Weight-Sharing Regularization

Mehran Shakerinava[*,1,2] **Motahareh Sohrabi**[*,2,3]
Siamak Ravanbakhsh[1,2,4] **Simon Lacoste-Julien**[2,3,4]
[1]McGill University [2]Mila - Quebec AI Institute [3]Université de Montréal [4]CIFAR AI Chair

Abstract

Weight-sharing is ubiquitous in deep learning. Motivated by this, we propose a "weight-sharing regularization" penalty on the weights $w \in \mathbb{R}^d$ of a neural network, defined as $\mathcal{R}(w) = \frac{1}{d-1}\sum_{i>j}^{d} |w_i - w_j|$. We study the proximal mapping of $\mathcal{R}$ and provide an intuitive interpretation of it in terms of a physical system of interacting particles. We also parallelize existing algorithms for $\text{prox}_{\mathcal{R}}$ (to run on GPU) and find that one of them is fast in practice but slow ($O(d)$) for worst-case inputs. Using the physical interpretation, we design a novel parallel algorithm which runs in $O(\log^3 d)$ when sufficient processors are available, thus guaranteeing fast training. Our experiments reveal that weight-sharing regularization enables fully connected networks to learn convolution-like filters even when pixels have been shuffled while convolutional neural networks fail in this setting. Our code is available on github.

1 INTRODUCTION

All modern deep learning architectures, from Convolutional Neural Networks (CNNs) (LeCun et al., 1989) to transformers (Vaswani et al., 2017), use some form of *weight-sharing*, *e.g.*, CNNs apply the same weights at different locations of the input image. More generally, requiring a linear function to be symmetric w.r.t. a permutation group induces weight-sharing in its corresponding matrix form (Shawe-Taylor, 1989; Ravanbakhsh et al., 2017), *e.g.*, symmetry w.r.t. the group of cyclic permutations C_n produces a circulant matrix, resulting in circular convolution.

Proceedings of the 27[th] International Conference on Artificial Intelligence and Statistics (AISTATS) 2024, Valencia, Spain. PMLR: Volume 238. *Equal contribution.

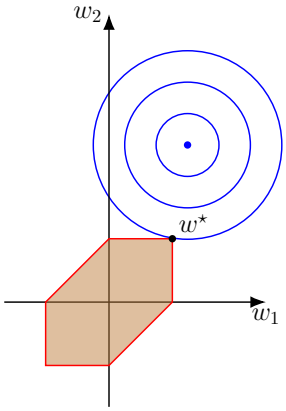

Figure 1: Depiction of the unregularized error function's contour lines (in blue), together with the constraint area for $\mathcal{R} + \ell_1$, where the optimal parameter vector w is marked by $w^\star$. Notice that the weights are set to be equal (shared) in the solution.

fully connected networks with no structure or weight-sharing are prone to overfitting the dataset. A common technique used in machine learning to avoid overfitting and improve generalization is regularization (Hastie et al., 2001). Motivated by these facts, we propose *weight-sharing regularization*. For a weight vector $w \in \mathbb{R}^d$, the weight-sharing regularization penalty is defined as

$$\mathcal{R}(w) = \frac{1}{d-1}\sum_{i>j}^{d} |w_i - w_j| \,. \tag{1}$$

Our goal is to train deep neural networks regularized with $\mathcal{R}$ (See Fig. 1). By analogy to ℓ_1 regularization where using a stochastic subgradient method (SGD) performs poorly for compression as it does not yield sufficiently small weights (Ziyin & Wang, 2023), we propose to use instead (stochastic) proximal gradient methods (Atchadé et al., 2017) which yield exactly tied weights. The efficient computation of the proximal update will be a central contribution of this paper.

Given a task-specific loss function $\mathcal{L}(w)$ we may con-

struct a weight-sharing and ℓ_1 regularized loss with coefficients α and β as

$$l(w) = \mathcal{L}(w) + \alpha\mathcal{R}(w) + \beta\ell_1(w). \quad (2)$$

The proximal gradient descent update for l with stepsize η is then given by

$$w^{(t+1)} = \text{prox}_{\eta(\alpha\mathcal{R}+\beta\ell_1)}\left(w^{(t)} - \eta\nabla\mathcal{L}(w^{(t)})\right), \quad (3)$$

where the proximal mapping is defined as

$$\text{prox}_f(x) = \underset{u}{\text{argmin}}\left(f(u) + \frac{1}{2}\|u - x\|_2^2\right). \quad (4)$$

It follows from Yu (2013, Corollary 4) that

$$\text{prox}_{\alpha\mathcal{R}+\beta\ell_1}(x) = \text{prox}_{\beta\ell_1}\left(\text{prox}_{\alpha\mathcal{R}}(x)\right).^{1} \quad (5)$$

Therefore, the main problem lies in computing $\text{prox}_{\mathcal{R}}$.

1.1 Contributions

In this work, we make the following contributions:

- We provide a new formulation of the proximal mapping of $\mathcal{R}$ in terms of the solution to an Ordinary Differential Equation (ODE). We give an intuitive interpretation of this ODE in terms of a physical system of interacting particles. This interpretation helps provide an intuitive understanding of algorithms for $\text{prox}_{\mathcal{R}}$.
- We analyze parallel versions of existing algorithms for $\text{prox}_{\mathcal{R}}$ and show that, in the worst case, they provide no speedup compared to the best sequential algorithm. Based on our physical system analogy, we propose a novel parallel algorithm with a depth of $O(\log^3 d)$ that provides an exponential speedup for worst-case inputs.
- In experiments, for the first time we are able to effectively apply weight-sharing regularization at scale in the context of deep neural networks and recover convolution-like filters in Multi-Layer Perceptrons (MLPs), even when pixels have been shuffled.

1.2 Related Works

Clustered lasso. Regularization with $\mathcal{R}$ and ℓ_1 has been studied in a linear regression setting and is known as *clustered lasso* (She, 2010). Clustered lasso is an instance of *generalized lasso* (Tibshirani & Taylor, 2011), where the regularization term is specified via a matrix $\mathbf{D}$ as $\|\mathbf{D}w\|_1$. In clustered lasso, $\mathbf{D}$ is a tall matrix with $d + \binom{d}{2}$ rows, where d rows form a diagonal matrix for ℓ_1, and each of the rest encodes the subtraction of a unique pair of weights for $\mathcal{R}$.

The proximal mapping of generalized lasso takes a simple form in the dual space and algorithms have been developed to exploit this fact (Arnold & Tibshirani, 2016). However, the dimensionality of the dual space is the same as the number of rows in $\mathbf{D}$, which makes dual methods infeasible for clustered lasso when d is large. We avoid this problem in this work by staying in the primal space.

Isotonic regression. Lin et al. (2019) show that the proximal mapping of $\mathcal{R}$ can be obtained via the isotonic regression problem

$$\min_{x\in\mathbb{R}^d} \|y - x\|_2^2 \quad \text{subject to} \quad x_1 \le x_2 \le ... \le x_n. \quad (6)$$

One can therefore use algorithms designed for isotonic regression (Best & Chakravarti, 1990) to compute the proximal mapping of clustered lasso in $O(d \log d)$ time. Moreover, based on this connection, our proposed algorithm can also be used as a fast parallel solver for isotonic regression. In this work, however, there will be no need to understand isotonic regression. Our physical interpretation provides a complete alternative specification of the problem that is much more intuitive.

Kearsley et al. (1996) propose a parallel algorithm for isotonic regression which we show to be *incorrect* with a simple counterexample, included in Appendix C.9.

Symmetry. Our work is also related to a body of work in invariant and equivariant deep learning with finite symmetry groups, as well as prior work on regularization techniques and network compression. In particular, using (soft) weight-sharing for regularization is motivated from two perspectives: symmetry, and minimum description length.

As shown in Shawe-Taylor (1989); Ravanbakhsh et al. (2017), requiring a linear function to respect a permutation symmetry induces weight-sharing in its corresponding matrix form. The same idea can be expressed using a generalization of convolution to symmetry groups (Cohen & Welling, 2016). Identification of weight-sharing patterns can be used for discovering symmetries of the data or task. Zhou et al. (2020); Yeh et al. (2022) pursue this goal in a meta-learning setting and using bilevel optimization, respectively.

Compression. However, not all weight-sharing patterns correspond to a symmetry transformation.[2] Minimum Description Length (MDL) provides a more gen-

[1] See Appendix C.8 for an alternative proof.

[2] In technical terms, a weight-sharing pattern corresponds to equivariance with respect to a permutation group, if and only if the parameters corresponding to the

eral motivation. According to MDL, the best model is the one that has the shortest description of the model itself plus its prediction errors (Rissanen et al., 2007). Assuming a prior over the weights so as to achieve the best compression, MDL leads to regularization. In particular, the soft weight-sharing of Hinton & Van Camp (1993) is derived from this perspective and motivates follow-up works on network compression (Ullrich et al., 2017). While these works assume a Gaussian mixture prior for the weights, leading to ℓ_2 regularization, we use sparsity inducing ℓ_1 regularization to encourage exact weight-sharing. Indeed, assuming ℓ_0 in Eq. (1), the objective is similar to vector quantization for compression; instead of using dc bits, where c is the number of bits-per-weight, using exact weight-sharing with k distinct weights, one requires only $kc + d \log k$ bits, leading to significant compression for small k. This observation motivates the regularization of Eq. (1) from the MDL point of view.

1.3 Outline

The required background on optimization and parallel computation for this paper is covered in Appendix A. We begin in Section 2 by studying the subdifferential of $\mathcal{R}$. Next, in Section 3, we show that the proximal mapping can be obtained by simulating a physical system of interacting particles. In Section 4, we study and analyze parallel algorithms for computing the proximal mapping of $\mathcal{R}$. We introduce a novel low-depth parallel algorithm while showing that previously existing algorithms have high depth. In Section 5, we describe a method for controlling the undesirable bias of ℓ_1-relaxation. In Section 6, we apply our algorithm to training weight-sharing regularized deep neural networks with proximal gradient descent on GPU. We discuss limitations and directions for future work in Section 7 and conclude in Section 8.

2 SUBDIFFERENTIAL OF $\mathcal{R}$

Obtaining the subdifferential of $\mathcal{R}$ will be essential to deriving its proximal mapping. We will do this by writing $\mathcal{R}$ as a point-wise maximum of linear functions. We will refer to each element of the weight vector w as a weight.

The function $\mathcal{R}$ *is a convex and piece-wise linear function.* It will be useful to keep in mind that an absolute value can be written as a max, *e.g.*, $|x| = \max(x, -x)$. Each piece of $\mathcal{R}$, when extended, forms a tangent hyperplane. Depending on the sign that we choose for each absolute value term of $\mathcal{R}$, we get the equation of one of these hyperplanes. The correct signs are determined by the ordering of weights. More specifically, there is a one-to-one correspondence between hyperplanes and the ordering of the weights. It follows that $\mathcal{R}$ has $d!$ pieces.

For each assumed increasing ordering $\pi \in S_d$[3], the corresponding hyperplane is given by

$$h_\pi(w) = \frac{1}{d-1} \sum_{i=1}^{d} (2i - d - 1) w_{\pi_i}. \tag{7}$$

The hyperplane equation is derived by noting that the element w_{π_i} appears with positive sign when compared to the $i-1$ smaller weights $w_{\pi_1}, ..., w_{\pi_{i-1}}$ and with negative sign when compared to the $d-i$ larger weights $w_{\pi_{i+1}}, ..., w_{\pi_d}$. We thus have an equation for each piece of $\mathcal{R}$.

The convexity of $\mathcal{R}$ implies that the tangent hyperplanes are below the graph of $\mathcal{R}$. Hence, $\mathcal{R}$ is the point-wise maximum of all hyperplanes. Writing $\mathcal{R}$ in this alternate way gives the equation below.

$$\mathcal{R}(w) = \max_{\pi \in S_d} h_\pi(w) \tag{8}$$

Example 2.1. Consider $d = 3$. Then,

$$\mathcal{R}(w) = \frac{1}{2} \left(|w_1 - w_2| + |w_2 - w_3| + |w_1 - w_3| \right).$$

There are $3! = 6$ possible orderings of the weights which result in 6 possible sign combinations for the terms inside the absolute values and 6 hyperplanes. These are listed below.

Ordering	Signs	Hyperplane
$w_1 < w_2 < w_3$	$- - -$	$w_3 - w_1$
$w_1 < w_3 < w_2$	$- + +$	$w_2 - w_3$
$w_2 < w_1 < w_3$	$+ - +$	$w_1 - w_2$
$w_2 < w_3 < w_1$	$+ - -$	$w_3 - w_2$
$w_3 < w_1 < w_2$	$- + -$	$w_2 - w_1$
$w_3 < w_2 < w_1$	$+ + +$	$w_1 - w_3$

Note that some sign combinations are impossible, *e.g.*, $+ + -$, which would imply $w_1 > w_2$, $w_2 > w_3$, $w_3 > w_1$, which is impossible.

By taking the point-wise maximum of the hyperplanes we can write $\mathcal{R}$ in the following equivalent

same orbit due to the action of the group on the rows and columns of the matrix are tied together.

[3] S_d is the group of all permutations of d objects.

form.

$$\mathcal{R}(w) = \max\Big\{w_1 - w_3, w_1 - w_2, w_2 - w_3, \\ w_2 - w_1, w_3 - w_2, w_3 - w_1\Big\}$$

It is easy to see that the permutations that maximize $h_\pi(w)$ are the ones that sort w. When some weights are equal (*i.e.*, there is weight-sharing) there are multiple permutations that sort the weights. Let $\Pi_{\text{sort}}(w)$ be the set of permutations that sort w, then, for all $\pi \in \Pi_{\text{sort}}(w)$, we have $\mathcal{R}(w) = h_\pi(w)$. We refer to these h_π as the *active hyperplanes* at w.

Using this new equation for $\mathcal{R}$, we can write its subdifferential as the convex hull of the gradient of active hyperplanes at w; See Proposition A.3:

$$\partial\mathcal{R}(w) = \text{conv} \bigcup_{\pi\in\Pi_{\text{sort}}(w)} \nabla h_\pi. \tag{9}$$

An intuitive description of the subgradients of $\mathcal{R}$ is that, for weights that are equal, you may assume any ordering and calculate a gradient. You may also take any convex combination of these gradients.

3 PROXIMAL MAPPING OF $\mathcal{R}$

We describe a dynamical system for w, specifically, an ODE, such that following it for one time unit produces the proximal map of $\mathcal{R}$ at the initialization point of w, *i.e.*, $w(0)$. We construct the ODE such that w follows the average of the negative gradient of the active hyperplanes. The ODE is given by

$$\dot{w}(t) \coloneqq \frac{dw}{dt}(t) = \frac{1}{|\Pi_{\text{sort}}(w(t))|} \sum_{\pi\in\Pi_{\text{sort}}(w(t))} -\nabla h_\pi. \tag{10}$$

Note that $-\dot{w}(t)$ is a subgradient of $\mathcal{R}$ at $w(t)$.

Lemma 3.1 (Weight-sharing)**.** The ODE preserves weight-sharing. Formally, for all t,

$$w_i(t) = w_j(t) \implies \dot{w}_i(t) = \dot{w}_j(t).$$

Lemma 3.2 (Monotonic inclusion)**.** The ODE satisfies monotonic inclusion for the sets Π_{sort} and $\partial\mathcal{R}$. Specifically, for all $t_2 > t_1$:

1. $\Pi_{\text{sort}}(w(t_1)) \subseteq \Pi_{\text{sort}}(w(t_2))$
2. $\partial\mathcal{R}(w(t_1)) \subseteq \partial\mathcal{R}(w(t_2))$

Theorem 3.3 (Proximal mapping of $\mathcal{R}$)**.** When w follows the ODE,

$$w(1) = \text{prox}_{\mathcal{R}}(w(0)).$$

We will not be solving the ODE numerically. Instead, we will exploit certain properties of the ODE to obtain algorithms for exact computation of $w(1)$. These properties become clearer when we interpret $(w, \dot{w})$ as the state of a physical system of particles.

Think of each weight as a *sticky* particle with unit mass moving in a 1-dimensional space. Furthermore, this system respects conservation of mass and momentum and Newton's laws of motion.

Proposition 3.4 (Conservation of Momentum)**.** Momentum is conserved in the ODE and is equal to 0. More precisely,

$$\sum_{i=1}^{d} \dot{w}_i = 0.$$

When two particles collide, they stick, due to Lemma 3.1, and their masses and momenta are added. There are no other forces involved. These laws allow us to predict the future of the system, given the positions, masses, and velocities of all particles. They can therefore act as an intuitive replacement for the ODE.

If we initialize particle positions with the elements of w and velocities with the elements of $-\nabla\mathcal{R}(w)$ (assuming all weights are distinct), then, the particle positions at time $t = 1$ give us the proximal mapping of $\mathcal{R}$ at w, due to Theorem 3.3. To calculate the positions at time $t = 1$ we need to identify collisions from $t = 0$ to $t = 1$.

We will be using the particles interpretation in the remainder of the paper. We denote the position, velocity, and mass of all particles by vectors $x, v \in \mathbb{R}^d$, and $m \in \mathbb{N}^d$, respectively. We index particles by their ordering in space from left to right, *i.e.*, $x_i \le x_{i+1}$. The vector $y \coloneqq x + v$ will be of use, so we assign it a symbol. It denotes the particle destinations after one time unit when no collisions occur. Lin et al. (2019) prove that applying isotonic regression to y results in the proximal mapping of $\mathcal{R}$. In this paper, however, no knowledge of isotonic regression will be required.

4 ALGORITHMS FOR $\text{prox}_{\mathcal{R}}$

All of the algorithms in this section expect w to have been sorted and assigned to x as a pre-processing step. Sorting has time complexity $O(d \log d)$ on a sequential machine. In the context of parallel algorithms, we assume access to a Parallel Random Ac-

cess Machine (PRAM) with p processors and $O(d)$ memory. The parallel time complexity of sorting is $O(\frac{d \log d}{p} + \log d)$ (Ajtai et al., 1983; Cole, 1988). The performance of sorting puts a ceiling on the performance that we can expect from algorithms for $\text{prox}_{\mathcal{R}}$.

4.1 Imminent Collisions Algorithm

Matching the performance ceiling is easy on a sequential machine. In fact, apart from sorting, the rest of the algorithm can be implemented in $O(d)$ time. The key idea is that the order of performing collisions does not matter, *i.e.*, the collision operation is associative. All that matters is which collisions occur. Thus, one can maintain a queue of all detected future collisions and iteratively pop the queue, perform a collision, check collisions for the new particle, and push newly detected collisions into the queue. To perform collisions efficiently, the vectors x, v, m must be stored in doubly linked lists.

A collision is detected whenever $(y_i = x_i + v_i) \geq (y_{i+1} = x_{i+1} + v_{i+1})$. We will refer to these collisions as *imminent collisions* since they will occur regardless of other collisions.[4] This algorithm is known in the literature as the *pool adjacent violators algorithm* (Best & Chakravarti, 1990).

The algorithm above can be parallelized to some extent. We can detect all imminent collisions and perform them in parallel. The process is repeated until there are no more imminent collisions. We call this algorithm, ImminentCollisions (pseudocode is provided in Appendix D).

ImminentCollisions is fast when the total number of collisions is small but can be slow when a large cluster of particles forms. For example, consider

$$y = \left[1, 1 - \frac{2}{d-1}, 1 - \frac{4}{d-1}, ..., 0, \epsilon, 2\epsilon, ..., \lfloor \frac{d}{2} \rfloor \epsilon\right],$$

for some $0 < \epsilon \ll 1$. One can work out that all particles will eventually collide, but it will take $\frac{d}{2}$ imminent collisions rounds for all particles in the right half to collide since they are sorted. Therefore, in the worst case, the number of rounds is $O(d)$. Consequently, ImminentCollisions has a worst-case parallel time complexity of $O(d)$ and might not benefit from parallelization.

4.2 End Collisions Algorithm

We continue our study of algorithms by noting some properties of the dynamical system. Let $\text{avg}(u)$ denote the average of the elements of the vector u, and let $u_{i:j}$ for integers $i \leq j$ denote a vector of size $j - i + 1$ constructed from elements i to j (inclusive) of u.

1. When particles collide, they stick, so they cannot cross each other. Thus, the ordering of particles always remains the same.

2. When particles i and $i + 1$ collide, it must have been the case that $v_i > v_{i+1}$. Hence, after collision, the velocity of particle i increases in the negative direction, and the velocity of particle $i + 1$ increases in the positive direction. This also tells us that imminent collisions definitely occur, regardless of other collisions; A property that ImminentCollisions relies on.

3. Conservation of momentum implies that $\text{avg}(y_{i:j})$ represents the final center of mass of particles i to j under the assumption that particles i to j form a closed system, that is, particles i and $i - 1$ do not collide and particles j and $j+1$ do not collide.

The proposition below tells us how many particles will collide with the left-most particle.

Proposition 4.1 (Rightmost collision)**.** The rightmost collision of particle 1 is with particle $\text{argmin}_j \text{avg}(y_{1:j})$. That is,

$$\text{particles 1 and } i \text{ collide} \iff i \leq \operatorname*{argmin}_j \text{avg}(y_{1:j}).$$

Proposition 4.1 suggests another algorithm. We repeatedly find all the particles that collide on one end, merge them, and then remove them, since they form a closed system that has no effect on the rest of the system. We call this algorithm, EndCollisions (pseudocode is provided in Appendix D). The sequential version of this algorithm is known in the literature as the *minimum lower sets algorithm* (Best & Chakravarti, 1990) and runs in $O(d^2)$ time.

If we let c be the final number of particle clusters, then the parallel time complexity is $O(\frac{dc}{p} + c \log d)$ for EndCollisions. The reason is that there are c iterations, each one taking parallel time $O(\frac{d}{p} + \log d)$ due to computing $\text{argmin}_j \text{avg}(y_{1:j})$. This algorithm is, thus, very fast when the number of collisions is very large and a small number of particle clusters forms in the end. In the worst case, however, there are no collisions, and the algorithm takes $O(\frac{d^2}{p} + d \log d)$ parallel time, which is slow. The algorithm can be made slightly faster by only considering particles that participate in an imminent collision, but the overall worst-case complexity remains the same.

[4] Non-imminent collisions are harder to detect. For an if and only if condition for the collision of neighbouring particles, see Appendix C.6.

4.3 Search Collisions Algorithm

So far we have seen rather trivial parallelizations of existing algorithms. We now describe a novel parallel divide and conquer algorithm that is fast for all inputs and has a depth of $O(\log^3 d)$.

We split the d particles into two halves of size $\frac{d}{2}$ (suppose d is even) and solve each half as a closed system in parallel recursively. Next, we need to consider the interactions between the two halves. There can only be an imminent collision at $(\frac{d}{2}, \frac{d}{2}+1)$. If there is no imminent collision, then there is no interaction between the halves and the solution has been found. Otherwise, $\frac{d}{2}$ and $\frac{d}{2}+1$ will collide, and the new particle may collide with one of its neighbours, and so on. The chain of collisions causes this cluster of particles to grow until eventually it consumes some particles $i^\star \le \frac{d}{2}$ to $j^\star > \frac{d}{2}$. The goal will be to find the ends $i^\star$ and $j^\star$ of the cluster so that we can perform the collisions and obtain the final solution.

We show that if we pick an index $i \ge i^\star$ from the left half and calculate its *rightmost collision*

$$\mathrm{rmc}(i) \coloneqq \mathrm{argmin}_{k \ge i} \mathrm{avg}(y_{i:k}), \tag{11}$$

then $\mathrm{rmc}(i) > \frac{d}{2}$. And if we pick an index $i < i^\star$, then $\mathrm{rmc}(i) = i < \frac{d}{2}$. We can therefore use the condition $\mathrm{rmc}(i) > \frac{d}{2}$ to perform a binary search on i to find $i^\star$. The other end $j^\star$ is then obtained from $\mathrm{rmc}(i^\star)$, due to Proposition 4.1.

Theorem 4.2. Suppose $(k, k+1)$ is the only imminent collision and particles $i^\star$ to $j^\star$ will collide. Then, for any index $i \le k$:

1. $i < i^\star \implies \mathrm{rmc}(i) = i$
2. $i \ge i^\star \implies \mathrm{rmc}(i) > k$

The binary search consists of $O(\log d)$ steps, each with a depth of $O(\log d)$, due to computing $\mathrm{argmin}_k \mathrm{avg}(y_{i:k})$. Thus, merging the solution of subproblems has depth $O(\log^2 d)$. There will be $\log d$ levels of merging in the divide and conquer binary tree, resulting in a total depth of $O(\log^3 d)$. Each step of binary search requires $O(d)$ operations in total on all subproblems. There are $\log^2 d$ binary search steps during the entire algorithm, giving a total work of $O(d \log^2 d)$. Altogether, we get a parallel time complexity of $O(\frac{d \log^2 d}{p} + \log^3 d)$. We call this algorithm, SearchCollisions; See Algorithm 1.

With $p \in \Omega(\frac{d}{\log d})$ processors, there is no asymptotic slowdown of the algorithm from a lack of processors and the running time is $O(\log^3 d)$. If there are too few processors, $p \in o(\log^2 d)$, then the work-inefficiency of SearchCollisions makes it worse than ImminentCollisions. However, with modern GPUs, the number of processors does not become a bottleneck for any realistic problem size. For example, considering each of the 5,120 CUDA cores of NVIDIA V100 as a processor and ignoring coefficients in the running times, SearchCollisions has better worst-case performance for d up to $2^{\sqrt{p}} = 2^{\sqrt{5120}} \approx 10^{21}$.

Algorithm 1 Search Collisions Algorithm

```
function PerformCollisions(x, v, m, i, j)
  m_total ← sum(m[i : j])
  x[i] ← sum(x[i : j] ⊙ m[i : j])/m_total
  v[i] ← sum(v[i : j] ⊙ m[i : j])/m_total
  m[i] ← m_total
  m[i + 1 : j] ← 0
end function

function RightmostCollision(x, v, m, i)
  d ← size(x)
  y_cumsum ← prefix_sum((x + v)[i : d] ⊙ m[i : d])
  avg ← y_cumsum/ prefix_sum(m[i : d])
  j ← i + argmin(avg) − 1
  return j
end function

function Merge(x, v, m)
  d ← size(x)
  le ← 1
  ri ← d/2 + 1
  for t = 1 to log2 d − 1 do
    mid = ⌊(le + ri − 1)/2⌋
    j ← RightmostCollisions(x, v, m, mid)
    if j ≥ d/2 + 1 then
      ri ← mid + 1
    else
      le ← mid + 1
    end if
  end for
  j ← RightmostCollision(x, v, m, le)
  if j > le then PerformCollisions(x, v, m, le, j)
end function

function SearchCollisions(x, v, m)
  if size(x) = 1 then return
  Concatenate zeros at the ends of x, v, m to make sizes be powers of 2
  d ← size(x)
  pardo
    SearchCollisions(x[1 : d/2], v[1 : d/2], m[1 : d/2])
    SearchCollisions(x[d/2 + 1 :], v[d/2 + 1 :], m[d/2 + 1 :])
  end
  Merge(x, v, m)
end function
```

5 REWINDING

It is common in machine learning to use ℓ_1 regularization to obtain sparse solutions. But sparsity is essentially an ℓ_0 constraint and ℓ_1 is employed as a *convex* relaxation of ℓ_0. The ℓ_1 norm does more than just

Table 1: Summary of the time-complexities of the parallel algorithms studied in this work.

Parallel Algorithm	time complexity
Imminent collisions	$O(d)$
End collisions	$O(\frac{d^2}{p} + d \log d)$
Search collisions	$O(\frac{d \log^2 d}{p} + \log^3 d)$

sparsify the solution; It also makes all weights smaller. To rectify this side-effect, we propose *rewinding*. The amount of rewinding will be parameterized by ρ.

Rewinding for ℓ_1 amounts to moving weights that were not zeroed back towards where they started. When $\rho = 1$, the weights completely return to where they started and the side-effect is entirely removed. This is equivalent to the proximal mapping of ℓ_0, which results in the iterative hard-thresholding algorithm (Blumensath & Davies, 2008). If $\rho = 0$, there is no rewinding, and we recover the proximal mapping of ℓ_1.

Algorithm 2 $\text{prox}_{\alpha\mathcal{R}+\beta\ell_1}$ with rewinding ρ.

1: **function** PROX(w, α, β, ρ)
2: $d \leftarrow \text{size}(w)$
3: $x, sort\_indices \leftarrow \text{sort}(w)$
4: $v \leftarrow \frac{\alpha}{d-1}(d - 2 \cdot \text{range}(d) + 1)$ ▷ This is $-\alpha\nabla\mathcal{R}(x)$
5: $m \leftarrow \text{ones\_array}(d)$
6: $x, v, m \leftarrow *\text{COLLISIONS}(x, v, m)$
7: $zero\_mask \leftarrow |x + v| < \beta$
8: $v \leftarrow v - \beta \cdot \text{sign}(x + v)$
9: $x[zero\_mask] \leftarrow 0$
10: $v[zero\_mask] \leftarrow 0$
11: $x \leftarrow x + (1 - \rho) \cdot v$
12: $x \leftarrow \text{repeat\_interleave}(x, m)$
13: $w[sort\_indices] \leftarrow x$
14: **end function**

The same concept can be applied to weight-sharing regularization.

Algorithm 2 applies weight-sharing and sparsity regularization with rewinding. Before line 11, x satisfies weight-sharing and sparsity and v contains the remaining displacement of the proximal mapping, that is, $\text{prox}_{\alpha\mathcal{R}+\beta\ell_1}(w) = x + v$. However, adding v to x does not induce any additional sparsity or weight-sharing. We, thus, consider v as the side-effect of convexifying sparsity and weight-sharing. To reduce the side-effect, we scale v by $1 - \rho$ for rewinding.

As an example, for sparsity regularization with ℓ_1, after processing constraints we have $x = w\mathbf{1}_{[|w|>\beta]}$ and $v = -\beta\mathbf{1}_{[|w|>\beta]}$. The ρ-rewinded update is then given by

$$x + (1 - \rho)v = (w - (1 - \rho)\beta)\mathbf{1}_{[|w|>\beta]}. \tag{12}$$

This is rather similar to β-LASSO (Neyshabur, 2020), which uses a parameter $\beta > 1$ (different from our β) to perform more aggressive thresholding. The analog of their β in our method is $\frac{1}{1-\rho}$. However, their proposed method has a distinction from proximal methods in that the gradients of ℓ_1 and the task-loss $\mathcal{L}$ are computed at the same point.

6 EXPERIMENTS

6.1 MNIST on a Torus

We consider a translation-invariant version of the MNIST dataset which we refer to as MNIST on a torus. We investigate whether our regularization can enhance the generalization of an MLP. While the answer to this question is positive, an invariant CNN model achieves higher accuracy compared to an MLP model regularized with $\mathcal{R}(w)$. However, CNNs make strong assumptions about the topology (i.e., pixel locality) and symmetry of data. When these assumptions are violated, they perform poorly, while the performance of our method is unaffected. To showcase this, we introduce a change in the dataset by performing a permutation on the pixels that destroys the translational symmetry of the images. In this setting, our results indicate that an MLP with weight-sharing regularization achieves the best accuracy, outperforming CNN by a significant margin.

Figure 2: Sample digits from MNIST on a torus.

Dataset description. We consider a version of the MNIST dataset (LeCun et al., 1989) in which the images lie on a torus. That is, pixels that cross the left boundary of the image reappear at the right boundary and vice versa. The same happens with the upper and bottom boundaries; See Fig. 2. Each image in the dataset is randomly translated. The task is, as usual, to classify the digits. In this setting, the task is invariant to the natural action of the group of 2D circular translations $\mathbb{Z}_k \times \mathbb{Z}_k$, where k is the width and height of the input images.

In our experiments, we also consider two variants of this dataset where pixels have been shuffled: (1) With symmetry but no locality. (2) With neither symmetry nor locality. Both of these variants are obtained by applying a fixed permutation of pixels on each image of the dataset. However, the order of this permutation w.r.t. the 2D circular translation is important. Performing the permutation before translation destroys the locality of pixels while maintaining symmetry whereas performing the operations in the opposite order removes symmetry as well as locality.

Experiment configuration. To simulate data scarcity, we train with 60% of the MNIST dataset. As a baseline, we use an invariant CNN model and an MLP that matches the architecture of the CNN, with weight-sharing and sparsity removed.

Prediction asymmetry metric. We also provide a metric to measure the symmetry of each model. We define the *prediction asymmetry* of a model as the percentage of test examples for which there exist two translations of the input whose model predictions do not agree. We see that weight-sharing regularization has produced a more symmetric model w.r.t. this metric compared to other baselines.

Results. As shown in Table 2, we observed that the unregularized fully connected neural network overfits the data. While it achieves 99.89% training accuracy, test accuracy peaks at 92.40%. Our weight-sharing regularization technique narrows the generalization gap, achieving 95.04% accuracy. Weight-sharing regularization also reduces the prediction asymmetry of an MLP model significantly. However, an invariant CNN is by design the best model for the MNIST on torus dataset and achieves the highest accuracy.

Table 2: Results for the MNIST on torus experiment.

Model	Test Acc.	Pred. Asym.
CNN	**98.82%**	**0.0%**
CNN (-locality)	97.98%	0.0%
CNN (-symmetry)	71.32%	85.7%
MLP	92.40%	44.5%
MLP + ℓ_1	93.57%	37.5%
MLP + $\mathcal{R}$	**95.04%**	**31.6%**

We now consider the shuffled MNIST on torus datasets. The training of MLP-based models is completely unaffected in these datasets, and all of them achieve the exact same accuracy. As only the ordering of the input has changed, the MLP-based models can simply learn a permutation of the weights in the first layer to achieve the same performance as before. From Table 2 we observe that the absence of locality does not seem to significantly affect CNNs; however, the accuracy of this model drops by more than 25% in the absence of symmetry. In this experiment, the advantage of weight-sharing regularization becomes evident, as it achieves the best accuracy among all models. Unlike CNN with its hard-coded bias, weight-sharing regularization aids MLP in dynamically learning the right weight-sharing pattern.

6.2 CIFAR10

We consider the task of training a shallow CNN and its corresponding fully connected network on CIFAR10 (Krizhevsky et al., 2009) as suggested by (Neyshabur, 2020). The author suggests a variant of the ℓ_1 regularizer, β-LASSO, to learn convolution-like structures from scratch. Our experiments show that adding weight-sharing regularization to the fully connected neural network enables it to more effectively learn convolution-like patterns, which are recognized as optimal for vision data.

Table 3: Results for the CIFAR10 experiment.

Model	Test Acc.	Sparsity	Weight Sh.
CNN	81.85%	99.99%	99.61%
MLP	69.20%	0.0%	91.05%
β-LASSO	71.37%	95.33%	41.96%
Our ℓ_1	73.66%	99.58%	2.04%
$\mathcal{R}$ (Subgradient)	69.96%	0.0%	2.98%
$\mathcal{R}$	73.50%	0.0%	99.96%
$\mathcal{R}$ + Our ℓ_1	**73.75%**	99.71%	1.17%

Experiment configuration. We follow the setup of (Neyshabur, 2020) and train a 3-layer fully connected network on a heavily augmented version of CIFAR10. The augmentations involve cropping and small rotations of the images, therefore, when training with weight-sharing regularization, we expect to see similar filters that are translated and tilted. We train the models for 400 epochs with a batch-size of 512 and a learning-rate of 0.1. We performed a hyperparameter sweep for α and β for each method and fixed ρ to 0.98.

Weight-sharing metric. Our measure of weight-sharing in Table 3 is obtained by the formula $\frac{\#\text{non-zero distinct weights}}{\#\text{non-zero weights}}$. We do not include zeroed weights so that sparsity does not affect the measure.

Results. Table 3 shows that both our weight-sharing regularization and our adaptation of ℓ_1 regularization improve test accuracy compared to MLP and β-LASSO. We note that regularization with $\mathcal{R}$ achieves a weight-sharing measure that is closely aligned with that of the CNN network. As shown in the last row of Table 3, we also observe that incorporating ℓ_1 regularization into weight-sharing is likely to drive the shared weights towards zero.
Table 3 also highlights the importance of using the proximal gradient descent algorithm compared to the subgradient method, which fails to achieve weight-sharing when trained with $\mathcal{R}$.

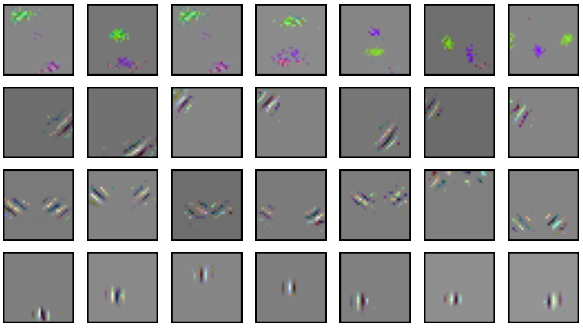

Figure 3: The emergence of learned convolution-like filters in a fully connected network with weight-sharing regularization on CIFAR10.

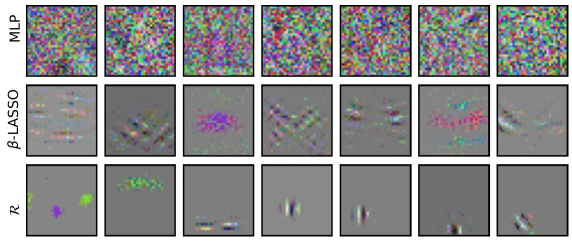

Figure 4: A random sample of learned filters from the 256 filters with the highest number of non-zero weights.

By visualizing the rows of the first layer's weight matrix, we notice some clusters of similar and convolution-like patterns. Some of these similar patterns are demonstrated in Fig. 3. While (Neyshabur, 2020) recreated sparsity of patterns, we also observe sparse weights with weight-sharing among them. These local patterns are translated across the input image to simulate convolution in the fully connected neural network.

Practical remarks. By comparing the training times of various algorithms, we observe that ImminentCollisions is faster for weights encountered during training compared to SearchCollisions. We observed that ImminentCollisions requires ∼500 rounds of iterations for $d \approx 10^8$ while in the worst case, it would require d rounds. We, therefore, recommend using SearchCollisions as a backup in case ImminentCollisions gets stuck on a difficult input. Such a setup will *guarantee* fast training. Further details regarding runtimes can be found in Appendix E.3.

7 FUTURE WORK

Even though $\text{prox}_{\mathcal{R}}$ greatly benefits from parallelization, training with weight-sharing regularization is still several times slower than training without. To bring down the cost of training, it might be possible to reuse computation from the previous training step. Is it possible to design an algorithm that starts with a candidate weight-sharing, and then edits it to obtain the correct weight-sharing?

Weight-sharing regularization with rewinding introduces new hyper-parameters α and ρ that have to be tuned. The hyper-parameters can be fixed or they can vary during training. There is, therefore, a large space of possibilities to explore, and it is currently unclear what kind of training scheme is best.

A possible generalization of $\mathcal{R}$ is given by

$$\mathcal{R}_{\mathbf{C}}(w) = \frac{1}{d-1} \sum_{i<j} \mathbf{C}_{i,j} \|w_i - w_j\|_2, \qquad (13)$$

where $\mathbf{C} \in \mathbb{R}_+^{d \times d} > 0$ is a symmetric matrix and $w \in \mathbb{R}^{d \times k}$. We have studied $\mathbf{C} = \mathbf{1}$, where all pairs of weights are encouraged to be equal, and $k = 1$, where each weight is a scalar. As an example, when

$$\mathbf{C}_{i,j} = \begin{cases} 1 & |i-j| = 1 \\ 0 & \text{otherwise} \end{cases},$$

we retrieve the fused lasso regularizer (Tibshirani et al., 2005). Does there exist fast parallel algorithms for $\text{prox}_{\mathcal{R}_{\mathbf{C}}}$ as well?

Lastly, we leave the study of rewinding and its properties for future work.

8 CONCLUSION

We have proposed "weight-sharing regularization" via $\mathcal{R}$ and obtained its proximal mapping by formulating it in terms of the solution to an ODE. By analogy to a physical system of interacting particles, we developed a highly parallel algorithm suitable for modern GPUs. Our experiments indicate that this method can be applied to the training of weight-sharing regularized deep neural networks using proximal gradient descent to achieve better generalization in the low-data regime. While our preliminary results are encouraging, there's room for further research and improvement in this domain.

Acknowledgments

We thank Juan Duque, Juan Ramirez, Gauthier Gidel, Quentin Bertrand, and Reza Babanezhad for their feedback on an initial draft of this paper. This research was enabled in part by computing resources provided by Mila (mila.quebec). This work was supported by the Canada CIFAR AI Chair Program. Simon Lacoste-Julien is a CIFAR Associate Fellow in the Learning Machines & Brains program.

References

M. Ajtai, J. Komlós, and E. Szemerédi. An O(n log n) sorting network. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pp. 1–9, 1983.

Taylor B Arnold and Ryan J Tibshirani. Efficient implementations of the generalized lasso dual path algorithm. *Journal of Computational and Graphical Statistics*, 25(1):1–27, 2016.

Yves F Atchadé, Gersende Fort, and Eric Moulines. On perturbed proximal gradient algorithms. *The Journal of Machine Learning Research*, 18(1):310–342, 2017.

Michael J Best and Nilotpal Chakravarti. Active set algorithms for isotonic regression; a unifying framework. *Mathematical Programming*, 47(1-3):425–439, 1990.

Thomas Blumensath and Mike E Davies. Iterative thresholding for sparse approximations. *Journal of Fourier analysis and Applications*, 14:629–654, 2008.

Frank H Clarke. *Optimization and nonsmooth analysis*. SIAM, 1990.

Taco Cohen and Max Welling. Group equivariant convolutional networks. In *International conference on machine learning*, pp. 2990–2999. PMLR, 2016.

Richard Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. The elements of statistical learning. springer series in statistics. *New York, NY, USA*, 2001.

Geoffrey E Hinton and Drew Van Camp. Keeping the neural networks simple by minimizing the description length of the weights. In *Proceedings of the sixth annual conference on Computational learning theory*, pp. 5–13, 1993.

Anthony J Kearsley, Richard A Tapia, and Michael W Trosset. An approach to parallelizing isotonic regression. In *Applied Mathematics and Parallel Computing: Festschrift for Klaus Ritter*, pp. 141–147. Springer, 1996.

Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

Meixia Lin, Yong-Jin Liu, Defeng Sun, and Kim-Chuan Toh. Efficient sparse semismooth newton methods for the clustered lasso problem. *SIAM Journal on Optimization*, 29(3):2026–2052, 2019.

Behnam Neyshabur. Towards learning convolutions from scratch. In *Advances in Neural Information Processing Systems*, volume 33, pp. 8078–8088, 2020.

Siamak Ravanbakhsh, Jeff Schneider, and Barnabas Poczos. Equivariance through parameter-sharing. In *International conference on machine learning*, pp. 2892–2901. PMLR, 2017.

Jorma Rissanen et al. *Information and complexity in statistical modeling*, volume 152. Springer, 2007.

John Shawe-Taylor. Building symmetries into feedforward networks. In *Artificial Neural Networks, 1989., First IEE International Conference on (Conf. Publ. No. 313)*, pp. 158–162. IET, 1989.

Yiyuan She. Sparse regression with exact clustering. *Electronic Journal of Statistics*, 4:1055 – 1096, 2010.

Robert Tibshirani, Michael Saunders, Saharon Rosset, Ji Zhu, and Keith Knight. Sparsity and smoothness via the fused lasso. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 67(1):91–108, 2005.

Ryan J. Tibshirani and Jonathan Taylor. The solution path of the generalized lasso. *The Annals of Statistics*, 39(3):1335 – 1371, 2011.

Karen Ullrich, Edward Meeds, and Max Welling. Soft weight-sharing for neural network compression. *arXiv preprint arXiv:1702.04008*, 2017.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Raymond A Yeh, Yuan-Ting Hu, Mark Hasegawa-Johnson, and Alexander Schwing. Equivariance discovery by learned parameter-sharing. In *International Conference on Artificial Intelligence and Statistics*, pp. 1527–1545. PMLR, 2022.

Yao-Liang Yu. On decomposing the proximal map. In *Advances in Neural Information Processing Systems*, volume 26, 2013.

Allan Zhou, Tom Knowles, and Chelsea Finn. Meta-learning symmetries by reparameterization. *arXiv preprint arXiv:2007.02933*, 2020.

Liu Ziyin and Zihao Wang. spred: Solving l1 penalty with SGD. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202, pp. 43407–43422. PMLR, 2023.

Checklist

1. For all models and algorithms presented, check if you include:
   (a) A clear description of the mathematical setting, assumptions, algorithm, and/or model. *Yes. We provide precise pseudocode for all algorithms.*
   (b) An analysis of the properties and complexity (time, space, sample size) of any algorithm. *Yes. time complexity is provided for all algorithms and space is always $O(d)$.*
   (c) (Optional) Anonymized source code, with specification of all dependencies, including external libraries. *Yes. Our code is available on github.*
2. For any theoretical claim, check if you include:
   (a) Statements of the full set of assumptions of all theoretical results. *Yes.*
   (b) Complete proofs of all theoretical results. *Yes. All proofs are provided in Appendix C.*
   (c) Clear explanations of any assumptions. *Yes.*
3. For all figures and tables that present empirical results, check if you include:
   (a) The code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL). *Yes. We have released our source code on github to facilitate the reproduction of all experimental results.*
   (b) All the training details (e.g., data splits, hyperparameters, how they were chosen). *Yes. Some training details are provided in the text and the rest is provided in the appendix.*
   (c) A clear definition of the specific measure or statistics and error bars (e.g., with respect to the random seed after running experiments multiple times). *No. The measures that we use are mentioned, but we do not provide error bars.*
   (d) A description of the computing infrastructure used. (e.g., type of GPUs, internal cluster, or cloud provider). *Yes. We use NVIDIA GPUs such as V100 and A100.*
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets, check if you include:
   (a) Citations of the creator If your work uses existing assets. *Yes. We have cited relevant publications for the MNIST and CIFAR10 datasets.*
   (b) The license information of the assets, if applicable. *No. These datasets are standard in machine learning.*
   (c) New assets either in the supplemental material or as a URL, if applicable. *Not Applicable.*
   (d) Information about consent from data providers/curators. *Not Applicable.*
   (e) Discussion of sensible content if applicable, e.g., personally identifiable information or offensive content. *Not Applicable.*
5. If you used crowdsourcing or conducted research with human subjects, check if you include:
   (a) The full text of instructions given to participants and screenshots. *Not Applicable.*
   (b) Descriptions of potential participant risks, with links to Institutional Review Board (IRB) approvals if applicable. *Not Applicable.*
   (c) The estimated hourly wage paid to participants and the total amount spent on participant compensation. *Not Applicable.*

A BACKGROUND

A.1 Optimization

Since $\mathcal{R}$ is not differentiable everywhere, we will be using concepts that generalize the gradient to non-differentiable functions, namely, the subgradient and subdifferential.

Definition A.1 (Subgradient)**.** g is a subgradient of a convex function f at $x \in \operatorname{dom} f$ if

$$f(y) \geq f(x) + g \cdot (y - x) \qquad \forall x \in \operatorname{dom} f.$$

Definition A.2 (Subdifferential)**.** The set of subgradients of f at the point x is called the subdifferential of f at x, and is denoted $\partial f(x)$.

$$\partial f(x) \coloneqq \left\{ g \mid f(y) \geq f(x) + g \cdot (y - x),\ \forall y \in \operatorname{dom} f \right\}$$

The subdifferential is a closed convex set. Each point $y \in \operatorname{dom} f$ puts a linear inequality constraint on g which results in a closed convex set. The result follows from the fact that the intersection of any collection of closed convex sets is closed and convex.

In the text, we rewrite $\mathcal{R}$ as a max and use the following proposition (Clarke, 1990, Proposition 2.3.12) to obtain its subdifferential.

Proposition A.3 (Subdifferential of max)**.** Let

$$f(x) = \max \{f_1(x), ..., f_k(x)\},$$

where $f_i(x)$ are differentiable convex functions. A function f_i is *active* at x if $f(x) = f_i(x)$. Let $I(x)$ denote the set of active functions at x. Then,

$$\partial f(x) = \operatorname{conv} \bigcup_{i \in I(x)} \nabla f_i(x),$$

where conv denotes the convex hull.

Proximal optimization methods rely on the proximal mapping, defined below.

Definition A.4 (Proximal mapping)**.** The proximal mapping of a closed convex function f is

$$\operatorname{prox}_f(x) \coloneqq \operatorname*{argmin}_u \left(f(u) + \frac{1}{2}\|u - x\|_2^2 \right).$$

Since $u \mapsto f(u) + \frac{1}{2}\|u - x\|_2^2$ is closed and strongly convex, the proximal map exists and is *unique*.

The *optimality condition* for minimizing a non-differentiable convex function g is $0 \in \partial g(x^\star)$. Applying this fact to the proximal mapping tells us that

$$u^\star = \operatorname{prox}_f(x) \iff x - u^\star \in \partial f(u^\star). \tag{14}$$

Intuitively, the result says that after taking a proximal step from x to $u^\star$, there exists a subgradient at $u^\star$ that takes you back to x.

Gradient descent can be seen as repeated application of the proximal mapping to a local linearization of the objective function. More explicitly, let f_y denote the linearization of f at y:

$$f_y(x) \coloneqq f(y) + (x - y) \cdot \nabla f(y). \tag{15}$$

Then, a gradient descent step on f with step-size η can be written as

$$x^{(t+1)} = x^{(t)} - \eta \nabla f(x^{(t)}) = \text{prox}_{\eta f_{x^{(t)}}}(x^{(t)}). \tag{16}$$

Whenever the objective function is the sum of a differentiable component f and a non-differentiable component h for which we can compute a proximal mapping, an optimization algorithm known as *proximal gradient descent* can be used. Similar to gradient descent, it uses a linearization of f at each time-step, however, h is not linearized. Specifically, the proximal mapping of $f_x + h$ is used. The algebraic manipulations below show that the algorithm alternates between gradient descent steps on f and proximal mappings of h.

$$\begin{aligned}
\text{prox}_{f_x+h}(x) &= \underset{u}{\text{argmin}} \left(f(x) + (u-x) \cdot \nabla f(x) + h(u) + \frac{1}{2}\|u-x\|_2^2 \right) \\
&= \underset{u}{\text{argmin}} \left(h(u) + \frac{1}{2}\|u - (x - \nabla f(x))\|_2^2 \right) \\
&= \text{prox}_h(x - \nabla f(x))
\end{aligned}$$

Therefore, proximal gradient descent is an efficient algorithm whenever prox_h can be computed efficiently. Similar to Eq. (16), a step-size can be included to get the update equation

$$x^{(t+1)} = \text{prox}_{\eta h}(x^{(t)} - \eta \nabla f(x^{(t)})). \tag{17}$$

Example A.5. As an example, let us obtain the proximal mapping of $f(x) = |x|$ for $x \in \mathbb{R}$. The subdifferential of f is given by

$$\partial f(x) = \begin{cases} \{1\} & x > 0 \\ [-1, 1] & x = 0 \\ \{-1\} & x < 0 \end{cases}.$$

The optimality condition is $0 \in \partial f(u) + u - x$, which gives us

$$0 \in \begin{cases} \{1 + u - x\} & u > 0 \\ [-1, 1] + u - x & u = 0 \\ \{-1\} + u - x & u < 0 \end{cases}.$$

We then solve for u. The three cases can be solved independently.

$$u = \begin{cases} x - 1 & x > 1 \\ 0 & x \in [-1, 1] \\ x + 1 & x < -1 \end{cases}$$

In $\mathbb{R}^d$, element-wise application of the above mapping gives us the proximal mapping of the ℓ_1 norm, also commonly known as *soft-thresholding*.

A.2 Parallel Computation

In parallel computation, we assume access to a machine with multiple processors that is able to perform multiple operations at the same time. With access to such a parallel machine with p processors, one could potentially gain a $p\times$ speedup of any sequential algorithm. The actual performance improvement depends on the structure of the algorithm and the inter-dependencies among the operations of the algorithm. For some problems, new algorithms have to be designed to be able to leverage parallelism, such as what we do in this work.

The commonly used framework for analyzing parallel algorithms is the *work-depth model*. Let n denote the size of the problem. The *depth* $D(n)$ of an algorithm, also known as the *span*, is the greatest number of basic operations

(with constant fan-in) that have to be performed *sequentially* by the algorithm due to data dependencies. Or, in other words, the length of the critical path of the algorithm's circuit (*a.k.a.,* computation graph). The *work* $W(n)$ of an algorithm is defined as the *total* number of basic operations that the algorithm has to perform.

Since no more that p operations can be performed at any time, an algorithm will require at least $W(n)/p$ rounds of parallel operations. On the other hand, an algorithm requires at least $D(n)$ rounds of parallel operations, by definition. It turns out that the running time $T(n)$ of a parallel algorithm is bounded by the sum of these lower-bounds:

$$T(n) = O\Big(\frac{W(n)}{p} + D(n)\Big). \tag{18}$$

There are certain fundamental parallel operations that are commonly used in parallel computation. We describe the ones that we use along with their parallel time complexities.

- `map` applies a given function to each element of the input list in parallel. The time complexity of `map` is $O(\frac{n}{p} + 1)$ as each processor can independently process one element of the input in constant time.
- `reduce` combines all elements of the input list using an associative operation in a hierarchical manner. We can perform a binary reduction in $O(\frac{n}{p} + \log n)$ time. In each step, half of the processors perform the associative operation with the result of their neighbouring processor, reducing the problem size by half.
- `scan` computes a running sum (or any other associative operation) of the input list. An algorithm, known as Hillis-Steele scan, works by having each processor i compute the sum of the input element at i and all preceding elements at distances of 2^j for $j = 0$ to $\log(n) - 1$, thus requiring $\log n$ steps to complete. There exists a work-efficient algorithm for `scan` that runs in time $O(\frac{n}{p} + \log n)$.

B COMPUTING $\mathcal{R}$

Proximal gradient descent does not require computing $\mathcal{R}$, nonetheless, it's interesting to analyze the complexity of computing $\mathcal{R}$ and contrast it to that of $\text{prox}_{\mathcal{R}}$.

At first glance, computing $\mathcal{R}(w)$ seems to require quadratic time in the size of w, *i.e.*, $O(d^2)$. However, if we sort the weights, $\mathcal{R}$ can be rewritten in a form that is faster to compute. Note that $\mathcal{R}$ *is permutation invariant.* Let x denote the sorted weight vector. Then,

$$\mathcal{R}(w) = \mathcal{R}(x) = \frac{1}{d-1}\sum_i \sum_{j<i} x_i - x_j = \frac{1}{d-1}\sum_i \Big((i-1)x_i - \sum_{j<i} x_j\Big).$$

Let us define a new vector s containing the prefix sum of w, that is, $s_i \coloneqq \sum_{j<i} x_j$. This vector can be computed in time $O(d)$. Next, $\mathcal{R}(w)$ can be computed with an additional $O(d)$ time. Sorting is, thus, the bottleneck of the algorithm, resulting in an $O(d \log d)$ time algorithm for $\mathcal{R}$.

This algorithm can greatly benefit from parallelization. Calculating $\mathcal{R}(x)$ only requires two prefix sums, a vector multiplication and subtraction, and a sum, which take $O(\frac{d}{p} + \log d)$ parallel time in total. Additionally, sorting can be performed in $O(\frac{d \log d}{p} + \log d)$ parallel time, giving us an $O(\frac{d \log d}{p} + \log d)$ parallel time algorithm for computing $\mathcal{R}$.

C PROOFS

C.1 Proof of Lemma 3.1

Proof. Let $\sigma \in S_d$ be the permutation that only swaps indices i and j. When $w_i = w_j$, the set $\Pi_{\text{sort}}(w)$ is the same as the set $\{\sigma\pi \mid \pi \in \Pi_{\text{sort}}(w)\}$. Therefore,

$$\sum_{\pi \in \Pi_{\text{sort}}(w)} (\nabla h_\pi)_i = \sum_{\pi \in \Pi_{\text{sort}}(w)} (\nabla h_{\sigma\pi})_i = \sum_{\pi \in \Pi_{\text{sort}}(w)} (\nabla h_\pi)_j .$$

Dividing by $-|\Pi_{\text{sort}}(w)|$ gives the desired result. □

C.2 Proof of Lemma 3.2

Proof. Let π be a permutation that sorts the weights. Assuming all weights $w_i \in \mathbb{R}$ to be distinct, we have

$$w_{\pi_1} < w_{\pi_2} < ... < w_{\pi_{d-1}} < w_{\pi_d}. \tag{19}$$

If some of these weights are equal, some $<$ signs should be replaced with $=$.

The weights $w_i(t)$ change *smoothly* in time according to the ODE, and when two weights meet, they stay equal forever, due to Lemma 3.1. Therefore, as time progresses, we may only have that, in Eq. (19), some $<$ signs turn to $=$. Clearly, previous sorting permutations are still valid while new ones are added to Π_{sort}, hence, Π_{sort} satisfies monotonic inclusion. It immediately follows that $\partial\mathcal{R}$ also satisfies monotonic inclusion by Eq. (9). □

C.3 Proof of Theorem 3.3

Proof. Note that $\dot{w}$ only changes when new weights become equal. Now suppose w follows the negative direction of subgradients $g_1, ..., g_k$ for $t_1, ..., t_k$ time units such that $\sum_{i=1}^{k} t_i = 1$. Due to monotonic inclusion of the subdifferential of $\mathcal{R}$, we have

$$g_1, ..., g_k \in \partial\mathcal{R}(w(1)).$$

The negative displacement of w (*i.e.*, $(w(0) - w(1))$ is equal to $\sum_{i=1}^{k} t_i g_i$, which is a convex combination of subgradients at $w(1)$. Therefore, the negative displacement is in the subdifferential of $w(1)$, which implies that $w(1)$ is the result of the proximal mapping. □

C.4 Proof of Proposition 3.4

Proof. Note that, for all tangent hyperplanes h of $\mathcal{R}$,

$$\sum_{i=1}^{d} (\nabla h)_i = \frac{1}{d-1} \sum_{i=1}^{d} (2i - d - 1) = 0. \tag{20}$$

Since any subgradient g of $\mathcal{R}$ is a convex combination of the gradient of tangent hyperplanes, $\sum_i g_i = 0$. As $-\dot{w}$ is a subgradient, the same fact applies. □

C.5 Proof of Proposition 4.1

Proof. $\Leftarrow$: Let us focus on where particle 1 will end up. Let y'_j denote the final position of particle j. Recall that $\text{avg}(y_{1:j})$ represents the final center of mass of particles 1 to j under the assumption that particles j and $j+1$ do not collide. If they do collide, the velocity of particle j increases in the negative direction, and so, the final center of mass will only be smaller.

Since particle 1 is always the left-most particle, we have $y'_1 \leq \text{avg}(y_{1:j})$ for all j. Minimizing over j we get $y'_1 \leq \min_j \text{avg}(y_{1:j})$. On the other hand, y'_1 is equal to the the final center of mass of *some* group of neighbouring particles that all collide, therefore,

$$y'_1 = \min_j \text{avg}(y_{1:j}), \tag{21}$$

and particle 1 collides with all particles up to $\text{argmin}_j \text{avg}(y_{1:j})$.

$\Rightarrow$: Only these particles collide with particle 1 because otherwise particle 1 would end up at a different greater position. □

C.6 Neighbour Collision

The following corollary of Proposition 4.1 provides an if and only if condition for the collision of two *neighbouring* particles.

Corollary C.1 (Neighbour collision)**.**

$$\text{Particles } i \text{ and } i+1 \text{ collide} \iff \max_{j \le i} \operatorname{avg}(y_{j:i}) \ge \min_{j \ge i+1} \operatorname{avg}(y_{i+1:j})$$

Proof. Considering particles 1 to i as a closed system then particle i is heading to $\max_{j \le i} \operatorname{avg}(y_{j:i})$. Similarly, considering particles $i+1$ to d as a closed system, particle $i+1$ is heading to $\min_{j \ge i+1} \operatorname{avg}(y_{i+1:j})$. A collision occurs if and only if the destinations cross each other. □

C.7 Proof of Theorem 4.2

Proof. **1.** If $i < i^\star$, we know that particle i does not participate in any collisions, so its rightmost collision is itself.

2. Recall that a chain of collisions starts from $(k, k+1)$ and extends outwards. By Proposition 4.1, $\operatorname{rmc}(i)$ gives us the rightmost collision of particle i when assuming it is the first particle. Therefore, we should assume that particles less than i don't exist. Without these particles, the final cluster of collided particles will still include i but will end at some j', where $k+1 \le j' \le j^\star$. The rightmost collision of i is, therefore, j', which satisfies $j' > k$. □

C.8 Combining Weight-Sharing and Sparsity

Proposition C.2 (Proximal mapping of $\mathcal{R} + \beta\ell_1$)**.**

$$\operatorname{prox}_{\mathcal{R}+\beta\ell_1}(x) = \operatorname{prox}_{\beta\ell_1}(\operatorname{prox}_{\mathcal{R}}(x))$$

Proof. We start with several definitions that will aid the readability of the proof.

$$\begin{aligned} &x \\ &\Big\downarrow \ \Delta'_{\mathcal{R}} \coloneqq x - x' \in \partial\mathcal{R}(x') \\ &x' \coloneqq \operatorname{prox}_{\mathcal{R}}(x) \\ &\Big\downarrow \ \Delta''_{\beta\ell_1} \coloneqq x' - x'' \in \partial(\beta\ell_1)(x'') \\ &x'' \coloneqq \operatorname{prox}_{\beta\ell_1}(\operatorname{prox}_{\mathcal{R}}(x)) \end{aligned}$$

We must show that $x - x'' \in \partial(\mathcal{R} + \beta\ell_1)(x'')$, or equivalently, $\Delta'_{\mathcal{R}} + \Delta''_{\beta\ell_1} \in \partial\mathcal{R}(x'') + \partial(\beta\ell_1)(x'')$. This is true if $\Delta'_{\mathcal{R}} \in \partial\mathcal{R}(x'')$. Since x'' is the result of soft-thresholding x', and since soft-thresholding preserves order and weight-sharing, similar to Lemma 3.2, we get monotonic inclusion for $\partial\mathcal{R}$, *i.e.*, $\forall x \in \mathbb{R}^d : \partial\mathcal{R}(x) \subseteq \partial\mathcal{R}(\operatorname{prox}_{\beta\ell_1}(x))$. Hence, $\Delta'_{\mathcal{R}} \in \partial\mathcal{R}(x') \subseteq \partial\mathcal{R}(x'')$, which completes the proof. □

C.9 Counterexample for Kearsley et al. (1996)

Kearsley et al. (1996) propose a parallel algorithm for isotonic regression which we show to be *incorrect* with a simple counterexample. Let

$$y = [0.7, 1, 0.9, 0.99]$$

in the isotonic regression problem of Eq. (6). Their algorithm proposes to average values $1, 0.9, 0.99$ and arrives at the solution

$$x = [0.7, 0.96\bar{3}, 0.96\bar{3}, 0.96\bar{3}],$$

while the correct solution only averages 1 and 0.9 to get

$$x^\star = [0.7, 0.95, 0.95, 0.99].$$

D PSEUDOCODES

The functions RightmostCollision and PerformCollisions are defined in Algorithm 1. We observed that Algorithm 3 is the fastest in practice for training neural networks with weight-sharing regularization. It performs an additional `prefix_sum` to detect successive imminent collisions, then performs them all at once in a single round. The `prefix_sum` adds a factor of $O(\log d)$ to the depth, however, since we've empirically observed that there aren't too many rounds in practice, the overhead is negligible.

Algorithm 3 Imminent collisions algorithm (v2)

function CollisionsRound(x, v, m)
  $n \leftarrow \text{size}(x)$
  $c \leftarrow \text{zeros\_array}(n)$
  $c[1:] \leftarrow (x+v)[:-1] < (x+v)[1:]$
  $i \leftarrow \text{prefix\_sum}(c)$
  $x \leftarrow \text{scatter}(x \odot m, i)$
  $v \leftarrow \text{scatter}(v \odot m, i)$
  $m \leftarrow \text{scatter}(m, i)$
  $x \leftarrow x/m$
  $v \leftarrow v/m$
  return x, v, m
end function

function ImminentCollisions(x, v, m)
  repeat
    $d \leftarrow \text{size}(x)$
    $x, v, m \leftarrow$ CollisionsRound(x, v, m)
  until $d = \text{size}(x)$
  return x, v, m
end function

Algorithm 4 Imminent collisions algorithm (v1)

function CollisionsRound(x, v, m)
  $d \leftarrow \text{size}(x)$
  for all odd indices $i < d$ **pardo**
    if $(x+v)[i] > (x+v)[i+1]$ **then**
      PerformCollisions($x, v, m, i, i+1$)
    end if
  end for
  return $x[m>0], v[m>0], m[m>0]$
end function

function ImminentCollisions(x, v, m)
  repeat
    $d \leftarrow \text{size}(x)$
    $x, v, m \leftarrow$ CollisionsRound(x, v, m)
    $x, v, m \leftarrow$ CollisionsRound($x[2:], v[2:], m[2:]$)
  until $d = \text{size}(x)$
  return x, v, m
end function

Algorithm 5 End collisions algorithm

function EndCollisions(x, v, m)
  $d \leftarrow \text{size}(x)$
  $i \leftarrow 1$
  while $i < d$ **do**
    $j \leftarrow$ RightmostCollision(x, v, m, i)
    PerformCollisions(x, v, m, i, j)
    $i \leftarrow j + 1$
  end while
end function

E EXPERIMENTS

E.1 MNIST on Torus

The CNN baseline is designed to be invariant to circular 2D translations. The architecture consists of two circular convolutional layers, outputting 32 and 64 channels, respectively, followed by global average pooling and a fully connected layer. Note that circular convolution is equivariant to circular translations, while global pooling is invariant, resulting in an invariant model.

The fully connected networks were obtained from the CNN baseline by replacing convolutional layers with linear layers with matching input and output sizes. In order to see convolution-like behaviour in the first two layers of the MLPs, we apply weight-sharing regularization to each of these layers separately.

We trained all the networks for 200 epochs using an initial learning-rate of 0.1 which is cosine annealed to 0. We

also used a momentum of 0.9 to achieve near-zero training error for all models within 200 epochs. As a result, all the models attained more than 99% training accuracy, except for our method when using dataset ratios of 0.8 and 1.0, where it achieved more than 98% training accuracy. To choose the weight-sharing parameter α for each layer, we ran a sweep on values 0.01, 0.001, 0.0001, and 0.00001. The selected parameters were 0.001 for the first layer and 0.0001 for the second layer. For the ℓ_1 coefficient β, we selected 0.0001 after a hyperparameter search. We don't use rewinding in these experiments.

Additionally, we use a modification to the standard ℓ_1 proximal update rule as proposed by Neyshabur (2020), which is to multiply the learning-rate in v in line 11 of Algorithm 2, instead of multiplying it in the coefficients α and β. This modification improves validation accuracy in practice.

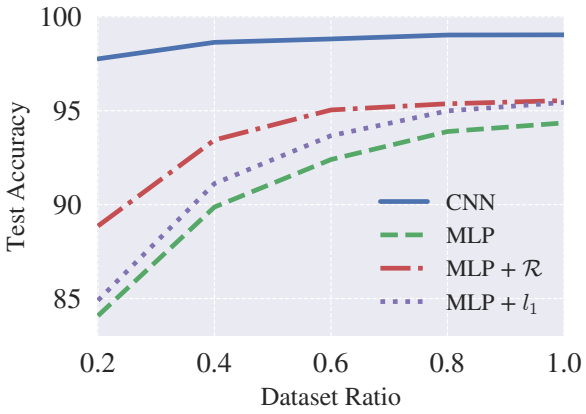

Figure 5: Test accuracy vs. dataset ratio on MNIST on torus for various models.

In Fig. 5, we investigate the effect of dataset size on test accuracy. We observe that the introduction of weight-sharing regularization serves as a beneficial inductive bias for the network, effectively narrowing the generalization gap, particularly when using a reduced dataset ratio.

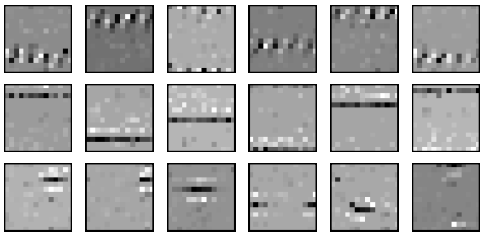

Figure 6: The emergence of learned convolution-like filters in a fully connected network with weight-sharing regularization in MNIST on torus.

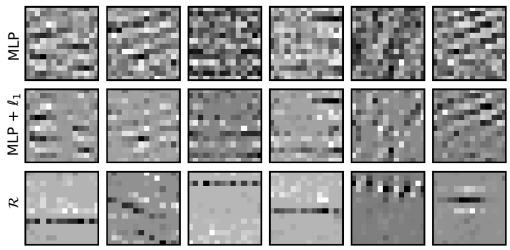

Figure 7: A random sample of learned filters from the 256 filters with the highest number of non-zero weights in MNIST on torus.

We visualize the learned filters in Figs. 6 and 7. We are able to find filters that look like translations of one

another, similar to what one finds in a convolutional layer. These are illustrated in Fig. 6. We manually grouped these filters by examining the top 256 filters with the highest number of non-zero weights. A random selection of these filters for each method is shown in Fig. 7. The same process was carried out for the CIFAR10 task in Figs. 3 and 4.

E.2 CIFAR10

This experiment is primarily designed based on the experiments in Neyshabur (2020). We train a shallow convolutional neural network with one convolutional layer followed by two fully connected layers. The number of output channels of the convolutional layer is 64. The fully connected network corresponding to this shallow convolutional network has approximately 75M parameters. Batch normalization is applied after each of the first two layers. A learning-rate of 0.1 and no momentum is used for all experiments, along with a cosine annealed learning-rate schedule, similar to Neyshabur (2020).

For the β-LASSO baseline we use their reported optimal $\beta = 50$ and a corresponding ρ of 0.98 for our own methods. We conducted a sweep for the regularization coefficient of β-LASSO (referred to as λ) which resulted in 0.00001 being selected.

For our methods, a hyperparameter search resulted in the following values: for MLP + $\mathcal{R}$, $\alpha = 0.001$; for MLP + ℓ_1, $\beta = 0.001$; and for MLP + $\mathcal{R}$ + ℓ_1, $\alpha = 0.0002$ and $\beta = 0.001$.

E.3 Runtime Comparison

Table 4: Approximate runtime of each algorithm per epoch on CIFAR10 experiments.

Algorithm	**Time (sec)**
Sequential PAV	1800
Search Collisions	200
Imminent Collisions (v2)	50
No Regularization	10

E.4 Worst-case Runtime Comparison

We compare the actual runtime of our implementations of ImminentCollisions and SearchCollisions for worst-case inputs on a NVIDIA V100 GPU. Fig. 8 shows the results. As expected, SearchCollisions is exponentially faster when there are enough processors available.

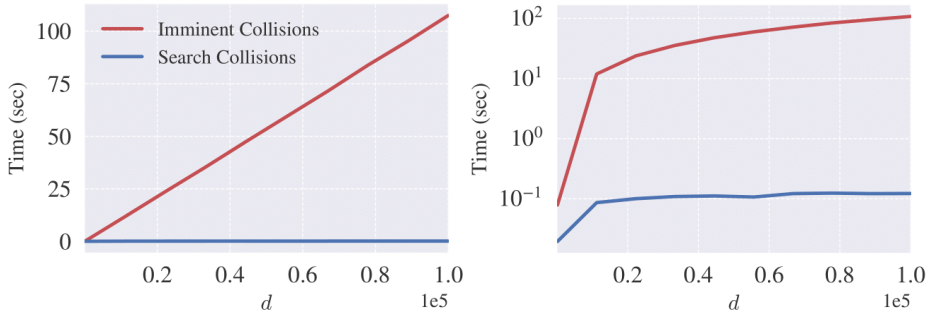

Figure 8: Worst-case runtimes of SearchCollisions vs. ImminentCollisions. **(left)** linear scale **(right)** log scale.