Sharpness Aware Data Poisoning Leads to Flatter Minima that Generalize Better

Alexander Murphy

MInf Project (Part 2) Report
Master of Informatics
School of Informatics
University of Edinburgh

2025

Abstract

At the core of machine learning, lies the hurdle of finding parameters that optimize a loss function. A lot of research has been done on understanding how to find the best parameters and what kind of properties these parameters have. One of those properties is the sharpness of the loss landscape, where research has shown that sharp minima generalize worse than flat minima. Based upon these findings, the Sharpness Aware Minimization (SAM) optimizer was developed to specifically target flat minima. In this dissertation we adopt a data poisoning algorithm to poison datasets, with the aim of leading normal SGD to minima that are flat and thus removing the need of using more compute intensive optimizers such as SAM to find flat minima. We perform experiments on F-MNIST, CIFAR-10 and CIFAR-100 with multiple models. Our results show that SGD does find better minima when training on poisoned datasets and thus using our method could result in considerable savings and performance improvements in a variety of machine learning problems. In addition to that, we perform more qualitative experiments on a toy dataset, to explore properties of our algorithm and identify future research directions. To aid future research we also present a variety of different objectives that our method can apply to, not limiting ourselves to neural network sharpness.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Alexander Murphy*)

Acknowledgements

I want to thank my family for supporting me and motivating me to do my best!

I want to thank my friends in AT9 and beyond, for keeping me from going crazy and helping me survive uni!

I want to thank Pavlos for being an amazing supervisor for the past two years!

Table of Contents

Chapter 1

Introduction

1.1 Optimization in Machine Learning

Nearly all machine learning methods can be formulated as optimizing a function, the loss function, through which parameters are learned. When there is a large amount of parameters and data to optimize for, it is computationally expensive to find a global minimum for the loss function, and thus local methods such as gradient descent are widely used, which find local minima at low computational cost [26].

Many optimization methods have been developed on top of gradient descent, trying to either find parameters faster or find better ones. Unfortunately, a lot of algorithms that try to find better parameters also take a much longer time [9]. Training over multiple runs, be it for new model architectures or hyperparameter searches, exacerbates this increase in training time even more. In an age where training models costs billions of dollars, produces tonnes of carbon emissions and uses millions of liters of water [25], it is of great importance to train models efficiently.

This dissertation introduces a method that aims to improve the optimization of the loss function, not by changing the algorithm used for optimization, but by changing the data used for training. Unlike other data augmentations, where flips or rotations are applied, we present an algorithm that optimizes data directly based on the loss landscape. Once this dataset is created, the goal is for it to be used with a variety of architectures and training techniques while still delivering improved performance, potentially reducing the amount of resources needed to train a model drastically.

1.2 Why optimize data and not weights?

Treating the input as dynamic instead of static has a number of advantages, many of which have yet to be fully explored. One of them is that new data can be created without going the traditional route of collecting data, especially in cases where it might be very expensive or technically challenging to collect more data. Running an algorithm once to create improved data would be very beneficial in those cases. Another reason for to optimize data, is to further our understanding of loss landscapes, their properties, and

optimization in general. What properties does data have that makes it easy/hard to train with? How much data do I need to generalize well? What kinds of data can be modified to make the loss landscape less chaotic and what kinds cannot? All these questions are not well understood and this dissertation hopefully brings some insight, while also raising some new questions.

1.3 Previous Work

MInf Project Part 1, focused on the effect of data poisoning on deepfake detection algorithms. The second part builds upon findings in Part 1 in relation to data poisoning. Some main takeaways from Part 1 were that data poisoning can, change loss landscapes by modifying data, and be transferrable between architectures and models. These changes in the loss landscape were only targeting a single target sample by turning some training data into poisons, successfully making the network misclassify the target. This inspired us to look further into how data poisoning techniques can be applied and combined with other fields, as well as whether we can poison a network by changing the whole dataset it is being trained on. We give a brief overview of data poisoning in the background chapter, in order to allow the reader to understand this work without having read Part 1.

1.4 Contributions

The main contribution of our paper is an algorithm that modifies data in a way that leads to flatter minima when trained on, that generalize better. It is the first algorithm, to the best of our knowledge, which uses data poisoning techniques to modify the loss landscape in such a way. We perform a range of experiments on our algorithm to evaluate it's performance. We also analyze the behavior of our algorithm in detail, leading to findings that can help further research to be done in this area.

We also make our code public with instructions on how to use it and easily extend it to related loss landscape optimization tasks: https://github.com/awesomealex1/minf2.

1.5 Dissertation Structure

Following this introduction, Chapter 2 covers necessary notation, background knowledge about the loss landscape, data poisoning and existing optimization methods for flat minima. In Chapter 3 we introduce our new method, giving the detailed theoretical motivation and the algorithm. Chapter 4 contains our first set of experiments, including methodology, results and discussion. Then in Chapter 5, we explore our proposed algorithm in detail, working with a toy dataset to make conclusions and motivate further improvements. In Chapter 6 we go over a range of improvements and extensions that could be made and frame the problem in a more general and formal way, in order to motivate further research in this area. Finally, we conclude with Chapter 7, giving a final summary of the dissertation and future research needing to be done.

Chapter 2

Background

In this chapter we present and summarize the necessary literature needed to understand our proposed method. Firstly, we go over the loss landscape of neural networks with a focus on sharpness and different measure of sharpness. Then we summarize Sharpness Aware Minimization (SAM), an optimization method that also tries to find flat minima. Finally, we go over data poisoning, with a focus on the gradient alignment algorithm used by a specific data poisoning method.

2.1 Machine Learning and Notation

For the purposes of this dissertation, machine learning is the process of learning a function from data by minimizing a loss function.

The function we learn is parametrized by a vector $\theta \in \mathbb{R}^m$ and denoted by $h_\theta : X \to Y$. We learn from training data $S \subseteq X \times Y$, containing n samples, by minimizing a loss function $\mathcal{L}_S : \mathbb{R}^m \to \mathbb{R}$. The subscript indicates the data over which the loss is defined: $\mathcal{L}_S$ represents loss over the entire training data, while $\mathcal{L}_B$ represents loss over a batch of data. When it is clear through context what the data is, we omit the subscript. The loss on a specific sample and label is given by $\ell_\theta : X \times Y \to \mathbb{R}$, where the subscript is omitted when it is clear which parameters are used. A learning algorithm $A : X \times Y \to \mathbb{R}^m$, finds parameters $A(S) = \theta$ from the training data to minimize $\mathcal{L}_S(\theta)$. Throughout this dissertation, we adapt notation from other literature to be consistent with ours.

2.2 The Loss Landscape

As is common in literature [24] [8], we use the term 'Loss Landscape' to refer to loss functions when talking about their properties and geometry. There are many works analyzing the loss landscape of neural networks, often with a focus on efficient optimization [9] and improved generalization [24] [8].

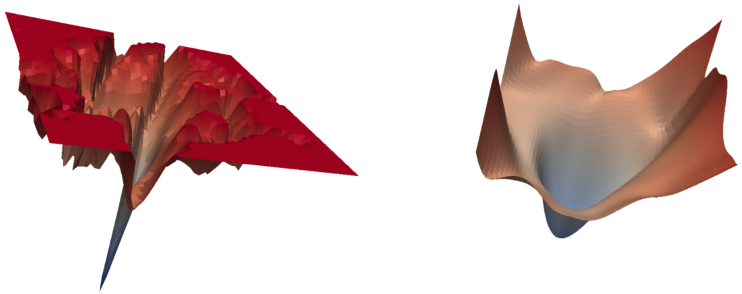


Figure 2.1: A sharp minimum on the left and a flat minimum on the right [9].

2.2.1 Sharpness

A widely used property of loss landscapes is sharpness. The sharpness of a local minimum in the loss landscape is loosely defined as how fast the loss increases when changing the value of the parameters at that point. A minimum where loss increases fast is called sharp, while a minimum where loss increases slowly is called flat [8]. There is no agreed upon way in literature to define sharpness [8], but there are a few proposed methods, each with their own shortcomings.

Volume Sharpness Hochreiter and Schmidhuber [13] define sharpness as the volume of the region around a minimum θ where loss is approximately the same. Formally, they define the m-dimensional hypercube M_θ with center θ. Each edge is parallel to one weight axis. Half the length of the edge in direction parallel to parameter θ_i is denoted as $\Delta\theta_i$. $\Delta\theta_i$ is the largest value such that for all m-dimensional vectors κ where $|\kappa_i| \leq \Delta\theta_i$ we have:

$$\mathcal{L}(\theta) - \mathcal{L}(\theta + \kappa) \leq \varepsilon \tag{2.1}$$

The volume of M_θ is then defined as:

$$V(\Delta\theta) := 2^m \prod_i^m \Delta\theta_i \tag{2.2}$$

Maximum-Loss Sharpness Keskar et al. [19] define sharpness by taking the maximum value the loss can be within a small ε around the minimum. To make sure that a high loss value is achieved not in just a tiny subspace of the parameters, loss is also maximized on random manifolds of the parameter space. To do this, they generate a $m \times p$ matrix W with randomly generated columns, where p determines the dimension of the manifold.

$\mathcal{C}_\varepsilon$ is a box around the minimum where $\mathcal{L}$ is maximized (W^+ is the pseudo-inverse of W, ε is size of the box):

$$\mathcal{C}_\varepsilon = \{z \in \mathbb{R}^p : -\varepsilon(|(W^+\theta)_i| + 1) \leq z_i \leq \varepsilon(|(W^+\theta)_i| + 1) \forall i \in 1, 2, \cdots, p\} \tag{2.3}$$

The sharpness of $\mathcal{L}$ at θ is defined as:

$$\phi_{\theta,\mathcal{L}}(\varepsilon, W) := \frac{(\max_{y \in \mathcal{C}_\varepsilon} \mathcal{L}(\theta + Wy)) - \mathcal{L}(\theta)}{1 + \mathcal{L}(\theta)} \times 100 \tag{2.4}$$

Spectrum of the Hessian Chaudhari et al. [5] use the spectrum of the Hessian (the distribution of the Hessian's eigenvalues) at the minimum to analyze sharpness. As the Hessian is formed of the second order partial derivatives of a function with a vector as a parameter, it characterizes the curvature of a function. Thus, it is natural to use it to describe sharpness. Chaudhari et al. [5] use the spectrum of the Hessian to characterize sharpness. A sharp minimum has a Hessian with a lot of large eigenvalues, compared to a flat minimum, which has small eigenvalues.

The spectra of two different Hessians can be compared in multiple ways. One simple way is to plot the n largest eigenvalues and compare how they are distributed. Another way is to compare the ratio of the largest to the smaller eigenvalues. Foret et al. [9] measure sharpness by calculating $\lambda_{\max}/\lambda_5$ and comparing this ratio where a higher ratio corresponds to a sharper minimum. Because the Hessian is intractable to compute for neural networks (it is quadratic with the number of parameters), approximations are used such as Lanczo's algorithm [28] to compute the most extreme (smallest/largest) eigenvalues of the Hessian without needing to compute the Hessian itself.

2.2.2 Shortcomings of Sharpness Measures

Dinh et al. [8] show that the previous methods are flawed to some extent. Due to inherent symmetries in neural networks, they show that equivalent models can be built that are arbitrarily sharp, and if allowed to parameterize, they can change the loss landscape of a network without affecting generalization.

They show that for a large number of common neural networks, scaling weights, changing the order of how weights are applied, or swapping certain weights with each other, doesn't change the output of the network and is an equivalent network for all possible input values. These transformations do affect all of the sharpness measures we discussed previously to some extent.

Nonetheless, even though theoretically the methods aren't always accurate for describing sharpness, they have successfully been used in literature to compare minima and explain generalization. In practice, a lot of the transformations pointed out by Dinh et al. [8] do not affect networks when training with common algorithms, as literature has kept using them and found correlation between sharpness and generalization to hold [9] [19]. Thus, we will too use the methods in this dissertation, keeping their flaws in mind.

2.2.3 Visualizing sharpness

There are multiple ways to visualize the sharpness of neural networks. They rely on reducing the high dimensional parameter space into a low dimensional 1-, 2- or 3-D representation. In Figure 2.1 we can see the sharpness visualized by reducing the

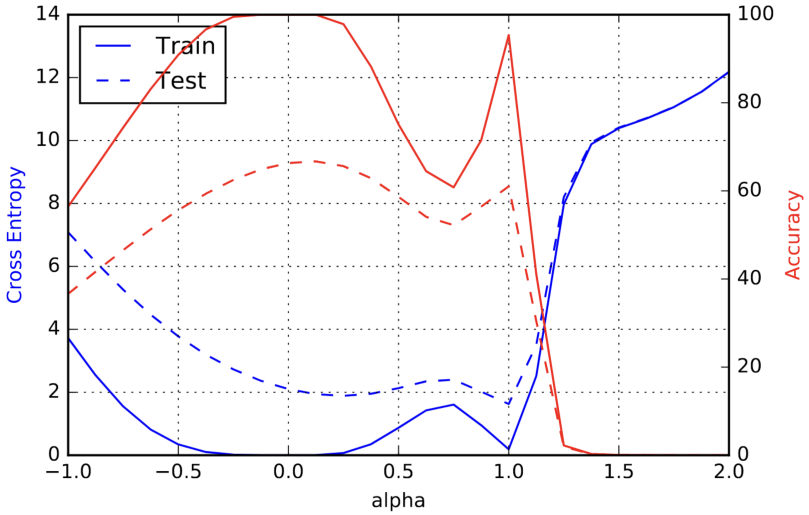


Figure 2.2: A one-dimensional linear interpolation plot showing train/test loss and accuracy.

parameter space to two axes and a third axis representing loss. On the left a very sharp minimum is shown while on the right there is a flatter minimum where the valley where the minimum is located is much wider and less steep.

One-Dimensional Linear Interpolation plots can also be used to compare the sharpness of two different minima. In Figure 2.2 a one-dimensional linear interpolation plot is shown that shows the loss and accuracy on test and training set, where at $\alpha = 0.0$ and $\alpha = 1.0$ the original minima are located. We can see that the minimum at 0.0 is much flatter than at 1.0. The formula used to calculate the weights along the axis is the following:

$$\theta(\alpha) = \alpha\theta_1 + (1-\alpha)\theta_2 \tag{2.5}$$

There are obvious shortcomings to this form of visualizing sharpness. Firstly, they just show the change in loss along the line connecting two minima. This is quite arbitrary and there is no concrete reasoning behind choosing this line to go along, except for that it is computationally easy. There also is the problem of what a sharper minimum represents in a 1-D visualization. The shortcomings of sharpness measures discussed by Dinh et al. [8] apply to one-dimensional linear interpolation plots as well, as weights can be scaled in theory to makes a minimum look sharp or flat. Nevertheless, it is commonly used in literature [19, 24] and thus we use these plots in the rest of the dissertation, keeping limitations in mind.

Filter Normalization can be used to improve the correlation between sharpness and test accuracy when visualizing minima. It addresses some of the scaling issues mentioned by Dinh et al. [8], where a network with large weights is less sensitive to perturbations than a network with small weights, raising the issue of scaling weights by a large number making the minimum less sharp, even though the networks are equivalent. Filter normalization normalizes weights in each filter of convolutional layers and each

neuron in fully connected layers, such that these scaled networks will have a consistent sharpness when visualizing them with one-dimensional linear interpolation for example.

2.2.4 Connecting sharpness and regularization

Flat minima are generally associated with better generalization [19]. It's also known that optimizers can often find flat minima by themselves, without needing explicit guidance to look for them. One theory for this, is that the high dimensional space, that is being optimized over, makes the volume of flat minima much larger than that of sharp ones. Thus, an optimizer has a much higher probability of finding a flat minimum with high volume, than a sharp minimum with low volume [15].

Flat minima have a smaller minimum description length (MDL) than sharp minima, since a lot of bits are needed to precisely describe a sharp minimum, while fewer are needed to describe a sharp one. Thus, we can view a push towards flat minima as a form of regularization, which just as L2 regularization for example, aims to reduce the MDL of the parameters we find. By Occam's razor, this simplest solution, should perform the best, as no noise is fitted and we focus only on relevant information [6].

2.3 Gradient Descent and Backpropagation

The most common optimization algorithm used in machine learning to find minima in the loss landscape is Stochastic Gradient Descent (SGD) or variations of it [33]. The idea behind gradient descent is to calculate the derivative of the loss with respect to the parameters, and then change the parameters in the direction where loss is decreasing. The rate at which to change the parameters is controlled by the learning rate. Because it is very costly to do calculations on the whole dataset, stochastic gradient descent approximates the gradient by computing it with a few randomly chosen samples, called a batch.

$$\theta_{i+1} = \theta_i - \alpha \nabla_\theta \mathcal{L}_B(\theta) \tag{2.6}$$

To calculate the gradient of the loss with respect to the loss function we use the back-propagation algorithm or reverse mode automatic differentiation [3]. This algorithm, computes the gradients for parameters layer-by-layer, utilizing the chain rule. The calculation of the gradient is often the most expensive part of training neural networks and using even higher order methods that calculate the Hessian of the loss is not practical due to the dimensionality of the parameter space and high cost of computing the second derivative.

There are various popular optimization algorithms based on SGD. Some of the most popular ones are SGD with momentum [30] or Adaptive Moment Estimation (Adam) [20]. Momentum adds a "velocity" vector to SGD that accumulates past gradients. This helps accelerate convergence and navigate ravines in the loss landscape more effectively. Adam maintains a moving average of both the mean if the gradients and the variance of the gradients, which works particular well with sparse, noisy gradients.

Batch size is the number of samples used for each gradient calculation. A large batch size will approximate the whole training set better, while a small batch size will have a lot of noise in the estimation. Keskar et al. [19] have observed that in many cases, large batch sizes lead to worse generalization compared to using a small batch size. They find that small batch sizes lead optimizers to flatter minima.

2.4 Sharpness-Aware Minimization

Sharpness-Aware Minimization (SAM) was proposed by Foret et al. [9] to explicitly find minima that are flat, and thus generalize better. The algorithm they describe is a modified version of SGD, but could easily be applied to other optimizers as well, such as Adam [20].

SAM work by performing two backpropagations. Firstly, it calculates the normal gradient of the loss with respect to the parameters. Instead of changing the parameters in the direction of decreasing loss, SAM calculates the following vector $\hat{\varepsilon}(\theta)$:

$$\hat{\varepsilon}(\theta) = \rho \cdot \frac{sign(\nabla_\theta \mathcal{L}_S(\theta))|\nabla_\theta \mathcal{L}_S(\theta)|^{q-1}}{(||\nabla_\theta \mathcal{L}_S(\theta)||_q^q)^{\frac{1}{p}}} \tag{2.7}$$

where $1/q + 1/p = 1$ (note that $|\cdot|^{q-1}$ denotes element wise absolute value and power). Empirically they show that they get the best results for $p = 2$. This means that the equation simplifies to:

$$\hat{\varepsilon}(\theta) = \frac{\rho|\nabla_\theta \mathcal{L}_S(\theta)|}{||\nabla_\theta \mathcal{L}_S(\theta)||_2^2} \tag{2.8}$$

This vector $\hat{\varepsilon}(\theta)$ is then summed with with current weights resulting in temporary weights: $\theta' = \theta + \hat{\varepsilon}(\theta)$. The geometric interpretation of this is, that instead of moving weights into the direction of decreasing loss, we move into the direction of increasing loss.

At the temporary gradients θ' a second backpropagation is performed. The resulting gradient we call $\nabla_\theta \mathcal{L}_S^{SAM}(\theta)$. After computing this gradient, SAM moves back to the original parameters θ and performs a normal SGD update using $\nabla_\theta \mathcal{L}_S^{SAM}(\theta)$ as the gradient.

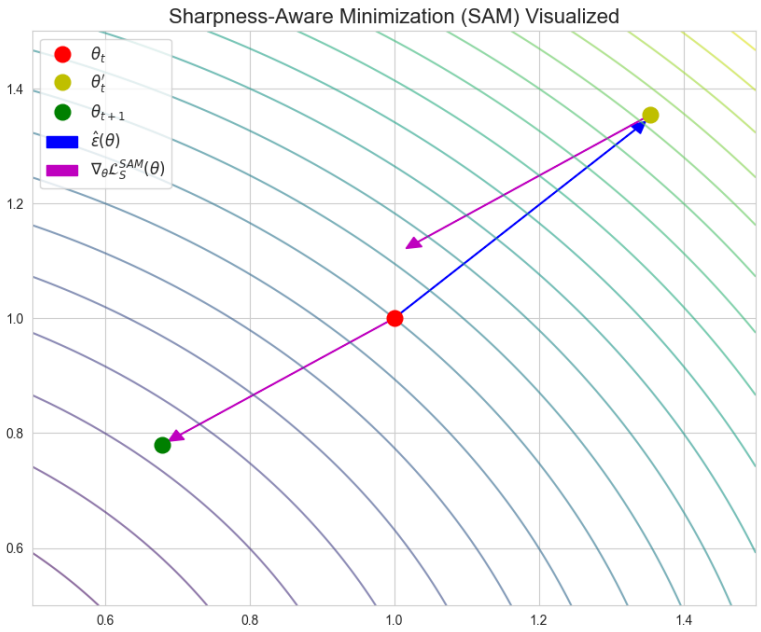


Figure 2.3: A visualization of a SAM update. First loss is maximized (blue arrow). Then the gradient is calculated at the maximized loss. This gradient is then used to perform gradient descent at the original parameters.

In summary, at each iteration, SAM first maximizes loss, then calculates the gradient of the loss with respect to the parameters at this maximum loss, before moving back to the original parameters where the final weight update happens with the second gradient. We illustrated this in Figure 2.3.

2.5 Data Poisoning

Data poisoning is a technique used by adversaries to attack neural networks, discussed heavily in the Part 1 of this dissertation. To summarize, when data poisoning a network, an adversary injects poison data into the training set, which when trained on causes the network to generate outputs, unintended by the owner of the network [29]. This may happen in a targeted way, where a specific input causes a specific output targeted by the adversary or an untargeted attack where the whole accuracy of the network is decreased for example. Geiping et al. [10] use Equation 2.9 to specify the targeted data poisoning problem:

$$min_{\Delta \in \mathcal{C}} \sum_{i=1}^{T} \ell(h_{\theta(\Delta)}(x_i^t), y_i^{adv}) \quad s.t.\ \theta(\Delta) \in argmin_\theta \frac{1}{N} \sum_{i=1}^{N} \ell(h_\theta(x_i + \Delta_i), y_i) \tag{2.9}$$

Here, x_I^t is the ith target and y_I^{adv} is the ith target label, which the adversarial wants the target to be labelled as. Δ is the perturbation added to the training data, turning all perturbed data into poison data. Thus, $\theta(\Delta)$ are the parameters that minimize the

loss when training the network on the poisoned data. Essentially, we want to find the optimal delta, which, after training the network, returns parameters that minimize the loss on the targets.

2.5.1 Gradient Alignment

'Witches Brew', a data poisoning attack by Geiping et al. [10], uses gradient alignment to generate poison data. During gradient alignment it is the goal of the attacker for Equation 2.10 to hold for any θ, since that will cause the minimization of the model's loss on the poison data to coincide with the minimization of the loss between the target and adversarial label. In Equation 2.10, x^t is the target input, y^{adv} is the adversarial label that should be assigned to the target input, P is the number if poisons, Δ_i is the perturbation applied to poison x_i and y_i is the corresponding label of the poison.

$$\nabla_\theta(\ell(x^t, y^{adv})) \approx \frac{1}{P}\sum_{i=1}^{P} \nabla_\theta(\ell(x_i + \Delta_i, y_i)) \tag{2.10}$$

Geiping et al. [10] achieve Equation 2.10 to hold, by optimizing an auxiliary problem. They minimize Equation 2.11, which is the negative cosine similarity between the gradient of the target's loss, and the gradient of the poison's loss. The parameters θ are taken from a clean, already trained model.

$$\mathcal{B}(\Delta, \theta) = 1 - \frac{\langle \nabla_\theta \ell(x^t, y^{adv}), \sum_{i=1}^{P} \nabla_\theta \ell(x_i + \Delta_i, y_i) \rangle}{||\nabla_\theta \ell(x^t, y^{adv})|| \cdot ||\sum_{i=1}^{P} \nabla_\theta \ell(x_i + \Delta_i, y_i)||} \tag{2.11}$$

In this dissertation we will use the gradient alignment poisoning algorithm to generate data with the aim of changing the loss landscape of neural networks to find minima that are flatter and generalize better. We choose gradient alignment, because it is to our knowledge the only poisoning method that uses gradients and thus naturally combines with the gradients used by SAM.

2.5.2 Other Forms of Data Poisoning

Gradient alignment is not the only way to perform data poisoning. In MInf Part 1, we used the Poison Frogs algorithm [32], which uses feature collisions to poison datasets. This data poisoning algorithm is most effective on finetuned models, as it takes advantage of having the same base feature extractor and directly uses the loss function of the feature extractor to collide the target representation in feature space with samples from the adversarial target class. This way, when finetuning only the last layer of a network, the attacker can guarantee the target input to be close to the desired label in feature space.

A similar attack is the convex polytope method [2]. It works similarly to the Poison Frogs attack, but explicitly creates poisons that form a convex polytope around the target sample in feature space. This increases the probability of the target being assigned the adversarial label, as the target is guaranteed to be surrounded by samples from the

adversarial target class, unlike with Poison Frogs, where the target might be on the edge of the adversarial target class. It takes a lot more iteration to generate poisons than Poison Frogs, thus the increased effectiveness comes with a computational trade-off.

Chapter 3

Gradient Alignment for Sharpness-Aware Data Poisoning

In this chapter we present our new method for generating data that assists an optimizer in finding flatter minima. The main idea, is that we want to combine the Sharpness Aware Minimization (SAM) by Foret et al. [9] with the gradient alignment used by Geiping et al. [10]. We want to create poisons that align first order gradients used by SGD with the second order ones used by SAM, with the final goal of finding minima that are flat, but not using a computationally expensive optimizer like SAM.

3.1 Theoretical Motivation

Foret et al. [9] approximate the sharpness aware gradient as follows:

$$\nabla_\theta \mathcal{L}_S^{SAM}(\theta) \approx \nabla_\theta \mathcal{L}_S(\theta)|_{\theta+\hat{\varepsilon}(\theta)} \tag{3.1}$$

where the equation for $\hat{\varepsilon}(\theta)$ is:

$$\hat{\varepsilon}(\theta) = \frac{\rho|\nabla_\theta \mathcal{L}_S(\theta)|}{||\nabla_\theta \mathcal{L}_S(\theta)||_2^2} \tag{3.2}$$

We want to enforce:

$$\nabla_\theta \mathcal{L}_{S'}(\theta) \approx \nabla_\theta \mathcal{L}_S^{SAM}(\theta) \tag{3.3}$$

for any θ where S' is our modified training set, since this would allow us to train on S' with normal SGD and follow the same optimization path achieved by SAM. By using Equation 3.1 we get :

$$\nabla_\theta \mathcal{L}_{S'}(\theta) \approx \nabla_\theta \mathcal{L}_S(\theta)|_{\theta+\hat{\varepsilon}(\theta)} \tag{3.4}$$

Thus we get the following equation for S':

$$S' = \text{argmin}_{S'} ||\nabla_\theta \mathcal{L}_S(\theta)|_{\theta+\hat{\epsilon}(\theta)} - \nabla_\theta \mathcal{L}_{S'}(\theta)|| \tag{3.5}$$

One of the main issues of optimizing this equation directly, is that magnitudes of gradients vary significantly during training. We can use the technique used by Geiping et al. [10], to solve an easier problem instead. We aim to align the gradients to point in the same direction, instead of minimizing the difference between them directly. Just as Geiping et al. [10] we will minimize the negative cosine similarity between them to do this. The following equation is the one we want to minimize:

$$\begin{aligned} \mathcal{B}(\Delta, \theta) &= 1 - \frac{\langle \nabla_\theta \mathcal{L}_S(\theta)|_{\theta+\hat{\epsilon}(\theta)}, \nabla_\theta \mathcal{L}_{S+\Delta}(\theta) \rangle}{||\nabla_\theta \mathcal{L}_S(\theta)|_{\theta+\hat{\epsilon}(\theta)}|| \cdot ||\nabla_\theta \mathcal{L}_{S+\Delta}(\theta)||} \\ &= 1 - \frac{\langle g', g \rangle}{||g'|| \cdot ||g||} \end{aligned} \tag{3.6}$$

where we set $S' = S + \Delta$.

Geiping et al. [10] choose their parameters θ to come from an already trained network, just performing the minimization on those final parameters. This works in the case of data poisoning, because the objective is to change the label that is assigned to the target by the final model, with no objective on training behavior during the beginning or middle stage of training. We on the other hand want to influence training behavior during all epochs. Thus, we choose the sample from models at different stages of training for our algorithm.

3.1.1 Potential shortcomings

A difficulty we might have when trying to use the same method to find poisons, is that if we move to very different parameters, the alignment might decrease and the poisoning becomes ineffective. Considering that we poison the whole dataset instead of just a few samples, there is a higher likelihood of the whole loss landscape changing considerably, and the optimization path we take not being close to the final parameters that the poisons were created with.

We try mitigating this by performing experiments with poisons not only created only on the final trained model, but also created during the whole training process, with the aim of encoding useful properties of SAM from the whole training process into the poison data.

The poisoning process also doesn't guarantee that the loss values on the poison data is the same as on the original data, thus even if the gradients align, the value of the loss might not and we could end up at non-optimal parameters. During normal data poisoning only a few samples are poisoned and thus the overall loss values shouldn't change much, compared to our case where loss values might vary dramatically.

To mitigate this we introduce a hyperparameter ε that limits the magnitude of the poisons, and thus limits the difference in loss between clean data and poisoned data.

3.2 Sharpness-Aware Data Poisoning Algorithm

Using the theoretical foundation we developed in the previous section we now present the final algorithm for sharpness aware data poisoning. The algorithm is split into two parts. Firstly the gradient computation that happens in the normal training loop, similarly to when training with normal SAM. The second part happens after the sharpness aware gradient has been computed, but weight haven't been updated yet. This is the part that created perturbations to add to the batch, such that vanilla SGD matches the SAM gradient.

The following pseudocode outlines the training loop that generates the SAM gradient and calls the poisoning. The inputs for the algorithm are the training set S and the model's loss $\mathcal{L}$. We then train as with normal SAM, but after computing the sharpness aware gradient g' we do not update the weights directly, but call the data poisoning algorithm first. This algorithm takes as input, the perturbations associated with each sample in the batch Δ^B, the current weights θ_i and the sharpness aware gradient g'.

Input: S, $\mathcal{L}$
Initialize $\theta_0 = 0$, $t = 0$
Initialize Δ to random noise
while not converged **do**
  Sample batch B from S
  Compute gradient $g = \nabla_\theta \mathcal{L}_B(\theta_i)$
  Compute $\hat{\varepsilon}(\theta_i)$ per Equation 3.2
  Compute $g' = \nabla_\theta \mathcal{L}_B(\theta_i)|_{\theta_i + \varepsilon(\hat{\theta}_i)}$
  Compute $\Delta^B = \text{argmin}_{\Delta^B} \mathcal{B}(\Delta^B, \theta_i)$
  Set $\theta_{t+1} = \theta_t - \eta g'$
  Set $t = t + 1$
end while

The following pseudocode is for minimizing cosine similarity between g' and $\nabla_\theta \mathcal{L}_{B+\Delta}(\theta)$ which we call $\mathcal{B}(\Delta^B, \theta_i)$ in the pseudocode above and in Equation 3.6. The algorithm takes as input the SAM gradient g', the model's loss $\mathcal{L}$, the weights θ, the batch B, the maximum number of iterations n and the batch's perturbation vector Δ^B. There also are the poison learning rate α and perturbation maximum ε. It outputs the perturbation for the batch, modified to reflect poisoning against the SAM gradient g'.

The algorithm performs the same optimization procedure for n steps or until the perturbation vector doesn't change between iterations. During each step of the optimization, the gradient of the loss on the poisoned batch $B + \Delta^B$ is computed with respect to the weights. This is the gradient that SGD would use when training on the poisoned dataset. Then, the cosine similarity between the poison gradient g and the SAM gradient g' is computed. After that the gradient of the cosine similarity with respect to the perturbation Δ^B is computed which then is used by SGD with learning rate α to minimize $\mathcal{B}(\Delta^B, \theta)$

with respect to Δ^B. After this, an optional projection step is called to constrain how big the perturbation is allowed to be, limited by ε.

Input: g', $\mathcal{L}$, θ, B, n, Δ^B, α, ε
Output: Δ^B
$i = 0$
while not converged and $i < n$ **do**
  Compute gradient $g = \nabla_\theta \mathcal{L}_{B+\Delta^B}(\theta)$
  Compute $\mathcal{B}(\Delta^B, \theta)$ between g and g' according to Equation 3.6
  Compute $\nabla_{\Delta^B} \mathcal{B}(\Delta^B, \theta)$ and minimize $\mathcal{B}(\Delta^B, \theta)$ with one step of SGD
  Project Δ^B onto C
  $i = i + 1$
end while

The final algorithm is very compute intensive, as for each batch that is used during training, the optimization procedure is called, which performs up to n backpropagations. If we have m batches in S, training for e epochs, the final runtime complexity of the algorithm is $O(e \cdot m \cdot n)$. It is to be noted that in a lot of settings the optimization algorithm converges after as few as 2 steps though, making the complexity $O(m \cdot n)$, the same as normal training of a model with SGD. If we assume that convergence happens after 2 steps, then it takes a total of 4 backpropagations per batch (2 for SAM + 2 for optimization) which is equivalent to 4 training runs with SGD.

Chapter 4

Data Poisoning on a range of datasets and models

In this chapter we run our algorithm on a range of different datasets and models. We first introduce the datasets, models and metrics that we will be using. The first experiments we run is a hyperparameter search to tune our poisoning algorithm. Using the hyperparameters we found, we run experiments on a diverse set of datasets and models, investigating the performance of the poisoning algorithm without having experiment specific hyperparameters.

4.1 Methodology

We use a similar combination of datasets and models as used by Foret et al. [9], as their experiments have shown substantial improvements using SAM and they are commonly used datasets and models in computer vision and ML theory literature.

4.1.1 Datasets

We use 4 different image classification datasets in our experiments.

MNIST [7] is a dataset of 60,000 training and 10,000 test greyscale images (28×28 pixels) of handwritten digits (0-9), with balanced class distribution.

Fashion-MNIST [36] maintains MNIST's structure (60,000 training/10,000 test, 28×28 grayscale images, 10 classes) but replaces digits with clothing items, designed to provide a more challenging benchmark.

CIFAR-10 [21] contains 60,000 RGB images (32×32 pixels, not 28×28) across 10 object classes, with 6,000 images per class, split into 50,000 training and 10,000 test samples.

CIFAR-100 [21] uses the same format as CIFAR-10 but features 100 classes organized in a two-level hierarchy, with 600 images per class, maintaining the 50,000/10,000 train/test split.

4.1.2 Models

We use 4 different kinds of convolutional networks in our experiments.

LeNet-5 [23] is a 7-layer CNN developed for handwritten digit recognition and trained on the MNIST dataset. It is very simple, but achieves high accuracy on MNIST.

ResNet [12] is a residual convolutional network that introduced skip connections (or "shortcuts") to allow information to bypass layers. This helps solve the vanishing gradient problem in deep networks, enabling training of networks with hundreds of layers.

WideResNet [37] modified the original ResNet by making it wider rather than deeper, increasing the number of channels in each layer while decreasing depth. This approach improves performance with fewer layers, reducing training time while maintaining accuracy.

DenseNet [14] connects each layer to every other layer in a feed-forward fashion. Unlike ResNet's additive skip connections, DenseNet concatenates feature maps from previous layers, promoting feature reuse and improving gradient flow. This architecture requires fewer parameters while achieving competitive performance.

4.1.3 Metrics

We use a variety of metrics to evaluate our experiments. Some of the metrics we use are for evaluating performance and others for sharpness calculations. The sharpness metrics will only be used in Chapter 5 in order to save computational cost, and as the focus of Chapter 4 is whether poisoning can improve performance.

Accuracy To measure the final performance of a model in computer vision, accuracy is used, which is the percentage of classes which are assigned the correct label by the model. Accuracy can be be computed over any collection of samples. A common way to divide data in machine learning is to use training, validation and test sets. Where training sets are used to train a model, validation sets are used to find hyperparameters or aid development in some way and test sets are only used to evaluate final model performance. This split is done to prevent overfitting and gain a sense of how the model would perform on truly unseen, new data.

Loss Similar to accuracy, loss is used to measure performance of a network, but it is the value calculated by the loss function on the outputs of the network directly. The loss function commonly is mean square error or cross entropy, which we use in our experiments, as is common in literature [12] [37] [14]. Due to logging errors we do not have loss values for experiments in Chapter 4, only in Chapter 5. Thus, accuracy is used in Chapter 4, which gives a very good approximation of the loss.

Spectrum of the Hessian The spectrum of the Hessian is the distribution of it's eigenvalues. An approximation of the largest eigenvalues is commonly used to measure sharpness [9]. We use implementation of Golmant et al. [11] for Lanczo's algorithm [22]. We compute λ_{max}/λ_5 to get a numerical representation of the spectrum, a commonly used approximation for sharpness [8] [16], in addition to plotting the eigenvalues.

One-dimensional linear interpolation. As presented in Chapter 2, we use one-dimensional linear interpolation loss plots [19] to visually represent sharpness. It also has the added benefit of being less computationally expensive compared to calculating the Hessian's eigenvalues.

4.2 Hyperparameter Search

The first experiment we do is to find out what parameters to use in the data poisoning algorithm. We choose relatively "simple" dataset and model in the interest of saving compute. We choose a computer vision task, as it is common in machine learning theory and optimization literature [19, 9, 8]. The follow up experiments that will use the hyperparameters found are also computer vision tasks and thus hyperparameters should hopefully translate between tasks well. The benefit of finding hyperparameters that work well across different datasets is a combination of saving us compute for follow up experiments, as well as giving others a starting point to try out our method.

4.2.1 Setup

Dataset	**Model**	**LR^P**	**ε (%)**	**Iterations**	**Start Epoch**
F-MNIST	ResNet18	0.001, 0.01, 0.1	0, 5	25, 50	0, 10

Table 4.1: Hyperparameter search space for poisoning experiments on Fashion-MNIST with ResNet18. LR^P represents the learning rate for poisoning, ε indicates perturbation magnitude, Iterations shows the number of poisoning steps, and Start Epoch denotes when poisoning begins during training.

In Table 4.1 we outline the different parameters we try. We run experiments on all possible combinations of the hyperparameters, resulting in 24 different runs. Unfortunately due to compute constraints we aren't able to run multiple repeats or try a wider range of parameters. We perform the search on F-MNIST with a ResNet-18.

To give a fair comparison between a baseline and our method we do the following:

1. We perform an initial hyperparameter search on the model to find learning rate and momentum.
2. We then use the values found to perform the hyperparameter search on the augmentation algorithm, with budget $\mathcal{B}$.
3. We perform another hyperparameter search to find better learning rates and momentum with budget $\mathcal{B}$. These values are not used by the poisoning algorithm, but to train models that we compare the poisoning performance against.

This procedure aims to give a fair comparison on whether our poisoning method works better due to the algorithm and not due to allocating more compute to hyperparameter searches. The hyperparameter search used to find initial parameters is the following. We search search for learning rate and momentum values, with the minimum learning

rate being 10^{-4} and the maximum being 0.1. The momentum is chosen to be between 0.01 and 0.99. We train for 100 epochs using SGD with momentum and a batch size of 64. We use the same setup to perform the second hyperparameter search as well, which we compare the final poison results against.

4.2.2 Results

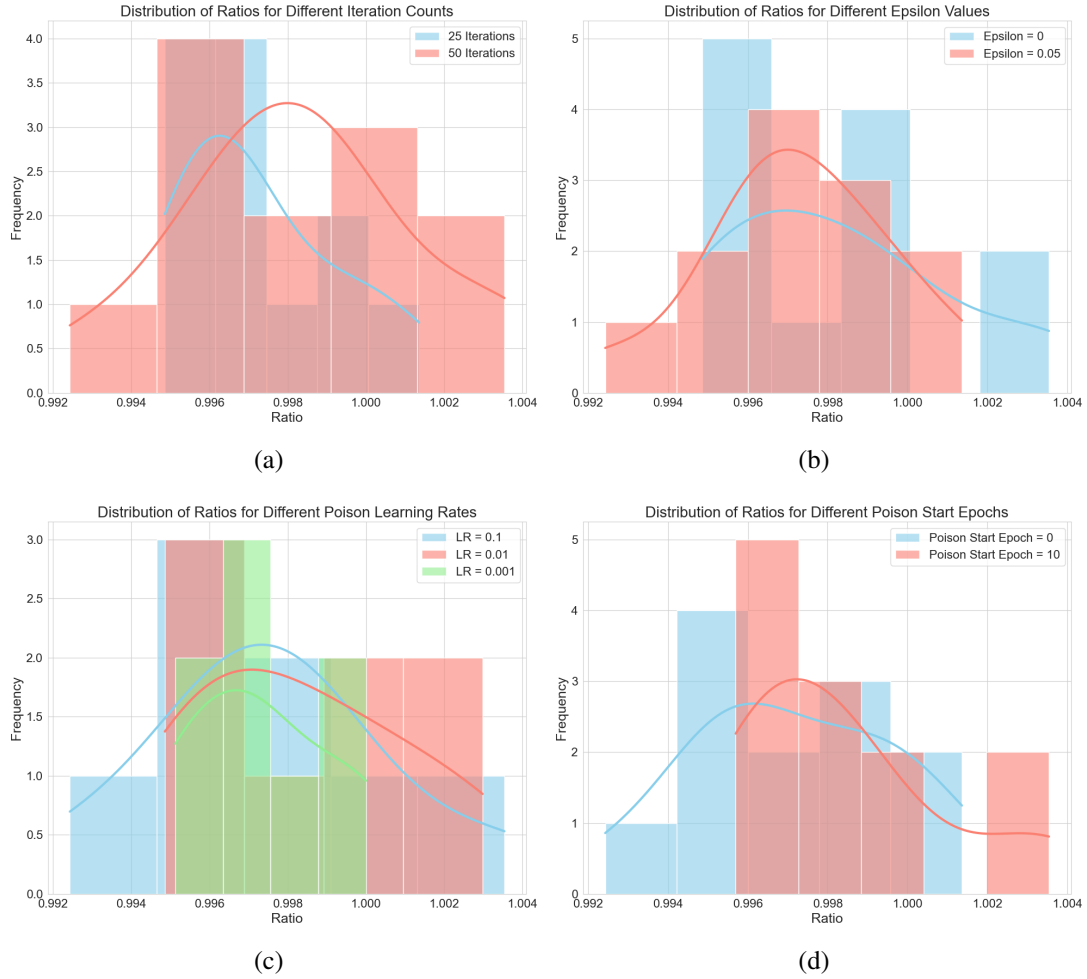


Figure 4.1: Hyperparameter run ratios of poison validation compared to SAM validation for (a) number of iterations, (b) epsilon value, (c) learning rate, and (d) poison start epoch. Higher ratio means poisoning is more effective.

Figure 4.1 is split up into 4 different histograms, each dedicated to one of the hyperparameters we were optimizing. The histograms show the ratios of the highest validation accuracy of a training run on poison data to the highest validation accuracy of a training run with SAM on clean data. The SAM run is the one used to create the poison data that is being trained on. We use ratios because the performance of the SAM run has a large impact on how effective the poison run is, thus comparing the ratios is more effective in choosing which poisoning parameters are better at matching SAM or exceeding it. The correlation between the performance of the SAM run and our poisoning method

highlights that it is able to encode information into the poison that comes from training with SAM.

We can see that there is a lot of variance in how effective parameters are, a large part of which is due to noise, seeing as we have only very few sample runs. Even so, on average we have 50 iterations outperforming 25, 0 epsilon outperforming 5% epsilon, 0.01 outperforming the other learning rates and starting at epoch 10 for poisoning actually is the most consistent improved hyperparameter outperforming poisoning from the beginning. The later poison start epoch supports our conjecture that poisoning after a few epochs gives stabler perturbations that are more focused on finding a minimum.

Dataset	**Model**	**LR^P**	ε	**Iterations**	**Start Epoch**
F-MNIST	ResNet18	0.01	0	50	10

Table 4.2: Optimal poisoning hyperparameters determined through validation testing for Fashion-MNIST with ResNet18. LR^P denotes the learning rate for poisoning, while Start Epoch indicates the epoch at which poisoning begins.

Finally, with the hyperparameters found, we perform 5 runs of training on clean data and 5 runs training on poisoned data and report average and standard errors. The results for SAM come from the poisoning runs. We can see a slight improvement in the model trained on poison data compared to the baseline, though following experiments will help us be more confident in the effectiveness of our method.

Dataset	**Model Configuration**	**Test Accuracy (%)**
F-MNIST	ResNet18	91.64 ± 0.45
	ResNet18 + Poison	$\mathbf{91.99 \pm 0.15}$
	ResNet18 + SAM	91.99 ± 0.16

Table 4.3: Performance comparison of standard training, poison-based training, and Sharpness-Aware Minimization (SAM) on Fashion-MNIST with ResNet18. Bold values indicate the best performance. Results demonstrate that both poisoning and SAM techniques improve model accuracy compared to standard training.

4.2.3 Discussion

Because of the limited amount of runs we could perform for our hyperparameter search, we are constrained in what we can deduce from it. The strongest improvements we could see were from 50 iterations compared to 25 and from starting poisoning after 10 epochs instead of 0. The increased number of iterations translates to us spending more compute on solving the optimization and thus increasing the quality of the poisons and how much SGD gradients align the the SAM gradient. The performance increase that comes from starting after 10 epochs instead of 0 might be because of very noisy gradients at the beginning of training, which are not aligned with SAM at all and which also do not get repeated when training on poison data. Not encoding information about

the noisy gradients at the start could make the poisons less noisy and more focused on encoding important information that comes in later stages of training.

4.3 Poisoning without hyperparameter search

In this section we run experiments to see how the data poisoning methods perform without dataset and model specific hyperparameters. A poisoning algorithm that does not rely on expensive finetuning beforehand is able to be widely used and save a lot of computational resources that would've been used to finetune.

Since we want to generalize beyond just one dataset and model for this, we choose a variety of models and datasets to run experiments on. In total we run experiments on four different combinations of datasets and models. We run experiments on DenseNet3, WideResNet-16 and WideResNet-18. The datasets we use are CIFAR-10 and CIFAR-100. Due to logging errors we unfortunately can only report accuracy and not loss, which is not ideal as we fundamentally want to change the loss landscape, but accuracy is a very good proxy for loss.

4.3.1 Setup

We keep the same poisoning hyperparameters as found by the hyperparameter search described previously. Poisoning Rate of 0.01, epsilon of 0, 50 iterations. Depending on the experiment we sometimes choose a later start epoch than 10, because some experiments train for a much longer time than the model in our hyperparameter search.

We run our method on CIFAR-10 and CIFAR-100 datasets and DenseNet-3, WideResNet-16 and WideResNet-28 models. We repeat each experiment 5 times and report mean test accuracy and standard errors.

For DenseNet-3 we train for 100 epochs with a batch size of 64 and a learning rate of 0.1 which gets divided by 10 at 50 epochs and divided by 10 again at 75 epochs. We use weight decay of 0.0001 and a Nesterov momentum of 0.9. For both WideResNets we train for 200 epochs with a weight decay of 0.0005, learning rate of 0.1, momentum of 0.9. We multiply the learning rate by 0.2 at 60, 120 and 160 epochs.

The hyperparameters are taken from the original papers presenting DenseNets and WideResNets [37, 14] and are running for fewer epochs than in the originals to save compute.

4.3.2 Results

As we can see from Table 4.4 there is no definite improvement when it comes to training with poison data across datasets and models. The largest improvement from training on poison data comes for CIFAR-10 + DenseNet-3, where test accuracy increases from 89.87% to 90.88%. For CIFAR-100 + WideResNet-16 the average test accuracy is slightly higher, but the standard error is extremely high compared to this improvement. On CIFAR-10 + WideResNet-16 we can see that training on poisoned data gives worse accuracy than training on clean data.

Dataset	Model Configuration	Test Accuracy (%)
CIFAR-10	DenseNet-3 (k=40)	89.87 ± 0.24
	DenseNet-3 (k=40) + Poison	$\mathbf{90.88 \pm 0.15}$
	DenseNet-3 (k=40) + SAM	89.56 ± 0.29
	WideResNet-16	$\mathbf{91.78 \pm 0.41}$
	WideResNet-16 + Poison	91.16 ± 0.24
	WideResNet-16 + SAM	90.01 ± 0.25
CIFAR-100	WideResNet-16	70.08 ± 0.22
	WideResNet-16 + Poison	$\mathbf{70.13 \pm 0.42}$
	WideResNet-16 + SAM	67.27 ± 0.46
	WideResNet-28	70.96 ± 0.78
	WideResNet-28 + Poison	$\mathbf{72.31 \pm 0.41}$
	WideResNet-28 + SAM	69.02 ± 0.37

Table 4.4: Performance comparison of different model configurations on CIFAR-10 and CIFAR-100 datasets. Bold values indicate the best performance within each model group. Poison and SAM denote poisoning and Sharpness-Aware Minimization optimization strategies, respectively.

4.3.3 Discussion

From our results, we see that in three out of four cases, training with the poison data improved test accuracy, when comparing to training on clean data, as well as when training with SAM. For CIFAR-100 with WideResNet-16, this improvement is very small compared to the standard deviation of the mean test accuracy, thus we can not rely on this to be an improvement. Thus, we are left with two out of four experiments improving by a meaningful amount when comparing with SGD on clean data. This does show that sharpness aware data poisoning can be promising and lead to results that beat training with normal SGD.

It is also interesting that the SAM optimizer actually performs worse in a lot of cases. We suspect that this is because SAM takes longer to find a minimum as it doesn't always follow the direction of steepest descent. In the original paper by Foret et al. [9], they train for more epochs than we do to reach a minimum, but because of compute restraints we could not train for that long. Thus, training for more epochs will likely give us results where SAM performs better than normal SGD. It may also be possible that letting SAM train for more epochs, and thus poisoning at a time in training where SAM outperforms SGD, makes poisoning even stronger. As we saw in the hyperparameter search, poisoning more towards the beginning of training does not provide as much benefit as poisoning towards the end. Combining this fact, with the facts that SAM needs to train for longer, there is space for improvement in how poisoning might be applied.

4.4 Limitations

In this section we discuss the limitations of both the hyperparameter search and of the further experiment involving no hyperparameter tuning. Due to the compute intensive nature of our experiments, there are a variety of limitations that our results need to be put in context with.

4.4.1 Hyperparameter Search Limitations

In the hyperparameter search we use a ResNet-18 with the F-MNIST dataset. We allocated extra compute to the non-poison setup compared to the base parameters we used for the poison search. Nevertheless, for our base setup we only finetuned learning rate and momentum. Additional hyperparameters such as batch size or weight decay could've also been tuned to extract even better performance our of the model, but weren't considered to be able to focus more on the search for good poison parameters.

We also only performed our search on one model and one dataset. This is an obvious limitation, as this is not a very representative of machine learning as a whole. Additionally, we only were able to perform one run for each hyperparameter combination, which causes results to not have as much statistical evidence. It also made it hard to properly identify the best combination of hyperparameters, as there was a lot of noise in the results. Additionally, the differences between poison performance and SAM performance were very small, which was to be expected, and a poison run's performance seemed to be dependent on the SAM performance to some degree.

4.4.2 Fixed Hyperparameters Experiments Limitations

For our experiments that did not involve a hyperparameter search, we used two different datasets and 3 different models. This is a significant improvement over the previous experiment using just one of each, but is still limited. Firstly, CIFAR-10 and CIFAR-100 have the same images, just with different labeling applied. In addition to that, we focus solely on image classification. Performing experiments in other domains, such as text would strengthen our results and let us make conclusions about how the algorithm would perform in other settings. We also only focus solely on convolutional neural networks. Other popular architectures such as transformers [34] are widely used and it would be interesting to see how our algorithm performs on a different type of architecture.

Chapter 5

An In Depth Analysis on how Sharpness Aware Data Poisoning works

In this chapter we further investigate our data poisoning method to gain insight into how it might be improved and whether it really does help SGD find flatter minima in some cases. We perform experiments on MNIST [7], training with a LeNet-5 [23], as well as CIFAR-10 and DenseNet-40.

5.1 Training Setup

For MNIST, we split the held out set into 4000 validation images and 6000 test images. The model we use is a LeNet-5, a simple convolutional network shown to be able to achieve around 99% test accuracy on MNIST. We follow a similar training setup as LeCun et al. [23]. We use cross entropy loss, SGD with a learning rate of 0.001, weight decay of 0.0001, momentum of 0.9, batch size of 64 and training for 30 epochs.

5.2 Poisoning Setup

From a few preliminary testing runs, we chose a starting poisoning at epoch 1, iterating for 5 iterations per sample, with a poison rate of 0.01 and no epsilon. We also train on the poison data from the first epoch. For some experiments we may vary from these default parameters and will mention the experiment specific ones.

5.3 Evaluation of LeNet-5 on MNIST

For our experiments we logged metrics of our poisoning algorithm in order to better understand its behavior. We do this in order to identify potential shortcomings and motivate further research that can be done in this area.

We begin by plotting validation and training losses of clean vs poisoned data. We follow this up by investigating how the SAM gradients compare to the poison gradients. Finally we look at how the poison gradients change, as the data gets poisoned over time and the drift between optimization objectives at different epochs. We also perform a sharpness investigation of clean vs poisoned data by plotting eigenvalues and using one-dimensional linear interpolation plots.

5.3.1 Losses when poisoning on MNIST

Firstly we can see the progression of losses when trained on clean MNIST vs the poisoned MNIST in Figure 5.1. We can see that the gap between training loss and validation loss is much higher for vanilla than poisoned. Validation loss on vanilla MNIST also drops much faster than on the poisoned data, but then flattens out earlier. Training on the poisoned data is able to improve validation loss slightly after around 25 epochs, though the difference is negligible.

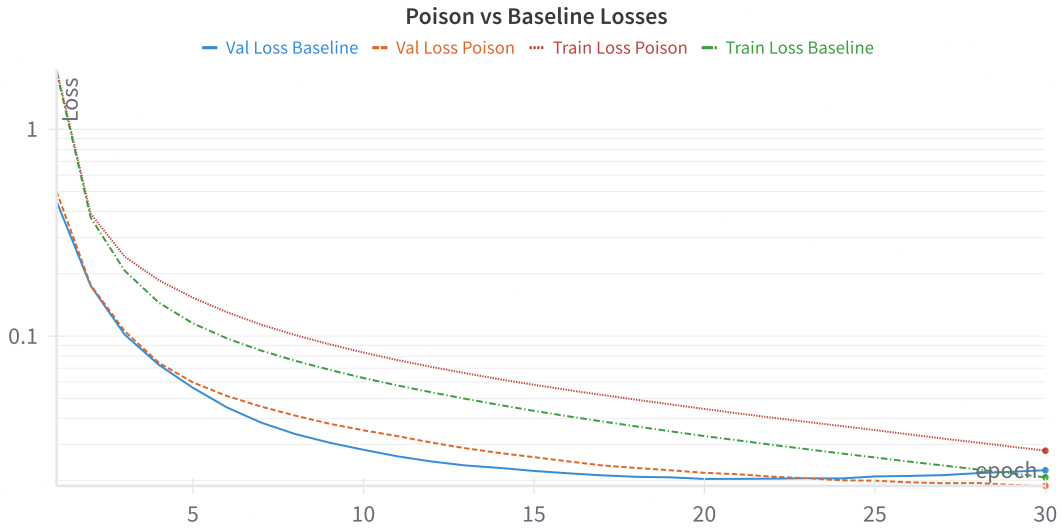


Figure 5.1: Train and Validation Losses for Poison and Baseline for LeNet-5 on MNIST

The slower decrease in loss on poison can be attributed to it being more difficult to learn from the noise we added. Nevertheless, this added noise had the effect of letting validation loss continually decrease, compared to clean data, which overfits. The training loss on both clean and poison data is higher than on the validation data. This could indicate that the validation set is harder than the train set. Normally we would expect a lower training loss than validation loss. We only notice this for the non-poisoned data toward the end of training.

5.3.2 Does random noise help with training?

To inspect how the magnitude of the perturbations changes during training, we visualized it in Figure 5.2. We initialize the poison perturbations to normally distributed random

noise at the beginning with a magnitude of 300. We can then see that magnitude steadily increases at each iteration, as the optimization algorithm tries to increase the cosine similarity between SAM and SGD gradients. The final magnitude after poisoning is at a little over 1000. (Note that we use the Frobenius norm [35], which is the default matrix norm of PyTorch)

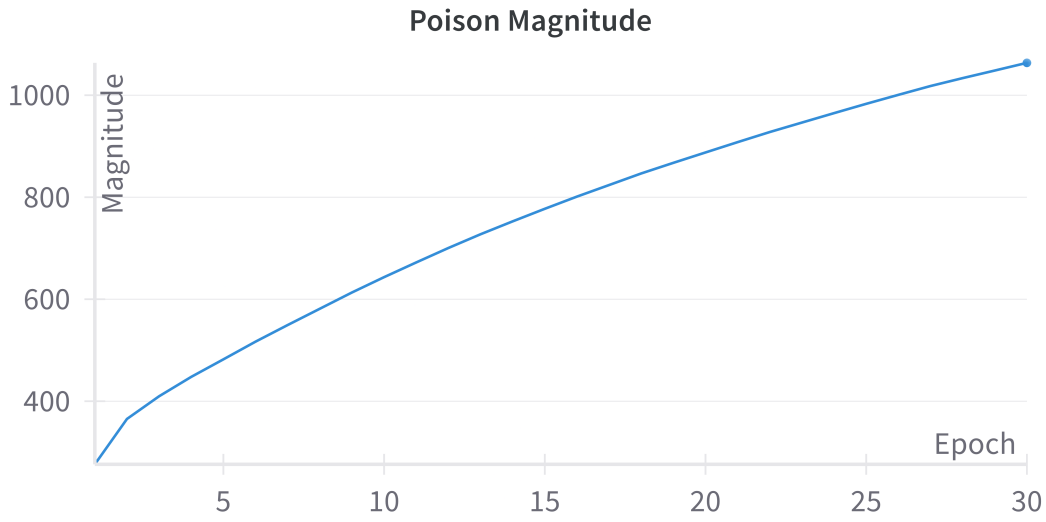


Figure 5.2: Magnitude of the poison deltas over epochs. Magnitude steadily increases as training progresses.

From this we can construct noise that we add to the clean data. It is a common practice to use noisy gradients [27] or add noise to the data directly [4] in order to improve generalization. Thus, we can make a comparison between training on our poison perturbations and the randomly distributed noise, in order to see whether the benefits of poison might just come because it is noise that causes robustness, or there is a structural benefit to the perturbations we generate.

In Figure 5.3 we make a comparison between the losses when training on random noise and when training with poisons. We can see that there is a noticeable benefit that comes from training with poison compared to random noise. Validation loss when training with poison is consistently below validation loss when training on random noise. In addition to that, the training loss decreases faster on random noise, showing that it is much easier to fit the random noise than the structure found in the poison data. We can also see that after around 24 epochs, validation loss starts to increase again, pointing to overfitting, even with random loss embedded into the data. The validation loss of training on poison on the other hand decreases steadily, showing that there are clear benefits against overfitting.

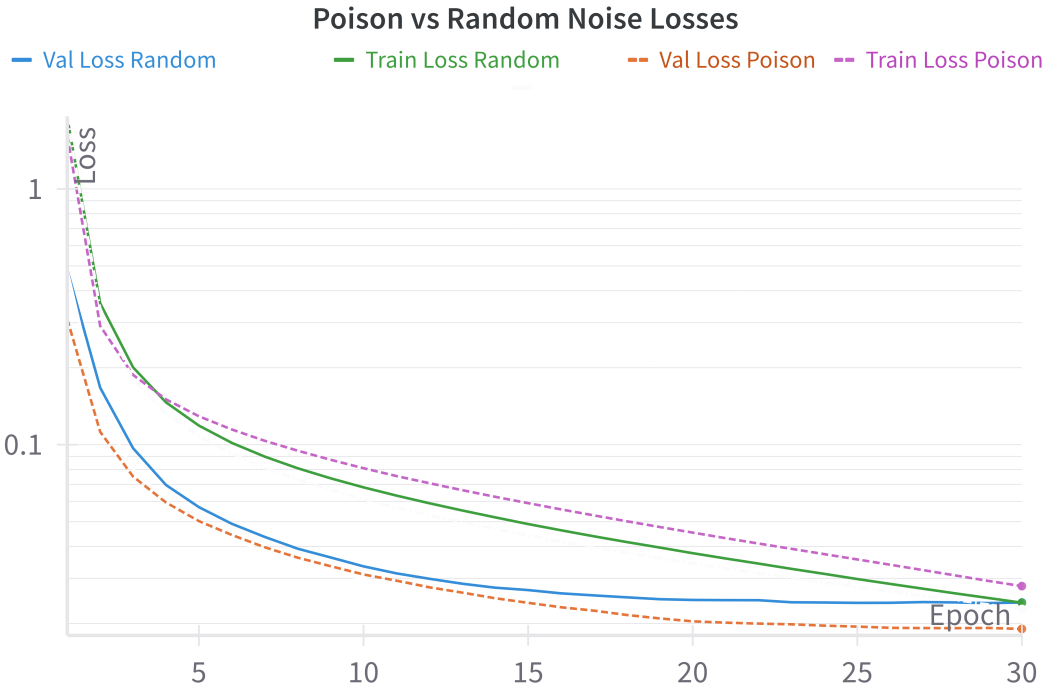


Figure 5.3: Train and Validation Losses for training with normally distributed noise and training with poison. Magnitude of the noise is equal to 1063, which is the same the poison magnitude after 30 epochs.

5.3.3 Poison vs SAM on MNIST

Previously we have compared training with poison and training without, with normal SGD. From this we have seen that there are considerable benefits that come from training with poison. Now we compare how training with SAM differs from training with poison using normal SGD.

In Figure 5.4 it looks like SAM performs a lot worse than SGD on Poison. This is mainly due to SAM converging much slower than SGD with poison does. Validation loss when training on poison data stays consistently under validation loss when training with SAM. Similarly, training loss on poison data also stays consistently under the training loss of SAM. The gap between validation loss and training loss is much higher for SAM than for SGD on poison. This shows that SAM might be better at preventing overfitting, though training for more epochs would give more insight. Nevertheless, the faster convergence of SGD on poison data may make training on the poison data even more viable than pure SAM, depending on the training setting.

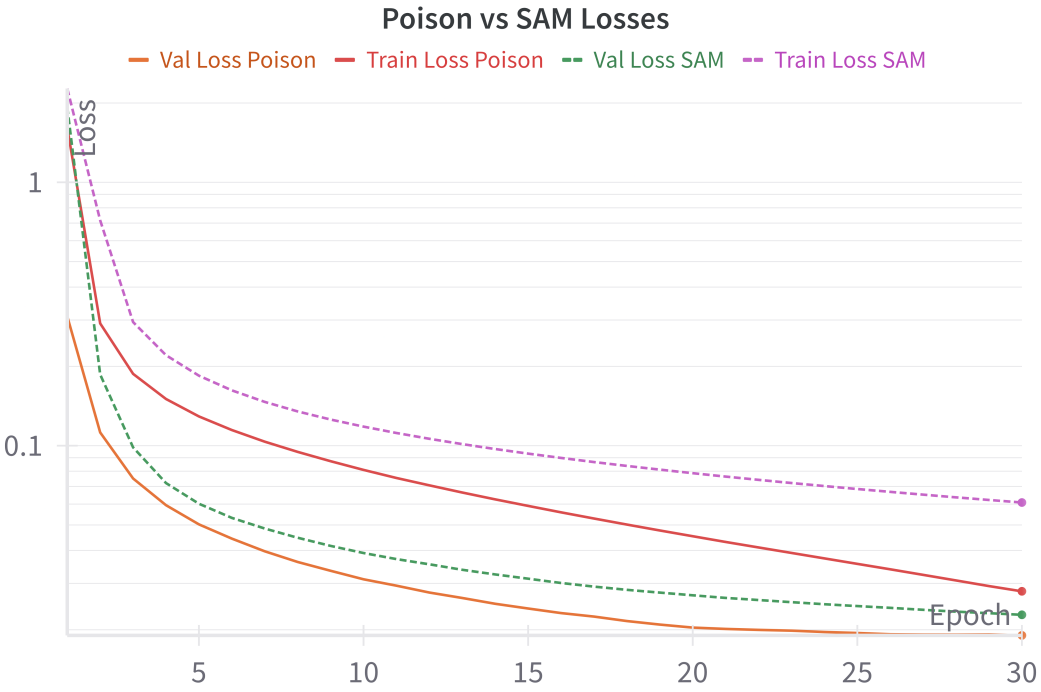


Figure 5.4: Train and Validation Losses for training poison using SGD and training on clean data with SAM.

5.3.4 Cosine Similarity of Gradients

To understand SAM and the poisoning process further, we can investigate how cosine similarity between gradients behaves over time. Since we optimize for cosine similarity, it is important to understand how this objective behaves and how easily we can optimize for it.

5.3.4.1 Cosine Similarity during poisoning

In Figure 5.5 we see the cosine similarity between the SGD gradient calculated on the poison and the SAM gradient. We can see that the cosine similarity loss is very low throughout the duration of training. At the beginning of training it is the largest, which might have to do with the network weights not being along a proper training trajectory yet and a lot of noise in the gradients. As training progresses, cosine similarity loss reduces and stays at around 0.05. This shows that as training progresses, the SAM and SGD on poison objective, while not being the same completely, are similar. It is to be noted that the cosine similarity loss in Figure 5.5 shows the gradients on the poison data, and thus as the poison evolves, gradients change as well, in addition to parameters.

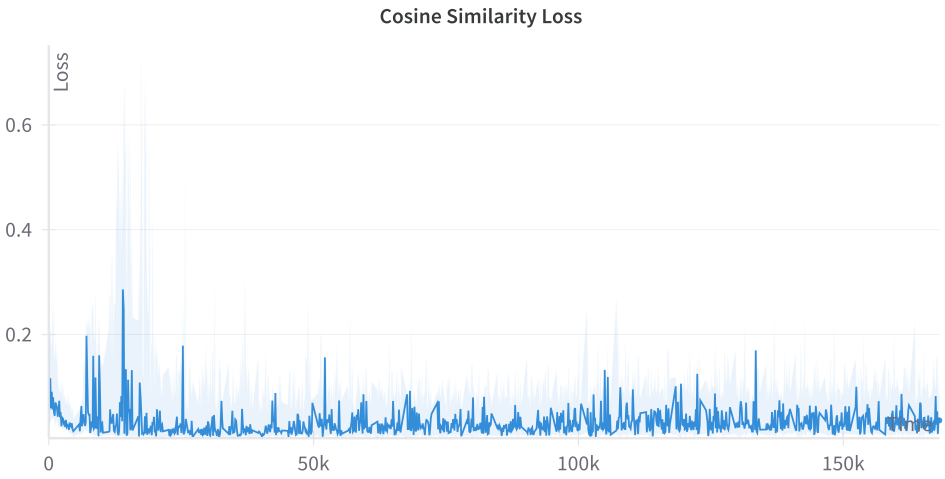


Figure 5.5: Cosine Similarity Losses between poison gradient and SAM gradient during poison time over all epochs for LeNet-5 on MNIST.

The previous figure shows the losses over the whole training run, when we zoom in, we can see how cosine similarity works on a batch by batch basis, pictured in Figure 5.6. Here we can see that there are spikes at regular intervals, which decrease until a next spike appears. Each of the spikes corresponds to a new batch, where the loss decreases with the number of poison iterations performed on the batch. This is why the spikes are spaced apart evenly. As we can see, the poisoning is very effective in minimizing the cosine similarity loss. There also is a lot of variance between different gradients. Just in the sample Figure 5.6 there are some gradients that start at a loss of over 0.6, while others start out at a very low values less than 0.05. This shows that the cosine similarity between gradients depends strongly on the data batch, not only on the parameters, as a small change in parameters would not cause such an abrupt change in similarity.

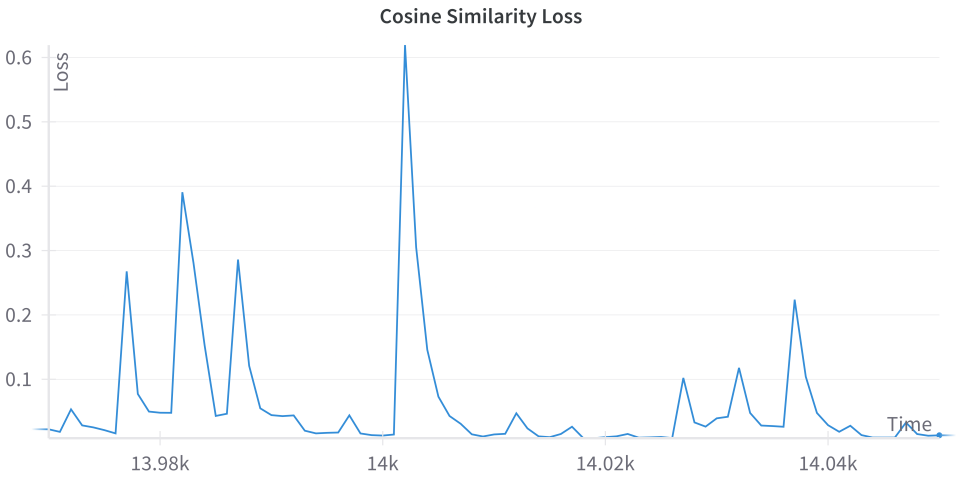


Figure 5.6: Zoomed in view on Cosine Similarity Losses between poison gradient and SAM gradient during poison time over all epochs for LeNet-5 on MNIST.

5.3.4.2 How does the cosine similarity objective change during training

The question is, whether this reduction in cosine similarity loss holds over time, or whether subsequent epochs destroy the similarity between poisons, and thus render our poisoning ineffective. To investigate this, we keep the parameters for the network trained at earlier epochs, and graph the average cosine similarity loss for it as the main networks trains, and poisons keep getting perturbed. As we can see in Figure 5.7, the loss for poisons does increase as the poison gets more perturbed. We can see that the perturbation is so large in facts, the similarity strays further apart than the baseline similarity which stays quite low. Thus, this naive poisoning process has a negative effect on cosine similarity as it goes on. A way to mitigate this would be to impart multiple weights from different training stages into the loss objective. That way, we could punish perturbation that increase distance between SAM and SGD gradients for all stages of training.

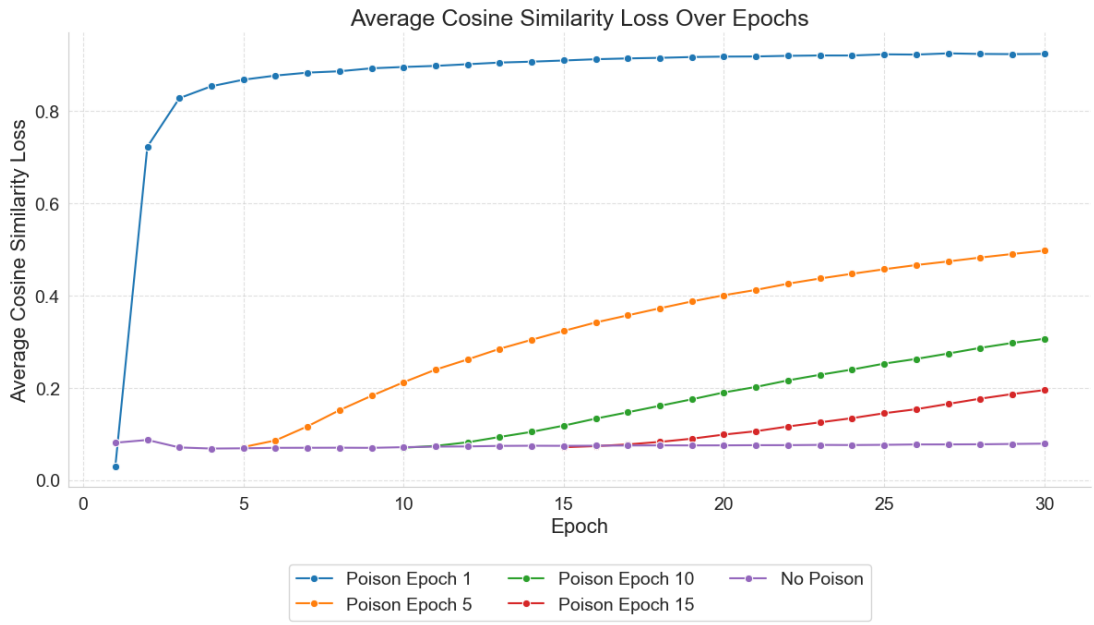


Figure 5.7: Average cosine similarity of SAM gradient and standard gradient on poison data over epochs. Shown for weights after training on 1 epoch, 5 epochs, 10 epochs, 15 epochs and for all epochs with no poisoning. We can see that as poisoning progresses, cosine similarity of the poisons decreases. It decreases even more than the baseline cosine similarity between SAM gradient and SGD gradient when no poisoning occurs. This shows that later poisoning counteracts any cosine similarity effects from previous epochs.

5.3.4.3 Cosine similarity on clean data with SAM

Looking at just the cosine similarity between SAM and SGD gradients on non-poisoned data, we can see that it isn't in fact constant, but has a sharp spike at the beginning before falling suddenly and then steadily increasing, as we can see in Figure 5.8. The loss is very small, with the average loss never exceeding 0.1, showing the when training with SAM, the SAM gradient does not differ too much from the SGD gradient. The

slow increase in loss signifies that when the optimizer approaches the minimum, SGD is much more likely to go in a direction of sharp loss, whereas SAM continues trying to find a flatter minimum. The spike in cosine similarity loss at the beginning of training can be attributed to the noisy irregular loss landscape that happens when network weights are close to their initially randomly initialized weights.

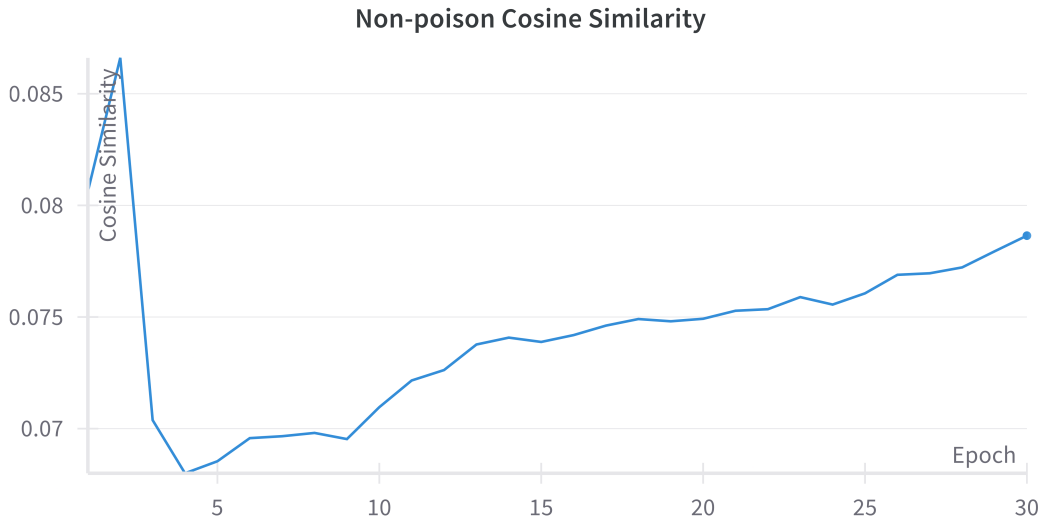
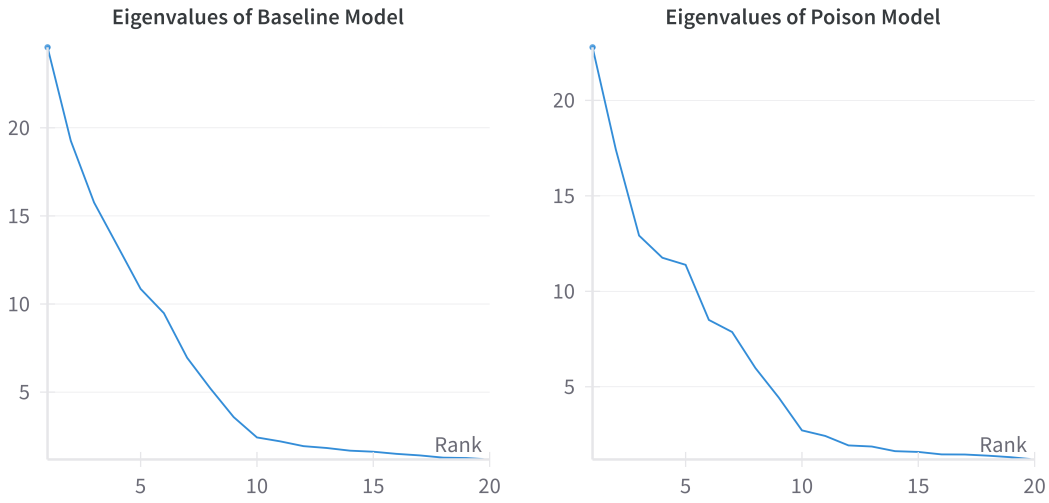


Figure 5.8: Cosine Similarity Loss between SAM gradients and normal SGD gradients when training with SAM over 30 epochs.

5.3.5 How does sharpness of poisoned and unpoisoned minima differ?

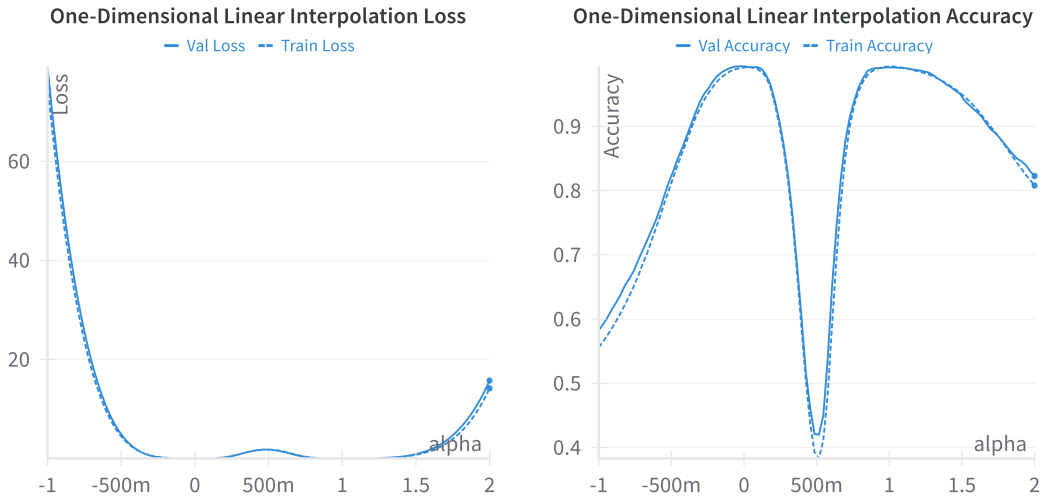
From an analysis of the eigenvalues of the Hessians of the minima found with vanilla and poison MNIST data, we can see that there wasn't a significant effect on the sharpness of the networks. Looking at the eigenvalues of the two networks, they are approximately similarly distributed as we can see in Figure 5.9 (As mentioned in previous chapters, large eigenvalues indicate a large curvature/sharpness). This means that training on the poison data, even if it increases performance, does not considerably change the sharpness of the final minimum.



(a) (b)

Figure 5.9: Eigenvalues for baseline and poison models.

We can also look at one-dimensional linear interpolation graphs to analyze the sharpness of the minima found when training on the poison vs baseline data. We can see in Figure 5.10 that loss is a little bit more sensitive around the poison minimum compared to the baseline minimum. Similarly, accuracy decreases around the poison minimum a bit sharper than around the baseline minimum. Overall, the minima do seem to behave quite similarly and there does not seem to be a significant change in the sharpness of the minimum we find on poison data compared to clean.



(a) (b)

Figure 5.10: One-dimensional linear interpolations of loss and accuracy. Poison minimum is at 0.0 and baseline minimum is at 1.0.

Chapter 6

Further possible expansions of the data poisoning algorithm

In this chapter we present ways that the algorithm could be extended to 1) ensembles of models, making it potentially more robust and effective and 2) data augmentations, such as crops or rotations, all common within computer vision and often used as a defense against data poisoning.

6.1 Ensembles of models

In data poisoning, it is common to poison with an ensemble of models as a target. These ensembles could be made up of different parameters applied to the same model architecture, found through different training runs for example. In this case, the idea is that poisoning against multiple sets of parameters makes attacks more robust and keeps poisons from ”overfitting” against one set of weights. Ensembles could also be different architectures, in which case the target is to have more effective poisoning even if the model architecture is unknown. For our algorithm both of these settings could be useful and potentially improve performance because poisons will be more robust to a different loss landscape.

The following pseudocode shows how to extend the current algorithm to work against an ensemble of models. The different parameters are denoted θ^j and there are different m parameters.

Input: S, $\mathcal{L}$
Initialize θ_0 randomly, $t = 0$
Initialize Δ to random noise
while not converged **do**
  for j from 1 to m **do**
    Sample batch B from S
    Compute gradient $g^j = \nabla_\theta \mathcal{L}_B(\theta_i)$
    Compute $\hat{\varepsilon}(\theta_i^j)$ per Equation 3.2

Compute $g'^j = \nabla_{\theta^j} \mathcal{L}_B(\theta_i^j)|_{\theta_i^j + \varepsilon(\hat{\theta}_i^j)}$
end for
Compute $\Delta^B = \text{argmin}_{\Delta^B} \mathcal{B}(\Delta^B, \theta_i^1, .., \theta_i^m)$
for j from 1 to m **do**
Set $\theta_{t+1}^j = \theta_t^j - \eta g'^j$
end for
Set $t = t + 1$
end while

The following is the adjusted minimization of the negative cosine similarity for multiple models:

Input: g', $\mathcal{L}$, $\theta^1, .., \theta^m$, B, n, Δ^B, α, ε
Output: Δ^B
$i = 0$
while not converged and $i < n$ **do**
total_loss $= 0$
for j from 1 to m **do**
Compute gradient $g = \nabla_{\theta^j} \mathcal{L}_{B+\Delta^B}(\theta^j)$
Compute $\mathcal{B}(\Delta^B, \theta^j)$ between g and g' according to Equation 3.6
total_loss = total_loss +
end for
Compute $\nabla_{\Delta^B} \mathcal{B}(\Delta^B, \theta)$ and minimize $\mathcal{B}(\Delta^B, \theta)$ with one step of SGD
Project Δ^B onto C
$i = i + 1$
end while

6.2 Data augmentations

In computer vision tasks, data augmentations are often used to extend datasets and improve network performance and robustness. Data augmentations are often used in data poisoning, as a defense, since poisons often can be made harmless by applying a simple rotation or flip. A common strategy to make data poisoning effective even in a training regime that utilizes data augmentations, is to pretend like the data augmentation is another layer of the neural network, backpropagating through the augmentation to create the poison. A requirement for this is that the augmentation is differentiable, which many common augmentation are such as flips, rotations or crops, and are provided by libraries [31].

We outline an algorithm to apply poisoning on augmented data below, which we already support in our codebase. (We denote augmented data with a superscript A and thus B^A is the augmented batch and $(B+\Delta)^A$ is the augmentation applied to the sum of the batch and delta):

Input: S, $\mathcal{L}$
Initialize $\theta_0 = 0$, $t = 0$

Initialize Δ to random noise
while not converged **do**
  Sample batch B from S
  Apply augmentation to B to produce B^A
  Compute gradient $g = \nabla_\theta \mathcal{L}_{B^A}(\theta_i)$
  Compute $\hat{\varepsilon}(\theta_i)$ per Equation 3.2
  Compute $g' = \nabla_\theta \mathcal{L}_{B^A}(\theta_i)|_{\theta_i + \varepsilon(\hat{\theta}_i)}$
  Compute $\Delta^B = \text{argmin}_{\Delta^B} \mathcal{B}(\Delta^B, \theta_i)$
  Set $\theta_{t+1} = \theta_t - \eta g'$
  Set $t = t + 19765$
end while

The following is the adapted optimization algorithm. Note that the augmentation needs to be differentiable, because we minimize the negative cosine similarity with respect to the perturbations and thus need to backpropagate through the augmentations to compute the gradient.

Input: g', $\mathcal{L}$, θ, B, n, Δ^B, α, ε
Output: Δ^B
$i = 0$
while not converged and $i < n$ **do**
  Compute gradient $g = \nabla_\theta \mathcal{L}_{(B+\Delta^B)^A}(\theta)$
  Compute $\mathcal{B}(\Delta^B, \theta)$ between g and g' according to Equation 3.6
  Compute $\nabla_{\Delta^B} \mathcal{B}(\Delta^B, \theta)$ and minimize $\mathcal{B}(\Delta^B, \theta)$ with one step of SGD
  Project Δ^B onto C
  $i = i + 1$
end while

For poisoning augmented data we set $\mathcal{B}(\cdot, \theta)$ to the following:

$$\mathcal{B}(\Delta, \theta) = 1 - \frac{\langle \nabla_\theta \mathcal{L}_{B^A}(\theta)|_{\theta+\hat{\varepsilon}(\theta)}, \nabla_\theta \mathcal{L}_{(B+\Delta)^A}(\theta) \rangle}{||\nabla_\theta \mathcal{L}_{B^A}(\theta)|_{\theta+\hat{\varepsilon}(\theta)}|| \cdot ||\nabla_\theta \mathcal{L}_{(B+\Delta)^A}(\theta)||} \tag{6.1}$$

6.3 A generalization of encoding information in the data

We do not need to limit ourselves to sharpness or gradient alignment when performing optimization of the data. In this section we introduce a more general problem statement, aimed at motivating future work on optimizing data and encoding data in it that can assist with machine learning.

6.3.1 A general statement

The most general statement for optimizing data to learn on is the following. Say we have a hypothesis class $\mathcal{H}$, a dataset $S \subset X \times Y$ and a learning algorithm $A : X \times Y \to \mathcal{H}$. We then have an objective function o for which we want the learning algorithm to minimize this objective function when training on data. Thus, we need a poisoning algorithm P,

which takes a dataset S, learning algorithm A and objective function o as parameters, and return a poisoned dataset S', such that $o(A(S')) \leq o(A(S))$.

This problem statement can be applied to the data poisoning performed in this dissertation. We have a learning algorithm A to train the model. We have an objective function o, the sharpness of the trained model. We have a clean dataset S on which we train. And we finally get a poison dataset S' that is sharpness aware.

6.3.2 Optimizing for an auxiliary objective

Of course, in this dissertation a big hurdle is that it is often hard to optimize directly for an objective, and we need to optimize for an auxiliary objective instead. Thus we can add another auxiliary function o' to the problem statement. Now we want the poisoning algorithm P to poison a dataset S by minimizing an auxiliary objective o' which should lead to a reduction in our main objective o. In this dissertation we used the negative cosine similarity between SAM and SGD gradients as our auxiliary objective and the final minimum's sharpness as our main objective.

6.3.3 Possible Objectives for Data Poisoning

While our primary focus has been on sharpness, the framework we developed can be extended to optimize for various objectives beneficial to neural network training and performance.

Loss Loss minimization represents perhaps the most fundamental objective. By strategically modifying training data, we can reshape the loss landscape to help with discovery of higher-quality minima. This approach requires careful implementation to ensure optimization produces lower loss on the original, clean data, not just on the poisoned dataset.

Privacy Preserving data privacy while maintaining model utility presents another compelling application. Differential privacy [17], typically enforced through algorithms like Differentially Private SGD [1], offers formal guarantees against data extraction. Developing poisoning strategies that enhance privacy protections could prove valuable, though establishing rigorous theoretical guarantees would be essential for practical adoption.

Regularization Optimizing data to implicitly impose regularization effects represents another promising direction. For example, data modifications that discourage rapid weight growth could potentially replace or complement explicit regularization terms like L2 penalties, potentially offering more nuanced control over model behavior.

Bias Mitigation Our approach enables formal characterization and targeted reduction of unwanted biases. This can be accomplished either through selective data exclusion or through strategic modifications that transform biased distributions into more balanced ones, thereby addressing a critical challenge in machine learning ethics.

Pseudo-Labeling Existing methods like pseudo-labeling [18] already demonstrate how data modification improves training outcomes. In this technique, a model trained on partially labeled data assigns labels to unlabeled examples, effectively expanding the training set. This process naturally reshapes the loss landscape, smoothing it and eliminating sharp minima that might otherwise trap models trained on limited data. Our poisoning framework shares conceptual similarities with this approach but offers more explicit control over the resulting landscape properties.

Multi-objective Poisoning Many practical applications will require balancing multiple objectives simultaneously. The ensemble poisoning method discussed earlier exemplifies this approach by combining cosine similarity objectives across multiple models. Similar principles could address other multi-objective scenarios, such as simultaneously optimizing for sharpness while constraining perturbation magnitude to prevent divergence between clean and poisoned data objectives. This represents a particularly rich avenue for future research.

While this dissertation has focused primarily on sharpness as an objective, the methodology we've developed opens numerous possibilities for future investigation. The range of potential objectives and their combinations suggests that data poisoning, when approached constructively, offers a powerful new tool to help train neural networks and optimize them for various objectives.

Chapter 7

Conclusion

In this dissertation we proposed a new method for optimizing data with data poisoning, such that this poisoned data uses gradients that match the gradients used by SAM. We introduced theory to ground this method and explained concepts such as sharpness, SAM and gradient alignment. Using this theory we proposed a formal algorithm to perform poisoning with. This algorithm we evaluated empirically in a few ways. We performed a hyperparameter search in order to find parameters that work well for our poisoning method. These hyperparameters can be used as a starting point for others who want to expand on this research in the future. Following this hyperparameter search, we performed experiments on various datasets and models, to evaluate performance of the algorithm without dataset and model specific hyperparameters. We found improvements when training on various datasets and models, even without dataset and model specific hyperparameters. In some cases, these improvements not only beat training on clean data with SGD, but also training on clean data with SAM. This can be attributed to the combination of poison data and SGD benefiting from sharpness aware gradients and the faster convergence of SGD compared to SAM.

To further explain our results, we performed experiments on MNIST with LeNet-5, measuring various metrics related to poisoning. We documented how alignment between gradients changes over time. We found that a major weakness of the current poison algorithm is the undoing of poisoning by later epochs. Furthermore, in some cases poison data actually reduces the cosine similarity between the SAM gradient and the SGD gradient, compared to when just training on non-poisoned data. We do not know whether this is specific to the small model and dataset we used and further research will be needed. We did propose a few potential solutions to mitigate this problem, which can be explored in further research. Additionally, we created a more generalized problem statement of using data poisoning to optimize for some objective. We hope that this problem statement can be used to try additional methods to poison a dataset and additional loss functions, not limited to just sharpness, but other properties of neural networks as well.

To the best of our knowledge, the algorithm we proposed is the first one to modify the loss landscape of neural networks directly, by applying data specific perturbations to the whole training set. Classic data augmentation differs, since those augmentation are

not data or loss landscape specific. In contrast to pseudo-labeling [18] we modify the loss landscape by modifying the inputs to the network, instead of the data labels.

In conclusion, we investigated and explained properties of the poisoning algorithm we created. We hope that this dissertation, inspires others to think about data in a non-fixed way and that optimization techniques should not just be limited to network weights. There are many avenues to explore in this region, with huge potential for impact and much left unexplored.

Bibliography

[1] Martin Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS'16. ACM, October 2016. doi: 10.1145/2976749.2978318. URL http://dx.doi.org/10.1145/2976749.2978318.

[2] Hojjat Aghakhani, Dongyu Meng, Yu-Xiang Wang, Christopher Kruegel, and Giovanni Vigna. Bullseye polytope: A scalable clean-label poisoning attack with improved transferability. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 159–178, 2021. doi: 10.1109/EuroSP51992.2021.00021.

[3] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey, 2018. URL https://arxiv.org/abs/1502.05767.

[4] Chris M. Bishop. Training with noise is equivalent to tikhonov regularization. *Neural Computation*, 7(1):108–116, 01 1995. ISSN 0899-7667. doi: 10.1162/neco.1995.7.1.108. URL https://doi.org/10.1162/neco.1995.7.1.108.

[5] Pratik Chaudhari, Anna Choromanska, Stefano Soatto, Yann LeCun, Carlo Baldassi, Christian Borgs, Jennifer Chayes, Levent Sagun, and Riccardo Zecchina. Entropy-SGD: Biasing Gradient Descent Into Wide Valleys, November 2016. URL https://arxiv.org/abs/1611.01838v5.

[6] Branton DeMoss, Silvia Sapora, Jakob Foerster, Nick Hawes, and Ingmar Posner. The Complexity Dynamics of Grokking, December 2024. URL http://arxiv.org/abs/2412.09810. arXiv:2412.09810 [cs].

[7] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012. doi: 10.1109/MSP.2012.2211477.

[8] Laurent Dinh, Razvan Pascanu, Samy Bengio, and Yoshua Bengio. Sharp Minima Can Generalize For Deep Nets. In *Proceedings of the 34th International Conference on Machine Learning*, pages 1019–1028. PMLR, July 2017. URL https://proceedings.mlr.press/v70/dinh17b.html. ISSN: 2640-3498.

[9] Pierre Foret, Ariel Kleiner, Hossein Mobahi, and Behnam Neyshabur. Sharpness-

Aware Minimization for Efficiently Improving Generalization, April 2021. URL http://arxiv.org/abs/2010.01412. arXiv:2010.01412.

[10] Jonas Geiping, Liam Fowl, W. Ronny Huang, Wojciech Czaja, Gavin Taylor, Michael Moeller, and Tom Goldstein. Witches' Brew: Industrial Scale Data Poisoning via Gradient Matching, May 2021. URL http://arxiv.org/abs/2009.02276. arXiv:2009.02276.

[11] Noah Golmant, Zhewei Yao, Amir Gholami, Michael Mahoney, and Joseph Gonzalez. pytorch-hessian-eigenthings: efficient pytorch hessian eigendecomposition, October 2018. URL https://github.com/noahgolmant/pytorch-hessian-eigenthings.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015. URL https://arxiv.org/abs/1512.03385.

[13] Sepp Hochreiter and Jürgen Schmidhuber. Flat Minima. *Neural Computation*, 9 (1):1–42, January 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.1.1. URL https://doi.org/10.1162/neco.1997.9.1.1.

[14] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks, 2018. URL https://arxiv.org/abs/1608.06993.

[15] W. Ronny Huang, Zeyad Emam, Micah Goldblum, Liam Fowl, J. K. Terry, Furong Huang, and Tom Goldstein. Understanding generalization through visualizations, 2020. URL https://arxiv.org/abs/1906.03291.

[16] Stanislaw Jastrzebski, Maciej Szymczak, Stanislav Fort, Devansh Arpit, Jacek Tabor, Kyunghyun Cho, and Krzysztof Geras. The break-even point on optimization trajectories of deep neural networks, 2020. URL https://arxiv.org/abs/2002.09572.

[17] Zhanglong Ji, Zachary C. Lipton, and Charles Elkan. Differential privacy and machine learning: a survey and review, 2014. URL https://arxiv.org/abs/1412.7584.

[18] Patrick Kage, Jay C. Rothenberger, Pavlos Andreadis, and Dimitrios I. Diochnos. A review of pseudo-labeling for computer vision, 2024. URL https://arxiv.org/abs/2408.07221.

[19] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima, February 2017. URL http://arxiv.org/abs/1609.04836. arXiv:1609.04836.

[20] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. URL https://arxiv.org/abs/1412.6980.

[21] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images.

[22] Cornelius Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators, 1950.

[23] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, volume 86, pages 2278–2324, 1998. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.7665.

[24] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the Loss Landscape of Neural Nets, November 2018. URL http://arxiv.org/abs/1712.09913. arXiv:1712.09913.

[25] Alexandra Sasha Luccioni and Alex Hernandez-Garcia. Counting carbon: A survey of factors influencing the emissions of machine learning, 2023. URL https://arxiv.org/abs/2302.08476.

[26] Kevin P. Murphy. *Probabilistic Machine Learning: An introduction*. MIT Press, 2022. URL probml.ai.

[27] Arvind Neelakantan, Luke Vilnis, Quoc V. Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, and James Martens. Adding gradient noise improves learning for very deep networks, 2015. URL https://arxiv.org/abs/1511.06807.

[28] C. C. Paige. Accuracy and effectiveness of the Lanczos algorithm for the symmetric eigenproblem. *Linear Algebra and its Applications*, 34:235–258, 1980. ISSN 0024-3795. doi: https://doi.org/10.1016/0024-3795(80)90167-6. URL https://www.sciencedirect.com/science/article/pii/0024379580901676.

[29] Nicolas Papernot, Patrick McDaniel, Arunesh Sinha, and Michael P. Wellman. Sok: Security and privacy in machine learning. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 399–414, 2018. doi: 10.1109/EuroSP.2018.00035.

[30] B. T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4 (5):1–17, 1964. ISSN 0041-5553. doi: https://doi.org/10.1016/0041-5553(64)90137-5. URL https://www.sciencedirect.com/science/article/pii/0041555364901375.

[31] E. Riba, D. Mishkin, D. Ponsa, E. Rublee, and G. Bradski. Kornia: an open source differentiable computer vision library for pytorch. In *Winter Conference on Applications of Computer Vision*, 2020. URL https://arxiv.org/pdf/1910.02190.pdf.

[32] Ali Shafahi, W. Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein. Poison frogs! targeted clean-label poisoning attacks on neural networks, 2018. URL https://arxiv.org/abs/1804.00792.

[33] Shiliang Sun, Zehui Cao, Han Zhu, and Jing Zhao. A survey of optimization methods from a machine learning perspective. *IEEE Transactions on Cybernetics*, 50(8):3668–3681, 2020. doi: 10.1109/TCYB.2019.2950779.

[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023. URL https://arxiv.org/abs/1706.03762.

[35] Eric W. Weisstein. Frobenius Norm. URL https://mathworld.wolfram.com/FrobeniusNorm.html. Publisher: Wolfram Research, Inc.

[36] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

[37] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks, 2017. URL https://arxiv.org/abs/1605.07146.

Appendix A

Experiments

A.1 FMNIST ResNet-18 Training Regime

We perform a hyperparameter search on the ResNet-18, with the following setup: We use Optuna to perform 100 trials where we train for 100 epochs, selecting a learning rate between 0.1 and 0.0001 and selecting a momentum value between 0.1 and 0.9. The best validation accuracy we get is 92.275 with a learning rate of 0.0897 and a momentum of 0.887. We use the Optuna MedianPruner to stop trials early that are unpromising. The seed we used is 112.

For our method we perform a hyperparameter search on the learning rate used by the cosine similarity maximization, where we select an a learning rate between 0.00001 and 0.001. We also select an epsilon value between 0.001 and 1. We perform this hyperparameter search for iteration set to 30 and iterations set to 100. We then run the hyperparameter searches for 100 trials using the hyperparameters found from the best model above.