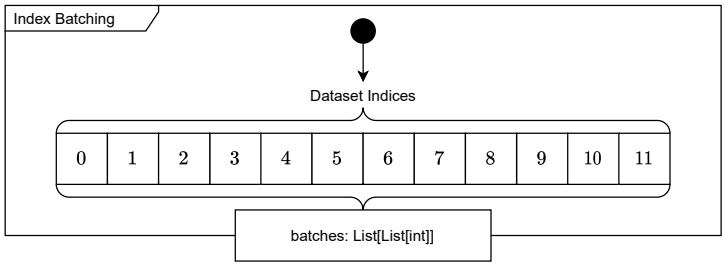
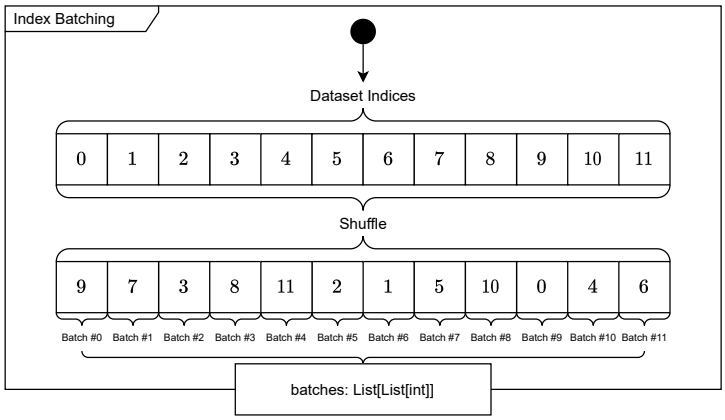


Figure 2: Batch Accumulation

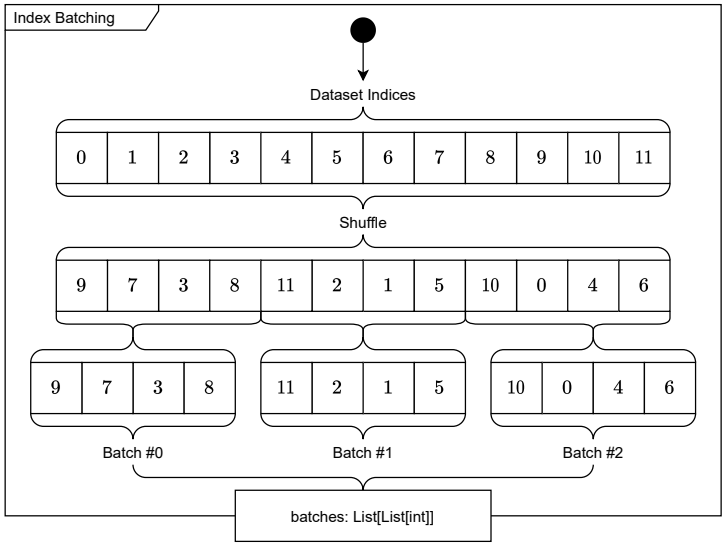
Figure 3: Gradient Descent Sampling Schemes



(a) Batch Gradient Descent Sampling Scheme

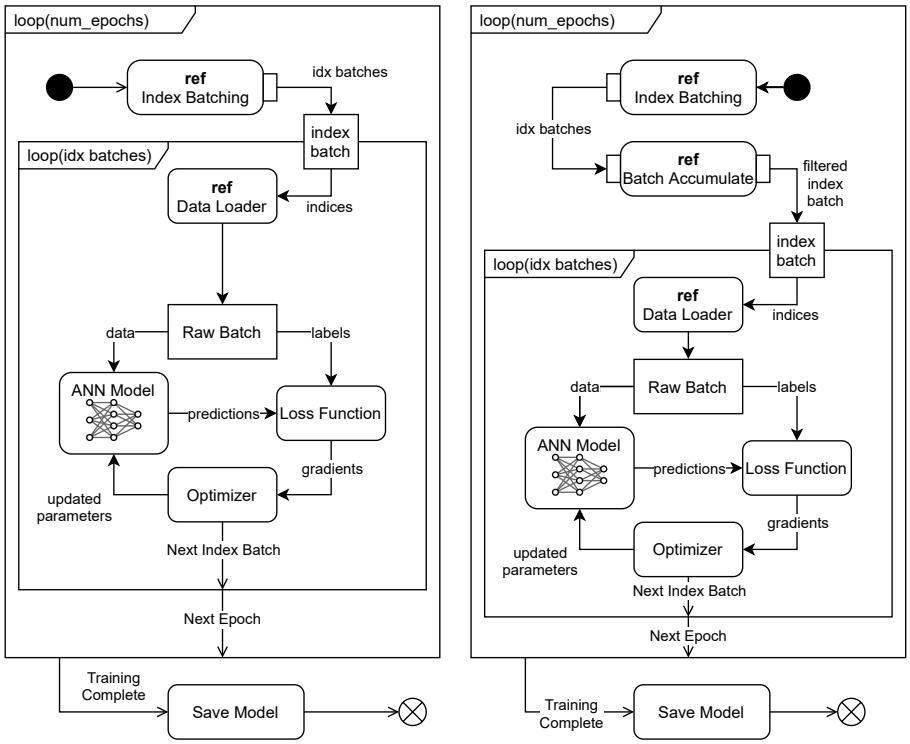


(b) Stochastic Gradient Descent Sampling Scheme



(c) Mini-Batch Stochastic Gradient Descent Sampling Scheme

Figure 4: Trainer Loops



(a) Archetype

(b) Accumulative Accuracy Filtering

II Experiments

Dataset	Batchwise filtering	Epochwise filtering	Model	Batch size	Optimizer	Learning rate	Gamma	Momentum	Scheduler	Num epochs	Weight decay	Ran to completion	Data augment	Milestone per nth step	Test accuracy	Train accuracy	Train mean loss	Test mean loss	Train q=0.9 loss	Test q=0.9 loss	Train worst loss	Test worst loss
CIFAR10	✗	✗	FCNet100	512	SGD	0.010	0.995	0.900	exp	500	0.001	✓	✓	✗	0.291	0.287	2.003	1.997	3.344	3.329	19.261	9.056
CIFAR10	✓	✗	FCNet100	512	SGD	0.010	0.995	0.900	exp	500	0.001	✓	✓	✗	0.279	0.277	1.975	1.966	2.697	2.679	12.644	6.851
CIFAR10	✗	✗	FCNet1000	512	SGD	0.010	0.995	0.900	exp	500	0.001	✓	✓	✗	0.324	0.323	1.978	1.984	3.543	3.591	12.994	13.454
CIFAR10	✓	✗	FCNet1000	512	SGD	0.010	0.995	0.900	exp	500	0.001	✓	✓	✗	0.306	0.304	1.956	1.949	2.803	2.793	8.276	10.779
CIFAR10	✗	✗	FCNet3000	512	SGD	0.010	0.995	0.900	exp	500	0.001	✓	✓	✗	0.331	0.327	2.007	2.023	3.703	3.745	11.853	13.368
CIFAR10	✓	✗	FCNet3000	512	SGD	0.010	0.995	0.900	exp	500	0.001	✓	✓	✗	0.305	0.306	1.960	1.957	2.839	2.845	7.714	9.661
CIFAR10	✗	✗	ResNet18	16	SGD	0.100	0.995	0.900	exp	300	0.001	✓	✓	✗	0.578	0.590	1.202	1.225	3.039	3.134	13.549	11.017
CIFAR10	✓	✗	ResNet18	16	SGD	0.100	0.995	0.900	exp	300	0.001	✓	✓	✗	0.521	0.530	1.264	1.278	2.100	2.139	9.423	8.714
CIFAR10	✗	✗	ResNet18	128	SGD	0.100	0.995	0.900	exp	300	0.001	✓	✓	✗	0.831	0.863	0.396	0.513	1.298	1.736	11.215	9.460
CIFAR10	✓	✗	ResNet18	128	SGD	0.100	0.995	0.900	exp	300	0.001	✓	✓	✗	0.783	0.832	0.671	0.777	1.057	1.213	6.556	10.651
CIFAR10	✗	✗	ResNet18	256	SGD	0.100	0.995	0.900	exp	300	0.001	✓	✓	✗	0.860	0.917	0.232	0.428	0.655	1.392	11.727	12.338
CIFAR10	✓	✗	ResNet18	256	SGD	0.100	0.995	0.900	exp	300	0.001	✓	✓	✗	0.824	0.910	0.487	0.654	0.825	1.100	5.140	12.086
CIFAR10	✗	✗	ResNet18	512	SGD	0.100	0.995	0.900	exp	300	0.001	✓	✓	✗	0.862	0.953	0.133	0.495	0.228	1.599	9.704	16.415
CIFAR10	✓	✗	ResNet18	512	SGD	0.100	0.995	0.900	exp	300	0.001	✓	✓	✗	0.812	0.903	0.510	0.711	0.821	1.140	5.756	12.914
CIFAR10	✗	✗	ResNet18	1024	SGD	0.100	0.995	0.900	exp	300	0.001	✓	✓	✗	0.849	0.972	0.079	0.582	0.084	1.993	11.435	15.933
CIFAR10	✓	✗	ResNet18	1024	SGD	0.100	0.995	0.900	exp	300	0.001	✓	✓	✗	0.787	0.885	0.579	0.783	0.875	1.214	7.834	12.946
CIFAR10	✗	✗	ResNet18	1024	SGD	0.100	0.990	0.900	exp	500	0.001	✓	✓	✗	0.937	1.000	0.001	0.259	0.001	0.088	0.048	9.688
CIFAR10	✓	✗	ResNet18	1024	SGD	0.100	0.990	0.900	exp	500	0.001	✓	✓	✗	0.935	1.000	0.001	0.273	0.002	0.105	0.063	9.714
CIFAR10	✗	✗	ResNet18	1024	SGD	0.100	0.995	0.000	exp	1000	0.001	✓	✓	✗	0.912	1.000	0.001	0.378	0.001	0.464	0.138	16.683
CIFAR10	✗	✗	ResNet18	1024	SGD	0.100	0.995	0.900	exp	1000	0.000	✓	✓	✗	0.925	1.000	0.000	0.567	0.000	0.087	0.031	33.354
CIFAR10	✗	✗	ResNet18	1024	SGD	0.100	0.995	0.000	exp	1000	0.000	✓	✓	✗	0.911	1.000	0.000	0.553	0.000	0.395	0.097	28.816
CIFAR10	✓	✗	ResNet18	1024	SGD	0.100	0.995	0.000	exp	1000	0.001	✓	✓	✗	0.861	0.984	0.175	0.571	0.588	1.082	5.338	13.192
CIFAR10	✓	✗	ResNet18	1024	SGD	0.100	0.995	0.900	exp	1000	0.000	✓	✓	✗	0.839	0.985	0.092	0.802	0.281	1.760	9.577	49.725
CIFAR10	✓	✗	ResNet18	1024	SGD	0.100	0.995	0.000	exp	1000	0.000	✓	✓	✗	0.860	0.981	0.176	0.610	0.578	1.135	7.770	15.904
CIFAR10	✗	✗	ResNet18	1024	SGD	0.100	0.100	0.900	ms	500	0.001	✓	✓	200	0.947	1.000	0.001	0.210	0.002	0.057	0.102	10.168
CIFAR10	✓	✗	ResNet18	1024	SGD	0.100	0.100	0.900	ms	500	0.001	✓	✓	200	0.943	1.000	0.001	0.210	0.002	0.103	0.778	12.404
CIFAR10	✗	✓	ResNet18	1024	SGD	0.100	0.500	0.900	ms	1000	0.001	✓	✗	150	0.757	0.845	0.527	0.841	1.973	3.144	13.575	12.086
CIFAR10	✗	✓	ResNet18	1024	SGD	0.100	0.500	0.900	ms	1000	0.001	✓	✓	150	0.837	0.875	0.502	0.703	1.483	2.648	17.435	16.139
CIFAR10	✗	✗	ResNet18	1024	SGD	0.100	0.500	0.000	ms	1000	0.001	✓	✓	150	0.922	1.000	0.001	0.313	0.001	0.347	0.080	13.873
CIFAR10	✗	✗	ResNet18	1024	SGD	0.100	0.500	0.900	ms	1000	0.000	✓	✓	150	0.929	1.000	0.000	0.584	0.000	0.030	0.020	38.028
CIFAR10	✗	✗	ResNet18	1024	SGD	0.100	0.500	0.000	ms	1000	0.000	✓	✓	150	0.917	1.000	0.000	0.514	0.000	0.239	0.574	28.849
CIFAR10	✓	✗	ResNet18	1024	SGD	0.100	0.500	0.900	ms	1000	0.001	✓	✗	150	0.763	0.997	0.039	0.885	0.084	3.081	12.268	16.040
CIFAR10	✓	✗	ResNet18	1024	SGD	0.100	0.500	0.900	ms	1000	0.001	✓	✓	150	0.886	0.996	0.042	0.451	0.110	1.035	6.238	14.749
CIFAR10	✓	✗	ResNet18	1024	SGD	0.100	0.500	0.000	ms	1000	0.001	✓	✓	150	0.890	0.995	0.127	0.437	0.436	0.806	6.997	22.265
CIFAR10	✓	✗	ResNet18	1024	SGD	0.100	0.500	0.900	ms	1000	0.000	✓	✓	150	0.861	0.993	0.055	0.693	0.143	1.653	20.171	68.813
CIFAR10	✓	✗	ResNet18	1024	SGD	0.100	0.500	0.000	ms	1000	0.000	✓	✓	150	0.879	0.993	0.139	0.514	0.486	0.881	3.614	19.868
CIFAR10	✗	✗	ResNet18	16	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✓	✗	0.404	0.411	1.637	1.650	3.090	3.091	11.322	8.899
CIFAR10	✓	✗	ResNet18	16	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✓	✗	0.368	0.368	1.645	1.645	2.560	2.549	8.362	6.832
CIFAR10	✗	✗	ResNet18	128	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✓	✗	0.610	0.615	1.167	1.196	3.312	3.364	14.515	13.705
CIFAR10	✓	✗	ResNet18	128	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✓	✗	0.626	0.650	1.025	1.060	1.543	1.618	8.048	8.214
CIFAR10	✗	✗	ResNet18	512	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✓	✗	0.797	0.832	0.485	0.610	1.654	2.138	12.917	11.246
CIFAR10	✓	✗	ResNet18	512	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✓	✗	0.748	0.787	0.774	0.855	1.313	1.507	9.276	7.040
CIFAR10	✗	✗	ResNet18	1024	SGD	0.100	0.000	0.900	✗	150	0.001	✓	✓	✗	0.937	1.000	0.002	0.239	0.003	0.152	0.908	13.125
CIFAR10	✗	✗	ResNet18	1024	SGD	0.100	0.000	0.900	✗	150	0.001	✓	✓	✗	0.936	1.000	0.002	0.235	0.003	0.158	1.826	12.402
CIFAR10	✓	✗	ResNet18	1024	SGD	0.100	0.000	0.900	✗	150	0.001	✓	✓	✗	0.892	0.966	0.098	0.392	0.122	0.993	7.927	13.078
CIFAR10	✓	✗	ResNet18	1024	SGD	0.100	0.000	0.900	✗	150	0.001	✓	✓	✗	0.887	0.957	0.122	0.436	0.147	1.124	9.838	12.249
CIFAR10	✗	✗	ResNet18	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✓	✗	0.813	0.880	0.352	0.605	1.096	2.170	10.141	14.323
CIFAR10	✗	✗	ResNet18	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✓	✗	0.947	1.000	0.001	0.197	0.002	0.067	0.296	9.943
CIFAR10	✗	✗	ResNet18	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✓	✗	0.946	1.000	0.001	0.199	0.002	0.079	0.276	9.793
CIFAR10	✓	✗	ResNet18	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✓	✗	0.760	0.825	0.616	0.779	1.047	1.368	8.503	10.788
CIFAR10	✓	✗	ResNet18	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✓	✗	0.887	0.957	0.127	0.414	0.152	1.123	9.181	12.018
CIFAR10	✓	✗	ResNet18	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✓	✗	0.885	0.960	0.120	0.450	0.132	1.271	11.054	13.457
CIFAR10	✗	✗	ResNet18	1024	SGD	0.100	0.000	0.000	✗	1000	0.001	✓	✓	✗	0.941	1.000	0.001	0.248	0.001	0.063	0.088	10.381
CIFAR10	✗	✗	ResNet18	1024	SGD	0.100	0.000	0.900	✗	1000	0.000	✓	✓	✗	0.928	1.000	0.000	0.635	0.000	0.024	0.005	37.619
CIFAR10	✗	✗	ResNet18	1024	SGD	0.100	0.000	0.000	✗	1000	0.000	✓	✓	✗	0.919	1.000	0.000	0.542	0.000	0.177	0.010	32.486
CIFAR10	✗	✓	ResNet18	1024	SGD	0.100	0.000	0.900	✗	1000	0.001	✓	✗	✗	0.725	0.821	0.623	0.968	2.470	3.468	13.519	13.483
CIFAR10	✗	✓	ResNet18	1024	SGD	0.100	0.000	0.900	✗	1000	0.001	✓	✓	✗	0.846	0.880	0.456	0.639	1.290	2.314	16.061	15.125
CIFAR10	✗	✗	ResNet18	1024	SGD	0.100	0.000	0.900	✗	1000	0.001	✓	✗	✗	0.810	1.000	0.002	0.688	0.004	2.679	0.020	11.773
CIFAR10	✗	✗	ResNet18	1024	SGD	0.100	0.000	0.900	✗	1000	0.001	✓	✓	✗	0.902	0.967	0.100	0.376	0.074	0.763	9.301	12.220
CIFAR10	✓	✗	ResNet18	1024	SGD	0.100	0.000	0.000	✗	1000	0.001	✓	✓	✗	0.904	0.992	0.154	0.337	0.398	0.758	7.877	20.599
CIFAR10	✓	✗	ResNet18	1024	SGD	0.100	0.000	0.900	✗	1000	0.000	✓	✓	✗	0.856	0.987	0.055	0.730	0.070	1.912	14.964	39.823
CIFAR10	✓	✗	ResNet18	1024	SGD	0.100	0.000	0.000	✗	1000	0.000	✓	✓	✗	0.905	0.995	0.118	0.320	0.327	0.738	8.202	43.800
CIFAR10	✗	✗	SimpleDLA	1024	SGD	0.100	0.000	0.900	✗	150	0.001	✓	✓	✗	0.939	1.000	0.002	0.242	0.002	0.094	2.296	11.669
CIFAR10	✓	✗	SimpleDLA	1024	SGD	0.100	0.000	0.900	✗	150	0.001	✓	✓	✗	0.880	0.945	0.157	0.438	0.224	1.282	9.091	11.129

Continues on next page...

Dataset	Batchwise filtering	Epochwise filtering	Model	Batch size	Optimizer	Learning rate	Gamma	Momentum	Scheduler	Num epochs	Weight decay	Ran to completion	Data augment	Milestone per nth step	Test accuracy	Train accuracy	Train mean loss	Test mean loss	Train q=0.9 loss	Test q=0.9 loss	Train worst loss	Test worst loss
CIFAR10	✗	✗	SimpleDLA	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✓	✗	0.947	1.000	0.001	0.236	0.002	0.030	0.429	9.438
CIFAR10	✓	✗	SimpleDLA	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✓	✗	0.889	0.953	0.146	0.405	0.155	1.034	9.094	11.192
CIFAR100	✗	✗	FCNet100	512	SGD	0.010	0.995	0.900	exp	500	0.001	✓	✓	✗	0.107	0.113	3.969	4.022	5.923	6.018	16.711	16.392
CIFAR100	✓	✗	FCNet100	512	SGD	0.010	0.995	0.900	exp	500	0.001	✓	✓	✗	0.094	0.100	3.944	3.995	5.567	5.641	17.227	16.989
CIFAR100	✗	✗	FCNet1000	512	SGD	0.010	0.995	0.900	exp	500	0.001	✓	✓	✗	0.134	0.137	3.871	3.940	6.048	6.162	23.154	18.758
CIFAR100	✓	✗	FCNet1000	512	SGD	0.010	0.995	0.900	exp	500	0.001	✓	✓	✗	0.109	0.114	3.880	3.935	5.670	5.760	18.734	18.176
CIFAR100	✗	✗	FCNet3000	512	SGD	0.010	0.995	0.900	exp	500	0.001	✓	✓	✗	0.138	0.143	3.854	3.931	6.050	6.185	23.568	20.028
CIFAR100	✓	✗	FCNet3000	512	SGD	0.010	0.995	0.900	exp	500	0.001	✓	✓	✗	0.115	0.119	3.848	3.904	5.636	5.705	19.525	17.375
CIFAR100	✗	✗	ResNet18	1024	SGD	0.100	0.990	0.900	exp	500	0.001	✓	✓	✗	0.742	1.000	0.005	1.123	0.009	3.997	1.146	12.190
CIFAR100	✓	✗	ResNet18	1024	SGD	0.100	0.990	0.900	exp	500	0.001	✓	✓	✗	0.739	1.000	0.005	1.133	0.010	4.076	0.919	12.878
CIFAR100	✗	✗	ResNet18	1024	SGD	0.100	0.995	0.900	exp	1000	0.001	✓	✓	✗	0.766	1.000	0.005	1.124	0.008	4.124	0.790	11.701
CIFAR100	✗	✗	ResNet18	1024	SGD	0.100	0.995	0.000	exp	1000	0.001	✓	✓	✗	0.705	1.000	0.006	1.262	0.014	4.739	1.088	15.157
CIFAR100	✗	✗	ResNet18	1024	SGD	0.100	0.995	0.900	exp	1000	0.000	✓	✓	✗	0.718	1.000	0.000	1.825	0.000	7.129	1.330	30.570
CIFAR100	✗	✗	ResNet18	1024	SGD	0.100	0.995	0.000	exp	1000	0.000	✓	✓	✗	0.692	1.000	0.002	1.711	0.004	6.503	1.386	23.564
CIFAR100	✓	✗	ResNet18	1024	SGD	0.100	0.995	0.900	exp	1000	0.001	✓	✓	✗	0.696	1.000	0.161	1.099	0.370	3.389	1.864	14.666
CIFAR100	✓	✗	ResNet18	1024	SGD	0.100	0.995	0.000	exp	1000	0.001	✓	✓	✗	0.662	0.998	0.272	1.232	0.590	3.638	2.233	15.049
CIFAR100	✓	✗	ResNet18	1024	SGD	0.100	0.995	0.900	exp	1000	0.000	✓	✓	✗	0.684	1.000	0.129	1.185	0.312	3.763	1.469	18.031
CIFAR100	✓	✗	ResNet18	1024	SGD	0.100	0.995	0.000	exp	1000	0.000	✓	✓	✗	0.659	0.999	0.257	1.253	0.566	3.764	1.834	12.905
CIFAR100	✗	✗	ResNet18	1024	SGD	0.100	0.100	0.900	ms	500	0.001	✓	✓	200	0.764	1.000	0.006	1.025	0.012	3.765	1.367	13.607
CIFAR100	✓	✗	ResNet18	1024	SGD	0.100	0.100	0.900	ms	500	0.001	✓	✓	200	0.747	1.000	0.004	1.057	0.009	4.022	1.398	15.784
CIFAR100	✗	✗	ResNet18	1024	SGD	0.100	0.500	0.900	ms	1000	0.001	✓	✓	150	0.771	1.000	0.004	1.098	0.007	4.082	0.847	10.819
CIFAR100	✗	✓	ResNet18	1024	SGD	0.100	0.500	0.900	ms	1000	0.001	✓	✗	150	0.485	0.766	1.068	2.160	3.540	4.517	9.933	10.185
CIFAR100	✗	✓	ResNet18	1024	SGD	0.100	0.500	0.900	ms	1000	0.001	✓	✓	150	0.658	0.788	0.863	1.414	3.353	4.378	14.151	13.252
CIFAR100	✗	✗	ResNet18	1024	SGD	0.100	0.500	0.000	ms	1000	0.001	✓	✓	150	0.709	1.000	0.005	1.189	0.012	4.278	1.266	17.223
CIFAR100	✗	✗	ResNet18	1024	SGD	0.100	0.500	0.900	ms	1000	0.000	✓	✓	150	0.719	1.000	0.000	1.921	0.000	7.573	1.132	33.370
CIFAR100	✗	✗	ResNet18	1024	SGD	0.100	0.500	0.000	ms	1000	0.000	✓	✓	150	0.701	1.000	0.001	1.664	0.002	6.428	1.157	23.444
CIFAR100	✓	✓	ResNet18	1024	SGD	0.100	0.500	0.900	ms	1000	0.001	✓	✗	150	0.434	0.683	1.292	2.310	4.071	5.658	17.225	14.992
CIFAR100	✓	✓	ResNet18	1024	SGD	0.100	0.500	0.900	ms	1000	0.001	✓	✓	150	0.609	0.765	0.903	1.462	2.784	4.282	18.311	15.836
CIFAR100	✓	✗	ResNet18	1024	SGD	0.100	0.500	0.900	ms	1000	0.001	✓	✗	150	0.512	1.000	0.099	1.969	0.273	5.323	1.561	16.287
CIFAR100	✓	✗	ResNet18	1024	SGD	0.100	0.500	0.000	ms	1000	0.001	✓	✓	150	0.676	1.000	0.196	1.162	0.455	3.485	1.802	14.123
CIFAR100	✓	✗	ResNet18	1024	SGD	0.100	0.500	0.900	ms	1000	0.000	✓	✓	150	0.693	1.000	0.099	1.165	0.248	3.751	1.433	15.486
CIFAR100	✓	✗	ResNet18	1024	SGD	0.100	0.500	0.000	ms	1000	0.000	✓	✓	150	0.681	0.999	0.182	1.163	0.436	3.568	1.539	14.332
CIFAR100	✓	✗	ResNet18	1024	SGD	0.100	0.500	0.900	ms	1000	0.001	✓	✓	150	0.698	1.000	0.135	1.090	0.323	3.410	1.305	16.066
CIFAR100	✗	✗	ResNet18	16	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✓	✗	0.145	0.144	3.590	3.602	5.474	5.490	15.282	12.813
CIFAR100	✓	✗	ResNet18	16	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✓	✗	0.146	0.145	3.583	3.585	5.270	5.272	11.979	12.108
CIFAR100	✗	✗	ResNet18	128	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✓	✗	0.425	0.462	2.011	2.185	4.645	4.947	12.182	13.316
CIFAR100	✓	✗	ResNet18	128	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✓	✗	0.470	0.539	1.635	1.932	3.455	4.116	10.297	10.369
CIFAR100	✗	✗	ResNet18	512	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✓	✗	0.480	0.631	1.338	2.339	4.079	6.437	15.471	15.910
CIFAR100	✓	✗	ResNet18	512	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✓	✗	0.565	0.885	0.478	1.647	1.190	4.413	9.125	14.732
CIFAR100	✗	✗	ResNet18	1024	SGD	0.100	0.000	0.900	✗	150	0.001	✓	✓	✗	0.743	1.000	0.008	1.028	0.019	3.753	1.756	17.252
CIFAR100	✓	✗	ResNet18	1024	SGD	0.100	0.000	0.900	✗	150	0.001	✓	✓	✗	0.658	0.929	0.239	1.516	0.676	5.250	8.866	15.076
CIFAR100	✗	✗	ResNet18	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✓	✗	0.512	0.790	0.704	2.371	2.407	6.920	14.541	19.328
CIFAR100	✗	✗	ResNet18	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✓	✗	0.758	1.000	0.006	0.994	0.013	3.703	1.479	15.377
CIFAR100	✓	✗	ResNet18	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✓	✗	0.560	0.924	0.388	1.659	0.957	4.484	9.300	15.359
CIFAR100	✓	✗	ResNet18	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✓	✗	0.663	0.944	0.194	1.517	0.505	5.350	9.973	17.731
CIFAR100	✗	✗	ResNet18	1024	SGD	0.100	0.000	0.900	✗	1000	0.001	✓	✓	✗	0.671	0.955	0.157	1.552	0.338	5.441	10.479	20.424
CIFAR100	✗	✗	ResNet18	1024	SGD	0.100	0.000	0.000	✗	1000	0.001	✓	✓	✗	0.743	1.000	0.004	1.111	0.008	4.130	6.993	13.056
CIFAR100	✗	✗	ResNet18	1024	SGD	0.100	0.000	0.900	✗	1000	0.000	✓	✓	✗	0.721	1.000	0.000	2.019	0.000	8.080	0.905	35.603
CIFAR100	✗	✗	ResNet18	1024	SGD	0.100	0.000	0.000	✗	1000	0.000	✓	✓	✗	0.706	1.000	0.000	1.838	0.000	7.123	1.142	24.977
CIFAR100	✓	✗	ResNet18	1024	SGD	0.100	0.000	0.900	✗	1000	0.000	✓	✓	✗	0.690	0.998	0.063	1.265	0.170	4.377	1.875	21.555
CIFAR100	✓	✗	ResNet18	1024	SGD	0.100	0.000	0.000	✗	1000	0.001	✓	✓	✗	0.683	0.997	0.094	1.206	0.245	3.964	4.807	15.065
CIFAR100	✓	✗	ResNet18	1024	SGD	0.100	0.000	0.000	✗	1000	0.000	✓	✓	✗	0.678	0.996	0.095	1.255	0.256	4.161	3.603	16.155
CIFAR100	✗	✗	SimpleDLA	1024	SGD	0.100	0.000	0.900	✗	150	0.001	✓	✓	✗	0.733	1.000	0.007	1.126	0.016	4.277	1.996	18.172
CIFAR100	✓	✗	SimpleDLA	1024	SGD	0.100	0.000	0.900	✗	150	0.001	✓	✓	✗	0.614	0.896	0.342	1.736	1.066	5.763	12.045	20.056
CIFAR100	✗	✗	SimpleDLA	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✓	✗	0.757	1.000	0.005	1.004	0.010	3.906	1.060	12.400
CIFAR100	✓	✗	SimpleDLA	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✓	✗	0.614	0.871	0.440	1.829	1.456	5.970	15.015	20.649
MNIST	✗	✓	DLA	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✗	✗	0.986	0.989	0.039	0.040	0.003	0.002	17.521	10.427
MNIST	✓	✓	DLA	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✗	✗	0.977	0.977	0.155	0.154	0.343	0.338	8.153	5.790
MNIST	✓	✗	DLA	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✗	✗	0.988	0.994	0.096	0.108	0.215	0.227	3.978	3.945
MNIST	✗	✗	FCNet100	256	SGD	0.010	0.995	0.900	exp	500	0.001	✓	✗	✗	0.978	0.994	0.037	0.069	0.066	0.076	9.663	7.282
MNIST	✓	✗	FCNet100	256	SGD	0.010	0.995	0.900	exp	500	0.001	✓	✗	✗	0.965	0.999	0.483	0.487	0.878	0.873	1.885	3.758
MNIST	✗	✗	FCNet1000	256	SGD	0.010	0.995	0.900	exp	500	0.001	✓	✗	✗	0.982	0.996	0.032	0.062	0.060	0.068	8.666	6.462
MNIST	✓	✗	FCNet1000	256	SGD	0.010	0.995	0.900	exp	500	0.001	✓	✗	✗	0.968	1.000	0.605	0.608	0.984	0.986	1.932	3.208
MNIST	✗	✗	FCNet3000	256	SGD	0.010	0.995	0.900	exp	500	0.001	✓	✗	✗	0.982	0.996	0.031	0.061	0.060	0.069	8.322	6.629

Continues on next page...

Dataset	Batchwise filtering	Epochwise filtering	Model	Batch size	Optimizer	Learning rate	Gamma	Momentum	Scheduler	Num epochs	Weight decay	Ran to completion	Data augment	Milestone per nth step	Test accuracy	Train accuracy	Train mean loss	Test mean loss	Train q=0.9 loss	Test q=0.9 loss	Train worst loss	Test worst loss
MNIST	✓	✗	FCNet3000	256	SGD	0.010	0.995	0.900	exp	500	0.001	✓	✗	✗	0.968	1.000	0.621	0.623	0.997	0.998	1.902	3.135
MNIST	✗	✓	LeNet	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✗	✗	0.964	0.968	0.140	0.156	0.008	0.007	27.821	30.966
MNIST	✗	✗	LeNet	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✗	✗	0.990	0.999	0.005	0.031	0.001	0.002	3.272	12.640
MNIST	✗	✗	LeNet	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✗	✗	0.991	1.000	0.004	0.028	0.002	0.002	6.344	10.296
MNIST	✗	✓	LeNet	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✗	✗	0.113	0.112	2.301	2.301	2.327	2.327	2.401	2.401
MNIST	✗	✗	LeNet	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✗	✗	0.113	0.112	0.112	0.113	2.329	2.329	0.112	0.113
MNIST	✓	✓	LeNet	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✗	✗	0.952	0.953	0.213	0.213	0.547	0.551	13.197	9.984
MNIST	✓	✗	LeNet	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✗	✗	0.988	0.998	0.072	0.086	0.194	0.195	2.664	4.684
MNIST	✓	✗	LeNet	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✗	✗	0.989	0.999	0.109	0.118	0.282	0.281	2.063	4.411
MNIST	✓	✓	LeNet	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✗	✗	0.973	0.970	0.222	0.215	0.508	0.494	10.290	5.299
MNIST	✓	✗	LeNet	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✗	✗	0.990	0.998	0.998	0.990	0.301	0.300	0.998	0.990
MNIST	✗	✓	ResNet18	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✗	✗	0.980	0.979	0.070	0.063	0.029	0.025	13.521	7.334
MNIST	✗	✗	ResNet18	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✗	✗	0.988	0.996	0.016	0.044	0.010	0.010	8.224	10.766
MNIST	✗	✗	ResNet18	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✗	✗	0.995	1.000	0.001	0.016	0.001	0.001	0.051	8.250
MNIST	✓	✓	ResNet18	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✗	✗	0.967	0.964	0.258	0.255	0.538	0.526	6.954	5.182
MNIST	✓	✗	ResNet18	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✗	✗	0.994	0.998	0.084	0.091	0.191	0.189	1.860	3.371
MNIST	✓	✗	ResNet18	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✗	✗	0.993	0.999	0.149	0.156	0.336	0.335	2.453	10.378
MNIST	✗	✗	SimpleDLA	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✗	✗	0.996	1.000	0.001	0.015	0.002	0.002	0.021	7.738
MNIST	✓	✗	SimpleDLA	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✓	✗	✗	0.993	0.999	0.063	0.074	0.147	0.154	1.205	3.006
CIFAR10	✗	✗	FCNet3000	512	Adam	0.010	0.995	0.900	exp	500	0.001	✓	✓	✗	0.280	0.271	2.714	2.703	5.336	5.357	32.401	27.625
CIFAR10	✓	✗	FCNet3000	512	Adam	0.010	0.995	0.900	exp	500	0.001	✓	✓	✗	0.239	0.237	2.366	2.363	4.007	3.996	13.864	13.392
CIFAR10	✗	✗	ResNet18	512	Adam	0.000	0.995	0.900	exp	300	0.001	✓	✓	✗	0.856	0.971	0.083	0.518	0.121	1.722	7.750	15.244
CIFAR10	✓	✗	ResNet18	512	Adam	0.000	0.995	0.900	exp	300	0.001	✓	✓	✗	0.762	0.848	0.756	0.898	1.103	1.372	5.587	10.844
CIFAR10	✗	✗	ResNet18	1024	Adam	0.010	0.995	0.900	exp	1000	0.001	✓	✓	✗	0.905	0.998	0.007	0.515	0.003	0.592	5.206	13.261
CIFAR10	✗	✗	ResNet18	1024	Adam	0.010	0.995	0.900	exp	1000	0.000	✓	✓	✗	0.931	1.000	0.000	0.859	0.000	0.005	0.017	89.479
CIFAR10	✓	✗	ResNet18	1024	Adam	0.010	0.995	0.900	exp	1000	0.001	✓	✓	✗	0.870	0.972	0.267	0.610	0.683	0.788	5.621	16.823
CIFAR10	✓	✗	ResNet18	1024	Adam	0.010	0.995	0.900	exp	1000	0.000	✓	✓	✗	0.880	0.997	0.128	0.460	0.386	0.902	6.959	24.010
CIFAR10	✗	✗	ResNet18	1024	Adam	0.010	0.500	0.900	ms	1000	0.000	✓	✓	150	0.935	1.000	0.000	0.900	0.000	0.001	0.010	66.934
CIFAR10	✗	✗	ResNet18	1024	Adam	0.010	0.500	0.900	ms	1000	0.001	✓	✓	150	0.899	0.991	0.026	0.494	0.010	0.778	5.669	16.719
CIFAR10	✓	✗	ResNet18	1024	Adam	0.010	0.500	0.900	ms	1000	0.001	✓	✓	150	0.865	0.944	0.339	0.655	0.694	0.757	7.095	17.396
CIFAR10	✓	✗	ResNet18	1024	Adam	0.010	0.500	0.900	ms	1000	0.000	✓	✓	150	0.898	1.000	0.182	0.413	0.474	0.736	2.403	19.141
CIFAR10	✗	✗	ResNet18	512	Adam	0.000	0.000	0.900	✗	300	0.001	✓	✓	✗	0.851	0.952	0.138	0.519	0.250	1.755	10.245	12.286
CIFAR10	✓	✗	ResNet18	512	Adam	0.000	0.000	0.900	✗	300	0.001	✓	✓	✗	0.781	0.862	0.688	0.834	1.008	1.267	6.386	10.810
CIFAR10	✗	✗	ResNet18	1024	Adam	0.010	0.000	0.900	✗	1000	0.001	✓	✓	✗	0.782	0.806	0.585	0.710	1.998	2.504	16.300	13.644
CIFAR10	✗	✗	ResNet18	1024	Adam	0.010	0.000	0.900	✗	1000	0.000	✓	✓	✗	0.924	0.999	0.005	0.805	0.000	0.028	18.560	103.220
CIFAR10	✓	✗	ResNet18	1024	Adam	0.010	0.000	0.900	✗	1000	0.001	✓	✓	✗	0.807	0.828	0.745	0.780	1.006	1.043	7.244	9.975
CIFAR10	✓	✗	ResNet18	1024	Adam	0.010	0.000	0.900	✗	1000	0.000	✓	✓	✗	0.895	0.995	0.064	0.397	0.183	0.797	9.023	19.532
CIFAR100	✗	✗	FCNet3000	512	Adam	0.010	0.995	0.900	exp	500	0.001	✓	✓	✗	0.061	0.063	13.443	13.631	36.039	36.912	206.069	179.007
CIFAR100	✓	✗	FCNet3000	512	Adam	0.010	0.995	0.900	exp	500	0.001	✓	✓	✗	0.051	0.048	13.651	13.745	35.070	35.275	235.672	169.895
CIFAR100	✗	✗	ResNet18	1024	Adam	0.010	0.995	0.900	exp	1000	0.001	✓	✓	✗	0.639	0.935	0.243	1.613	0.713	5.470	7.049	19.080
CIFAR100	✗	✗	ResNet18	1024	Adam	0.010	0.995	0.900	exp	1000	0.000	✓	✓	✗	0.641	1.000	0.000	4.731	0.000	17.627	0.888	73.412
CIFAR100	✓	✗	ResNet18	1024	Adam	0.010	0.995	0.900	exp	1000	0.001	✓	✓	✗	0.635	0.967	0.435	1.341	0.891	3.797	3.122	17.521
CIFAR100	✓	✗	ResNet18	1024	Adam	0.010	0.995	0.900	exp	1000	0.000	✓	✓	✗	0.573	0.988	0.101	2.200	0.249	6.991	12.408	28.764
CIFAR100	✗	✗	ResNet18	1024	Adam	0.010	0.500	0.900	ms	1000	0.001	✓	✓	150	0.637	0.889	0.371	1.544	1.117	5.079	9.023	20.528
CIFAR100	✗	✗	ResNet18	1024	Adam	0.010	0.500	0.900	ms	1000	0.000	✓	✓	150	0.642	1.000	0.000	5.190	0.000	19.032	1.029	94.879
CIFAR100	✓	✗	ResNet18	1024	Adam	0.010	0.500	0.900	ms	1000	0.001	✓	✓	150	0.635	0.947	0.471	1.355	0.973	3.817	4.127	18.957
CIFAR100	✓	✗	ResNet18	1024	Adam	0.010	0.500	0.900	ms	1000	0.000	✓	✓	150	0.582	0.987	0.085	2.318	0.191	7.599	9.711	43.225
CIFAR100	✗	✗	ResNet18	512	Adam	0.000	0.000	0.900	✗	300	0.001	✓	✓	✗	0.549	0.939	0.202	2.228	0.513	6.832	11.559	15.936
CIFAR100	✓	✗	ResNet18	512	Adam	0.000	0.000	0.900	✗	300	0.001	✓	✓	✗	0.531	0.906	0.510	1.751	1.166	4.449	7.361	12.660
CIFAR100	✗	✗	ResNet18	1024	Adam	0.010	0.000	0.900	✗	1000	0.001	✓	✓	✗	0.405	0.427	2.162	2.364	4.881	5.512	17.339	16.486
CIFAR100	✗	✗	ResNet18	1024	Adam	0.010	0.000	0.900	✗	1000	0.000	✓	✓	✗	0.618	0.996	0.013	7.125	0.000	25.151	25.721	147.561
CIFAR100	✓	✗	ResNet18	1024	Adam	0.010	0.000	0.900	✗	1000	0.001	✓	✓	✗	0.558	0.631	1.368	1.569	2.737	3.562	8.437	13.461
CIFAR100	✓	✗	ResNet18	1024	Adam	0.010	0.000	0.900	✗	1000	0.000	✓	✓	✗	0.559	0.967	0.127	3.300	0.130	10.702	23.490	47.198
MNIST	✗	✗	FCNet3000	256	Adam	0.000	0.995	0.900	exp	500	0.001	✓	✗	✗	0.983	0.997	0.028	0.058	0.056	0.063	7.274	6.294
MNIST	✓	✗	FCNet3000	256	Adam	0.000	0.995	0.900	exp	500	0.001	✓	✗	✗	0.965	1.000	0.690	0.693	1.074	1.084	1.907	3.315
CIFAR10	✗	✗	DLA	1024	SGD	0.100	0.500	0.900	ms	1000	0.001	✗	✓	150	0.953	1.000	0.001	0.208	0.002	0.015	0.025	8.921
CIFAR10	✗	✗	ResNet18	1024	SGD	0.100	0.500	0.900	ms	1000	0.001	✗	✗	150	0.828	1.000	0.001	0.710	0.001	2.993	0.004	10.203
CIFAR10	✗	✗	ResNet18	1024	SGD	0.100	0.500	0.900	ms	1000	0.001	✗	✓	150	0.946	1.000	0.001	0.236	0.001	0.038	0.038	9.484
CIFAR100	✗	✗	ResNet18	1024	SGD	0.100	0.500	0.900	ms	1000	0.001	✗	✗	150	0.608	1.000	0.004	1.900	0.005	4.487	0.768	9.867
CIFAR100	✗	✗	ResNet18	1024	SGD	0.100	0.500	0.900	ms	1000	0.001	✗	✓	150	0.768	1.000	0.004	1.086	0.008	4.050	1.003	11.873
CIFAR100	✓	✗	ResNet18	1024	SGD	0.100	0.000	0.900	✗	1000	0.001	✗	✓	✗	0.699	0.999	0.092	1.119	0.241	3.653	2.665	16.029
MNIST	✗	✗	DLA	1024	SGD	0.100	0.000	0.900	✗	300	0.001	✗	✗	✗	0.996	1.000	0.001	0.016	0.002	0.002	0.039	7.763

II.I Supplementary plots

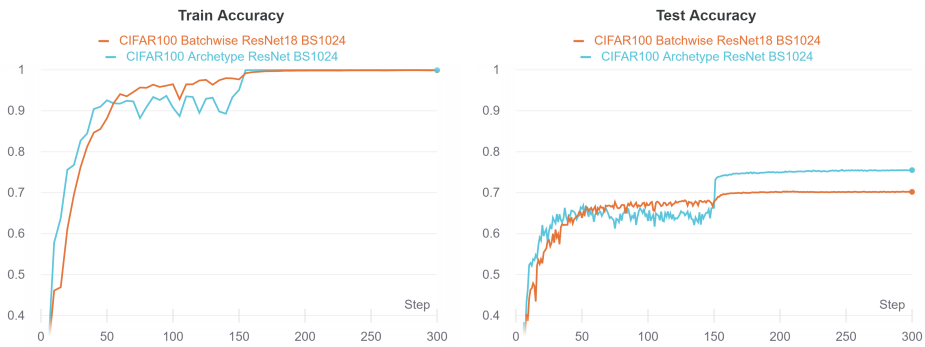
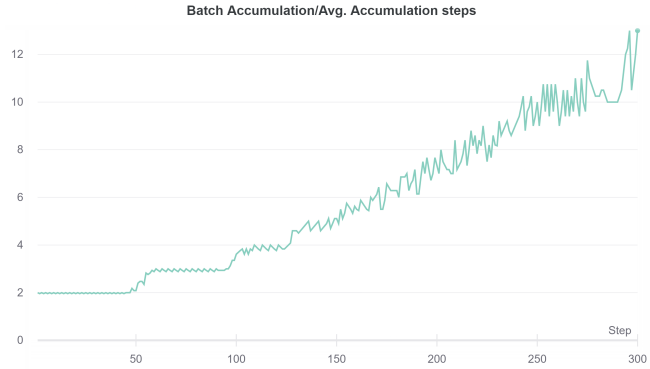
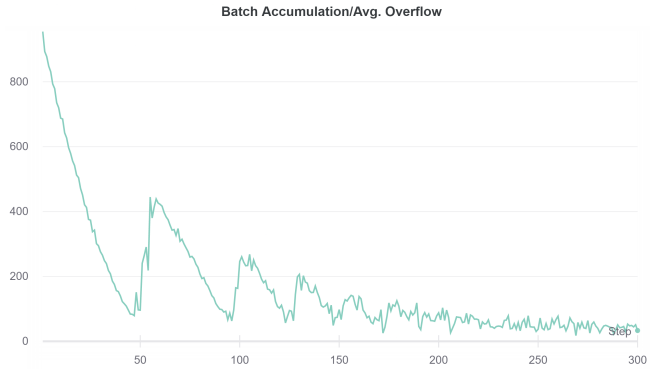


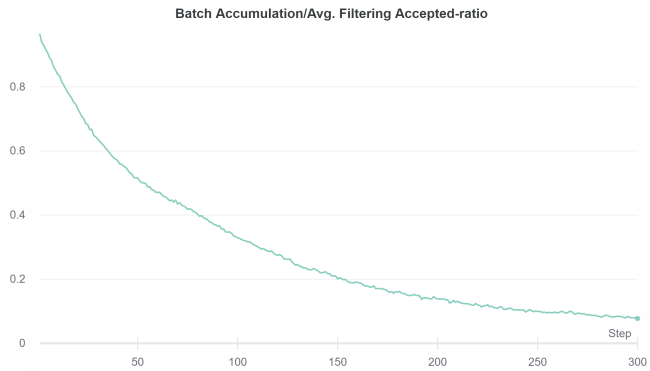
Figure 5: This is the seen behaviour when the CIFAR-100 experiment (see fig. 4.7) is ran with inclusion of a multistep learning rate scheduler. The milestone reached at epoch 150 causes the learning rate to be a 10th of its original value, this causes stabilisation enough for the archetype to continue it's gradient descent. However the batchwise filtering causes our own approach to stagnate at a lower accuracy.



(a) The average number of raw batches of data that were sampled from the dataset in order to construct one filtered batch. The further into training, the higher the chances are of having epochs occur with few or no training steps being performed.



(b) The average number of samples that were passed through the model but were discarded because the batch size was already reached.



(c) The average ratio of samples that passed through the model that ultimately were included in the batch. Through training, this grows smaller and smaller, as the model has gotten better and better, and gradually reject more and more samples.

Figure 6: Various effects of using a batch accumulation procedure.

III Flowcharts & Figures

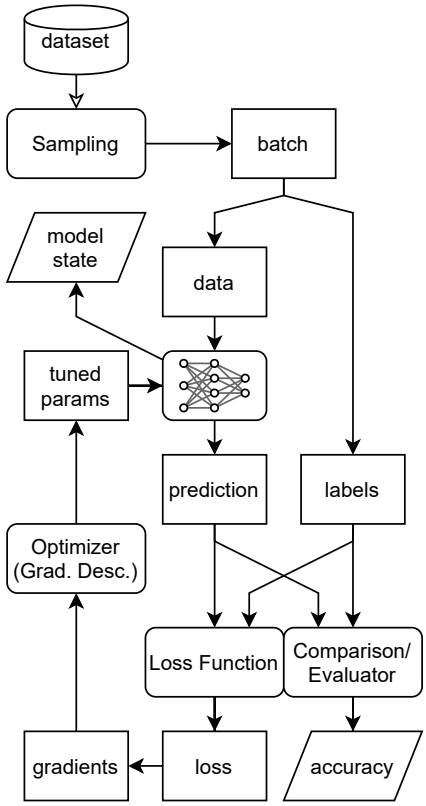


Figure 7: Archetype Trainer Flow

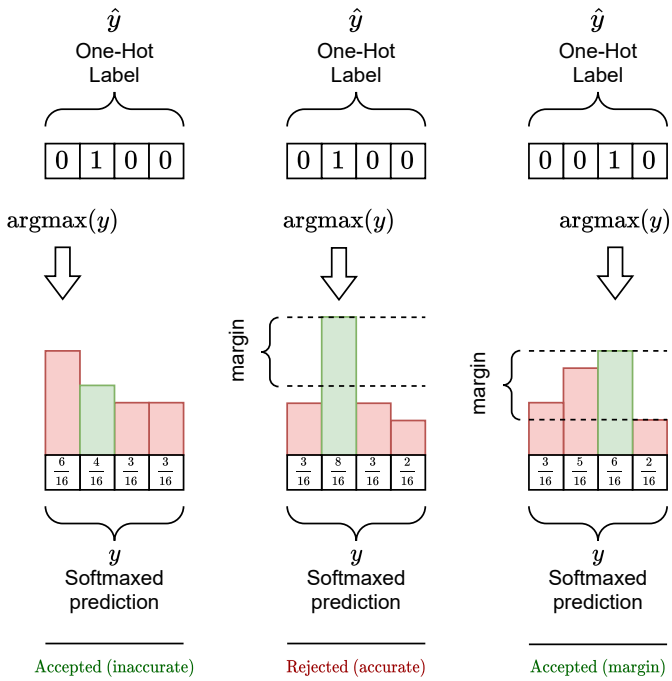


Figure 8: Soft Accuracy Filtering: An example case in which we are working on a 4-class problem. Samples that are either incorrectly classified, or samples that are correctly classified, but not far enough from the decision boundary to be considered properly learnt yet will be accepted. In this case, the left-side sample is accepted, because it's incorrectly classified. The second (in the middle) is rejected, because the sample is correctly classified, and falls well within the decision boundary of the model. The third is accepted, because the network has yet to manage to confidently distinguish it from other classes.