A Projection-Based Framework for Gradient-Free and Parallel Learning

Andreas Bergmeister
TU Munich, MCML

Manish Krishan Lal
TU Munich, MCML

Stefanie Jegelka
TU Munich, MCML
MIT CSAIL

Suvrit Sra
TU Munich, MCML
MIT LIDS

Abstract

We present a feasibility-seeking approach to neural network training. This mathematical optimization framework is distinct from conventional gradient-based loss minimization and uses projection operators and iterative projection algorithms. We reformulate training as a large-scale feasibility problem: finding network parameters and states that satisfy local constraints derived from its elementary operations. Training then involves projecting onto these constraints, a local operation that can be parallelized across the network. We introduce PJAX, a JAX-based software framework that enables this paradigm. PJAX composes projection operators for elementary operations, automatically deriving the solution operators for the feasibility problems (akin to autodiff for derivatives). It inherently supports GPU/TPU acceleration, provides a familiar NumPy-like API, and is extensible. We train diverse architectures (MLPs, CNNs, RNNs) on standard benchmarks using PJAX, demonstrating its functionality and generality. Our results show that this approach is a compelling alternative to gradient-based training, with clear advantages in parallelism and the ability to handle non-differentiable operations.

1 INTRODUCTION

Deep learning models have achieved remarkable success across diverse applications, largely driven by the effectiveness of gradient-based optimization. The backpropagation algorithm (Rumelhart et al., 1986), paired with stochastic gradient descent (SGD) and its adaptive variants (Duchi et al., 2011; Tieleman, 2012; Kingma and Ba, 2014), forms the bedrock of neural network training by efficiently computing loss gradients to iteratively adjust network parameters. Despite their undeniable success, gradient-based methods have several limitations. They often converge to local minima or high-error saddles (Dauphin et al., 2014; Choromanska et al., 2015), can suffer from vanishing or exploding gradients in deep architectures (Hochreiter, 1991; Bengio et al., 1994), and fundamentally require network components and loss functions to be (sub)differentiable. Moreover, the sequential nature of backpropagation limits parallelism and prolongs update latency. Finally, the global error backpropagation mechanism, requiring symmetric feedback pathways, is widely considered biologically implausible (Crick, 1989; Lillicrap et al., 2016). These challenges, along with inspiration from neuroscience and alternative optimization paradigms, motivate the search for fundamentally different approaches to training neural networks.

The research landscape includes various alternatives to end-to-end backpropagation. Zeroth-order methods such as Evolution Strategies (Salimans et al., 2017) or Genetic Algorithms (Holland, 1992) optimize the global loss using only function evaluations, at the cost of high sample complexity and poor scalability. Biologically inspired approaches use local rules but face distinct challenges: Hebbian methods (Hebb, 1949) lack supervised error integration, while spiking neural networks (Song et al., 2000; Gerstner and Kistler, 2002) contend with non-differentiable dynamics and credit assignment difficulties. Gradient approximation methods like Target Propagation (Lee et al., 2015), Feedback Alignment (Lillicrap et al., 2016) attempt to alleviate some backpropagation issues but often trade off convergence speed and performance. The recent *Forward–Forward* algorithm (Hinton, 2022) eliminates the backward pass altogether by training each layer on a local contrastive "goodness" objective; though its effectiveness on large-scale benchmarks has yet to be demonstrated.

This paper investigates a paradigm shift: reformulating

Proceedings of the 29th International Conference on Artificial Intelligence and Statistics (AISTATS) 2026, Tangier, Morocco. PMLR: Volume 300.

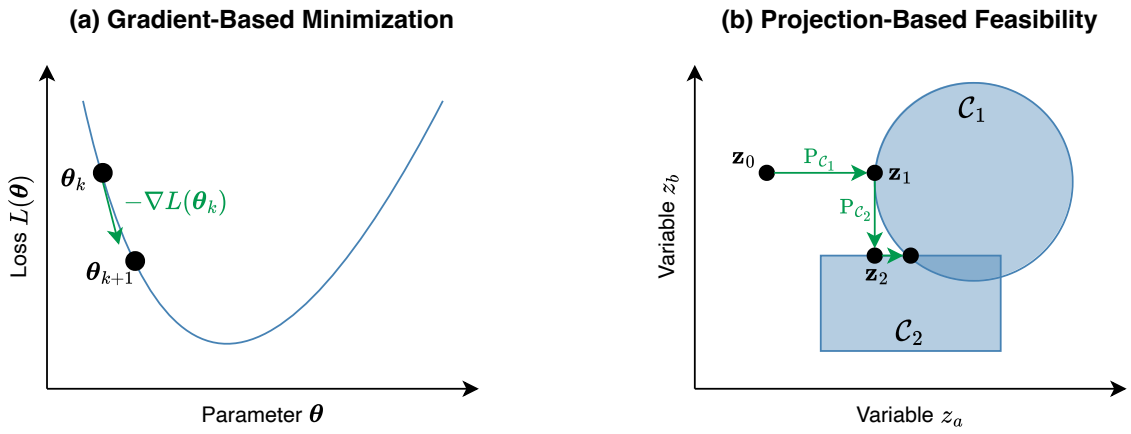

Figure 1: Neural network training paradigm shift. (a) Gradient-based methods iteratively minimize a loss function $L(\boldsymbol{\theta})$ using local gradients. (b) Our projection-based feasibility approach finds a point $\boldsymbol{z}$ in the intersection of constraint sets (e.g., $\mathcal{C}_1, \mathcal{C}_2$) via iterative projections onto these sets.

neural network training from a loss minimization task (Fig. 1a) into a large-scale *feasibility problem* (Fig. 1b). Instead of navigating a complex loss landscape, we seek to identify network parameters and intermediate states that simultaneously satisfy a collection of local constraints derived from the network's structure and the desired input-output mappings. Our approach achieves this through a fine-grained decomposition of the network into its elementary operations—termed *primitive functions* (e.g., inner products, pointwise activations). For these primitive functions, orthogonal projections onto their graphs (set of valid input–output pairs) are often computationally inexpensive. The availability of such efficient projections allows us to recast training as the problem of finding a point in the intersection of numerous local constraint sets, a task well-suited for iterative projection algorithms rooted in convex optimization (Bauschke and Combettes, 2011). This feasibility perspective builds upon conceptual work by Elser (2021).

This feasibility-driven paradigm offers several inherent advantages over traditional gradient-based methods. First, training relies on the availability of projection operators for primitive functions rather than on their differentiability, naturally accommodating non-differentiable components within network architectures. Second, the updates are local, modifying only adjacent variables in the computation graph. This eliminates the need for global error backpropagation and its associated weight transport problem, aligning more closely with notions of biological plausibility. Crucially, this locality, particularly when coupled with a bipartite structuring of the computation graph (Section 3), enables fully parallelizable updates across the network. While other strategies also decompose the global training objective into simpler, coupled sub-problems to enable parallel execution, such as layer-wise ADMM (Glowinski and Marroco, 1975; Boyd et al., 2011) or the Method of Auxiliary Coordinates (MAC) (Carreira-Perpiñán and Wang, 2014), they often face computational challenges such as large matrix inversions or intricate dual-variable bookkeeping (Taylor et al., 2016). In contrast, our fine-grained decomposition requires only a collection of efficiently computable local projections.

This paper delivers the rigorous formulation, a robust software framework, and systematic empirical validation necessary to establish projection-based training as a concrete, implementable, and explorable alternative. Our specific contributions are:

1. A detailed graph-structured *feasibility formulation* using edge variables. This formulation allows us to solve the problem efficiently with parallelizable projection algorithms (e.g., Alternating Projections and Douglas-Rachford) by exploiting a bipartitioning of the computation graph (Section 3).
2. A set of projection operators for key primitive functions (e.g., dot product, ReLU of sum, max pooling) that are fundamental building blocks for neural networks. The mathematical derivations for these operators are provided in Section C.1.
3. **PJAX** (Projection JAX)[1], a complete numerical framework built upon JAX (Bradbury et al., 2018). Designed for this compositional projection-based paradigm, it serves a role analogous to automatic differentiation systems for gradient-based methods. PJAX inherits JAX's GPU/TPU acceleration

[1]PJAX is available at https://github.com/AndreasBergmeister/pjax

and JIT compilation capabilities, provides a familiar API mirroring NumPy/JAX, automatically orchestrates the iterative solution of user-defined feasibility problems, and supports extension with new primitive functions and projection operators.

4. Extensive empirical validation across diverse neural network architectures (MLPs, CNNs, RNNs) on standard benchmarks (Section 5). These experiments demonstrate the viability of our projection-based approach and provide an initial characterization of its performance.

2 BACKGROUND AND PRELIMINARIES

We work in finite-dimensional Euclidean spaces $\mathbb{R}^d$. The inner product is $\langle \boldsymbol{x}, \boldsymbol{y} \rangle = \boldsymbol{x}^\top \boldsymbol{y}$, inducing the Euclidean norm $\|\boldsymbol{x}\| = \sqrt{\langle \boldsymbol{x}, \boldsymbol{x} \rangle}$, which in a product space is $\|(\boldsymbol{x}_1, \dots, \boldsymbol{x}_n)\|^2 = \sum_{i=1}^n \|\boldsymbol{x}_i\|^2$.

2.1 Projection and Proximal Operators

The concept of projection onto sets is central to our method. Given a non-empty closed set $\mathcal{C} \subseteq \mathbb{R}^d$, the *projection* of $\boldsymbol{x} \in \mathbb{R}^d$ onto $\mathcal{C}$ is

$$\mathrm{P}_{\mathcal{C}}(\boldsymbol{x}) = \arg\min_{\boldsymbol{y} \in \mathcal{C}} \|\boldsymbol{x} - \boldsymbol{y}\|^2. \tag{1}$$

This projection always exists. If $\mathcal{C}$ is convex, $\mathrm{P}_{\mathcal{C}}(\boldsymbol{x})$ is unique; the operator $\mathrm{P}_{\mathcal{C}}$ is non-expansive, and its fixed points constitute $\mathcal{C}$. If $\mathcal{C}$ is non-convex, the minimizer may not be unique, rendering $\mathrm{P}_{\mathcal{C}}$ set-valued; $\mathrm{P}_{\mathcal{C}}(\boldsymbol{x})$ then denotes an arbitrary choice from the set of minimizers.

Projections onto product sets $\mathcal{C} = \mathcal{C}_1 \times \cdots \times \mathcal{C}_m$ (where each $\mathcal{C}_i \subseteq \mathbb{R}^{d_i}$ is non-empty and closed) separate as follows: for $(\boldsymbol{x}_1, \dots, \boldsymbol{x}_m) \in \mathbb{R}^{d_1} \times \cdots \times \mathbb{R}^{d_m}$,

$$\mathrm{P}_{\mathcal{C}_1 \times \cdots \times \mathcal{C}_m}(\boldsymbol{x}_1, \dots, \boldsymbol{x}_m) = (\mathrm{P}_{\mathcal{C}_1}(\boldsymbol{x}_1), \dots, \mathrm{P}_{\mathcal{C}_m}(\boldsymbol{x}_m)). \tag{2}$$

The *proximal operator* is associated with functions. For a proper, lower semi-continuous function $f : \mathbb{R}^d \to (-\infty, +\infty]$ and $\lambda > 0$, the proximal operator of f at $\boldsymbol{x}_0 \in \mathbb{R}^d$ is

$$\mathrm{prox}_{\lambda f}(\boldsymbol{x}_0) = \arg\min_{\boldsymbol{x} \in \mathbb{R}^d} \left(f(\boldsymbol{x}) + \frac{1}{2\lambda} \|\boldsymbol{x} - \boldsymbol{x}_0\|^2 \right). \tag{3}$$

If f is convex, this minimizer is unique (Moreau, 1965). A point $\boldsymbol{x}$ is a minimizer of f if and only if it is a fixed point of the proximal operator, $\mathrm{prox}_{\lambda f}(\boldsymbol{x}) = \boldsymbol{x}$. The proximal operator generalizes the projection operator: if f is the indicator function of a closed convex set $\mathcal{C}$, then $\mathrm{prox}_{\lambda f}(\boldsymbol{x}_0) = \mathrm{P}_{\mathcal{C}}(\boldsymbol{x}_0)$.

2.2 Feasibility Problems and Projection Algorithms

Many problems involve finding a point in the intersection of multiple constraint sets. Given closed sets $\mathcal{C}_1, \dots, \mathcal{C}_N \subseteq \mathbb{R}^d$, the *feasibility problem* seeks $\boldsymbol{x} \in \mathbb{R}^d$ such that

$$\boldsymbol{x} \in \bigcap_{i=1}^N \mathcal{C}_i, \tag{4}$$

assuming a non-empty intersection. Iterative projection algorithms are apt for such problems, particularly when individual projections $\mathrm{P}_{\mathcal{C}_i}$ are computationally simpler than directly finding a point in the intersection.

Classical algorithms include **Alternating Projections (AP)** for two sets $\mathcal{C}_1, \mathcal{C}_2$, with the sequence:

$$\boldsymbol{x}_{k+1} = \mathrm{P}_{\mathcal{C}_1}(\mathrm{P}_{\mathcal{C}_2}(\boldsymbol{x}_k)). \tag{5}$$

Cyclic Projections (CP) extends this to $N > 2$ sets:

$$\boldsymbol{x}_{k+1} = \mathrm{P}_{\mathcal{C}_N}(\mathrm{P}_{\mathcal{C}_{N-1}}(\dots \mathrm{P}_{\mathcal{C}_1}(\boldsymbol{x}_k))). \tag{6}$$

Douglas-Rachford (DR) for two sets $\mathcal{C}_1, \mathcal{C}_2$ uses reflections $\mathrm{R}_{\mathcal{C}}(\boldsymbol{x}) = 2\mathrm{P}_{\mathcal{C}}(\boldsymbol{x}) - \boldsymbol{x}$:

$$\boldsymbol{x}_{k+1} = \tfrac{1}{2} \left(\boldsymbol{x}_k + \mathrm{R}_{\mathcal{C}_1}(\mathrm{R}_{\mathcal{C}_2}(\boldsymbol{x}_k)) \right). \tag{7}$$

When the sets $\mathcal{C}_i$ are convex with a non-empty intersection, AP (Bregman, 1965) and CP (Gubin et al., 1967) converge to a point in this intersection. For DR, under similar conditions, the sequence of projections (e.g., $\{\mathrm{P}_{\mathcal{C}_2}(\boldsymbol{x}_k)\}$) converges to such a point (Douglas and Rachford, 1956; Lions and Mercier, 1979). These algorithms form the basis for solving feasibility problems in our work.

3 METHOD

Consider a supervised learning setting with dataset $\mathcal{D} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^N$, where $\boldsymbol{x}_i \in \mathbb{R}^{d_{\text{in}}}$ and $\boldsymbol{y}_i \in \mathbb{R}^{d_{\text{target}}}$. Let $\boldsymbol{f} : \mathbb{R}^{d_{\text{in}}} \times \mathbb{R}^{d_\theta} \to \mathbb{R}^{d_{\text{out}}}$ be a parametric function (e.g., a neural network), and let $\ell : \mathbb{R}^{d_{\text{out}}} \times \mathbb{R}^{d_{\text{target}}} \to \mathbb{R}$ be a loss function. The conventional approach minimizes the empirical risk

$$\min_{\boldsymbol{\theta} \in \mathbb{R}^{d_\theta}} \frac{1}{N} \sum_{i=1}^N \ell(\boldsymbol{f}(\boldsymbol{x}_i, \boldsymbol{\theta}), \boldsymbol{y}_i). \tag{8}$$

We reformulate training as a *feasibility problem* by translating the per-sample objective of minimizing the loss into a set of hard constraints. The architecture of $\boldsymbol{f}$, viewed as a composition of elementary operations, provides further constraints: each operation's output must align with its inputs according to its defining function. Training then becomes the search for

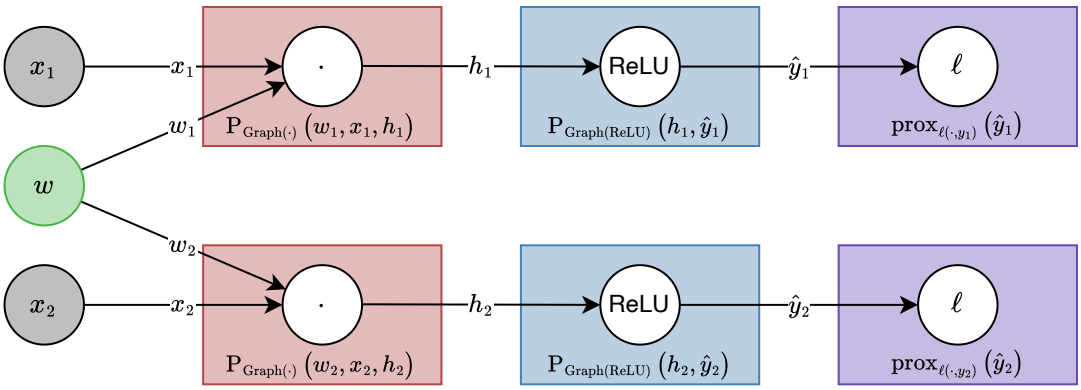

Figure 2: Computation graph for $\ell(\mathrm{ReLU}(w \cdot x_i), y_i)$ on two samples, showing projection operators for hidden function and loss nodes.

a state (network parameters and internal activations) that simultaneously satisfies this entire collection of local constraints. We formally construct this feasibility problem by defining these constraints over variables within a computation graph.

3.1 Constraint formulation via computation graph

For a given input $\boldsymbol{x}$, parameters $\boldsymbol{\theta}$, and target $\boldsymbol{y}$, we represent the computation of $\ell(\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{\theta}), \boldsymbol{y})$ as a directed acyclic graph (DAG) $G = (\mathcal{V}, \mathcal{E})$, the *computation graph*. Nodes represent constant inputs, parameters, elementary operations of $\boldsymbol{f}$ (primitive scalar functions), and the loss function. Edges represent data flow: an edge $(u, v) \in \mathcal{E}$ indicates that the value from node u serves as an input to the operation at node v. We denote the set of parent nodes of v as $\mathcal{N}^-(v)$ and its children as $\mathcal{N}^+(v)$. Crucially, we decompose $\boldsymbol{f}$ into its scalar component functions (e.g., addition, inner products, ReLU activations); the nodes in the graph that correspond to operations therefore represent these scalar functions. Consequently, only scalar values pass along the edges. This granularity allows us to derive tractable projection operators onto the constraint sets associated with these primitive functions.

We introduce a variable $z_{uv} \in \mathbb{R}$ for each edge $(u, v) \in \mathcal{E}$, representing the value carried along that edge. The state vector $\boldsymbol{z} = (z_{uv})_{(u,v)\in\mathcal{E}} \in \mathbb{R}^{\mathcal{E}}$ collects all edge variables. For each node in $\mathcal{V}$, we define a constraint set in $\mathbb{R}^{\mathcal{E}}$. For a single sample $(\boldsymbol{x}, \boldsymbol{y})$ (where $\boldsymbol{x} = (x_1, \ldots, x_{d_{\text{in}}})$ and $\boldsymbol{y} = (y_1, \ldots, y_{d_{\text{target}}})$), we categorize these constraints by node type as follows:

Constant input nodes $c_j \in \mathcal{V}$ correspond to each input component x_j for $j \in [d_{\text{in}}]$. The constraints

$$\mathcal{C}_{c_j} = \{\boldsymbol{z} \in \mathbb{R}^{\mathcal{E}} \mid \forall w \in \mathcal{N}^+(c_j) : z_{c_j w} = x_j\}$$

enforce that edges outgoing from the respective node carry its value. The projection operator $\mathrm{P}_{\mathcal{C}_{c_j}}(\boldsymbol{z})$ sets $z_{c_j w}$ to x_j.

Parameter nodes $p_k \in \mathcal{V}$ correspond to each parameter component θ_k for $k \in [d_\theta]$. The constraints

$$\mathcal{C}_{p_k} = \{\boldsymbol{z} \in \mathbb{R}^{\mathcal{E}} \mid \forall u, w \in \mathcal{N}^+(p_k) : z_{p_k w} = z_{p_k u}\}$$

enforce consensus among all outgoing edge values from a parameter node. The projection operator $\mathrm{P}_{\mathcal{C}_{p_k}}(\boldsymbol{z})$ sets the values of outgoing edges $z_{p_k w}$ to the average of the current values $\{z_{p_k w} \mid w \in \mathcal{N}^+(p_k)\}$.

Target node $t \in \mathcal{V}$ receives the network's output and ensures minimal loss with respect to the target $\boldsymbol{y}$. Depending on the loss function ℓ, we either project onto the constraint set

$$\mathcal{C}_t = \{\boldsymbol{z} \in \mathbb{R}^{\mathcal{E}} \mid (z_{ut})_{u\in\mathcal{N}^-(t)} \in \arg\min_{\boldsymbol{y}'} \ell(\boldsymbol{y}', \boldsymbol{y})\}$$

or apply the proximal operator of the loss function (with $\lambda > 0$) to the current predicted outputs

$$(z'_{ut})_{u\in\mathcal{N}^-(t)} \leftarrow \mathrm{prox}_{\lambda\ell(\cdot,\boldsymbol{y})}\big((z_{ut})_{u\in\mathcal{N}^-(t)}\big).$$

See Section C.2 for details. Note that a fixed point of the proximal operator is a minimizer of the loss function (see Section 2.1), so it satisfies the constraint $\mathcal{C}_t$.

Hidden function nodes $h \in \mathcal{V}$ represent the application of a primitive function f_h to their inputs. The constraints

$$\mathcal{C}_h = \{\boldsymbol{z} \in \mathbb{R}^{\mathcal{E}} \mid \forall w \in \mathcal{N}^+(h) : z_{hw} = f_h\big((z_{uh})_{u\in\mathcal{N}^-(h)}\big)\}$$

enforce that all outgoing edge values equal the result of applying f_h to its inputs. Projecting onto $\mathcal{C}_h$ involves: (1) computing an average of current values on outgoing edges, $\bar{z} = (\sum_{w\in\mathcal{N}^+(h)} z_{hw})/|\mathcal{N}^+(h)|$; (2) projecting the incoming edge values $(z_{u'h})_{u'\in\mathcal{N}^-(h)}$ and

the average $\bar{z}$ onto the graph of f_h: $(z'_{u'h}, z'_{hw}) = \mathrm{P}_{\mathrm{Graph}(f_h)}((z_{u'h})_{u' \in \mathcal{N}^-(h)}, \bar{z})$; (3) set incoming edge values $z_{u'h} \leftarrow z'_{u'h}$ and outgoing edge values $z_{hw} \leftarrow z'_{hw}$ for all $w \in \mathcal{N}^+(h)$. Theorem 3 formally justifies these steps.

The overall feasibility problem involves finding a state vector $\boldsymbol{z}$ that lies in the intersection of all such individual node constraints

$$\text{Find } \boldsymbol{z} \in \bigcap_{v \in \mathcal{V}} \mathcal{C}_v. \tag{9}$$

The formulation extends to batches of samples by constructing a single, larger computation graph with N instances of data-dependent components (input nodes, function nodes, and loss nodes for each sample i), while sharing parameter nodes across instances. Conceptually, we construct a computation graph for the function

$$\begin{aligned}&((\boldsymbol{x}_1, \boldsymbol{y}_1), \ldots, (\boldsymbol{x}_N, \boldsymbol{y}_N), \boldsymbol{\theta}) \mapsto \\ &\quad (\ell(\boldsymbol{f}(\boldsymbol{x}_1, \boldsymbol{\theta}), \boldsymbol{y}_1), \ldots, \ell(\boldsymbol{f}(\boldsymbol{x}_N, \boldsymbol{\theta}), \boldsymbol{y}_N)),\end{aligned} \tag{10}$$

as Fig. 2 illustrates. The following theorem justifies the correctness of our formulation, showing that if a solution to the feasibility problem exists, it yields a set of parameters that minimizes the empirical risk in Eq. (8).

Theorem 1 (Optimality of feasibility solution)**.** *Let $\boldsymbol{z}^* \in \mathbb{R}^{\mathcal{E}}$ be a solution to the feasibility problem in Eq. (9) for the function $\ell(\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{\theta}), \boldsymbol{y})$ applied to a dataset $\mathcal{D}$ as described above. Then the consensus values of the outgoing edges from the parameter nodes p_k ($k \in [d_\theta]$) in $\boldsymbol{z}^*$ minimize the empirical risk in Eq. (8).*

Proof. A feasible state vector $\boldsymbol{z}^*$ satisfies all node constraints. Specifically, satisfying the target node constraints means that the outputs of the network, $\hat{\boldsymbol{y}}_i^*$, minimize the per-sample loss. Parameter node constraints ensure their outgoing edges yield common consensus values, θ_k^*, which define the overall network parameters $\boldsymbol{\theta}^* \in \mathbb{R}^{d_\theta}$. The input and hidden function node constraints then ensure that these loss-minimizing values $\hat{\boldsymbol{y}}_i^*$ are precisely the outputs of the network function $\boldsymbol{f}(\boldsymbol{x}_i, \boldsymbol{\theta}^*)$. Consequently, $\boldsymbol{\theta}^*$ minimizes the empirical risk in Eq. (8). □

We solve the feasibility problem in Eq. (9) with the iterative projection algorithms described in Section 2.2. When the intersection is non-empty and the sets $\mathcal{C}_v$ are convex, these algorithms converge to a feasible point under standard assumptions. In the nonconvex setting, they should be viewed as heuristic iterative methods that often find useful approximate solutions in practice, but without comparable general convergence guarantees. To efficiently compute the projections in parallel across the computation graph, we leverage the structure of the graph, specifically its bipartite nature, as the following theorem details. Notably, neural network computation graphs are often bipartite; if not, they can be made bipartite by inserting identity operations (dummy nodes with $f_v(z) = z$).

Theorem 2 (Parallelizable projections via bipartition)**.** *Let $G = (\mathcal{V}, \mathcal{E})$ be a computation graph with a bipartition $\mathcal{V} = \mathcal{A} \cup \mathcal{B}$, where $\mathcal{A}$ and $\mathcal{B}$ are disjoint sets of nodes. The constraints for each node $v \in \mathcal{V}$ are defined as above. Then, the feasibility problem Eq. (9) is equivalent to the two-set feasibility problem*

$$\textit{Find} \quad \boldsymbol{z} \in \mathcal{C}_{\mathcal{A}} \cap \mathcal{C}_{\mathcal{B}}, \tag{11}$$

$$\textit{where} \quad \mathcal{C}_{\mathcal{A}} = \bigcap_{v \in \mathcal{A}} \mathcal{C}_v \ \textit{and} \ \mathcal{C}_{\mathcal{B}} = \bigcap_{v \in \mathcal{B}} \mathcal{C}_v. \tag{12}$$

Furthermore, the projection $\mathrm{P}_{\mathcal{C}_{\mathcal{A}}}(\boldsymbol{z})$ (and analogously $\mathrm{P}_{\mathcal{C}_{\mathcal{B}}}(\boldsymbol{z})$) can be computed by independently (and in parallel) applying the projection operators $\mathrm{P}_{\mathcal{C}_v}(\boldsymbol{z})$ for all $v \in \mathcal{A}$ (or $v \in \mathcal{B}$).

Proof. The equivalence $\bigcap_{v \in \mathcal{V}} \mathcal{C}_v = \mathcal{C}_{\mathcal{A}} \cap \mathcal{C}_{\mathcal{B}}$ is definitional. Consider distinct nodes $u, w \in \mathcal{A}$. By the bipartition, they are not adjacent. Since $\mathrm{P}_{\mathcal{C}_u}$ only modifies edge variables incident to u (similarly for w), and u, w share no incident edges, these projections act on disjoint sets of coordinates in $\boldsymbol{z}$. Therefore, all operators $\{\mathrm{P}_{\mathcal{C}_v}\}_{v \in \mathcal{A}}$ modify mutually disjoint components of $\boldsymbol{z}$. By Eq. (2) (projections onto product sets), $\mathrm{P}_{\mathcal{C}_{\mathcal{A}}}(\boldsymbol{z})$ is then computed by applying these individual projections independently, enabling parallel execution. An analogous argument for $\mathrm{P}_{\mathcal{C}_{\mathcal{B}}}(\boldsymbol{z})$ reduces the original problem to a two-set feasibility problem with parallelizable projection steps. □

Algorithm 1 summarizes the complete projection-based training procedure with batch processing.

4 COMPLEXITY ANALYSIS

Having detailed the projection-based training methodology, we now analyze its computational and memory complexity. All considered projection algorithms perform a single projection per step onto the constraints $\mathcal{C}_v$ for each node $v \in \mathcal{V}$. We focus on the complexity of these projections, as they dominate the overall computational cost.

Algorithm 1 Projection-based training

Input: Model $\boldsymbol{f}$, Loss function ℓ, Initial parameters $\boldsymbol{\theta}_0$, Dataset $\mathcal{D}$, Batch size B, Projection steps per batch K, Projection method `ProjMethod`.
Output: Optimized parameters $\boldsymbol{\theta}$.

$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_0$
while not converged **do**
  $((\boldsymbol{x}_1, \boldsymbol{y}_1), \ldots, (\boldsymbol{x}_B, \boldsymbol{y}_B)) \sim \mathcal{D}$
  $G \leftarrow$ computation graph for batch loss (10)
  $\boldsymbol{z} \leftarrow$ initial edge state vector $\in \mathbb{R}^{\mathcal{E}}$
  for $k = 1$ to K **do**
    $\boldsymbol{z} \leftarrow \texttt{ProjMethod}(G, \boldsymbol{z}, \{\mathrm{P}_{\mathcal{C}_v}\}_{v \in \mathcal{V}})$
  end for
  $\boldsymbol{\theta} \leftarrow$ extract parameters from $\boldsymbol{z}$
end while
return $\boldsymbol{\theta}$

Computational complexity per step Projecting onto a single node's constraint $\mathcal{C}_v$ typically has a cost comparable to the forward evaluation of the primitive function f_v. For example, the projection for the dot product, which implements neuron pre-activation, involves solving a scalar quintic equation (e.g., via 5 Newton steps) followed by rescaling inputs/outputs. Similarly, the projection for ReLU computes two candidate solutions in closed form and selects the one minimizing distance. See Section C.1 for details on these and other primitive functions. Therefore, the total cost per step of the projection method is roughly proportional to the cost of executing all primitive operations in the computation graph, similar to a forward pass. However, unlike the sequential forward and backward passes that backpropagation requires, the projection updates within each partition ($\mathcal{A}$ or $\mathcal{B}$) are fully parallelizable.

Memory complexity Storing the state vector $\boldsymbol{z} \in \mathbb{R}^{\mathcal{E}}$ leads to $O(|\mathcal{E}|)$ memory complexity. In contrast, standard backpropagation only requires storing a primitive function's output and its gradient, resulting in $O(|\mathcal{V}|)$ complexity. Since many networks have significantly more edges $|\mathcal{E}|$ than nodes $|\mathcal{V}|$, our method generally requires more memory. This requirement is pronounced in architectures with extensive weight sharing. For instance, during **batch processing**, processing N samples requires replicating the state associated with shared parameters N times in $\boldsymbol{z}$, as each sample interacts with the parameters via distinct edges in the expanded graph. Similarly, in **sequence models (e.g., RNNs)**, unrolling over T time steps necessitates storing distinct edge states for shared parameters at each step. **Convolutional networks (CNNs)** also exhibit this effect: applying a convolutional kernel at multiple spatial locations means each application corresponds to graph connections that require separate edge states. Our experiments made this evident: the high memory demands of the 4-layer CNN necessitated reducing the number of hidden units per layer to 16 to fit within the available 96GB GPU memory. In contrast, architectures with less parameter sharing, like MLPs, could accommodate larger sizes. Gradient-based methods are more memory-efficient in these cases because they aggregate gradients for shared parameters (e.g., by summation) and only need to store one copy of the parameters and their accumulated gradients.

In summary, the projection-based approach trades potentially higher memory usage, especially with shared parameters, for significant gains in parallelizability compared to gradient-based methods. The overall convergence rate (number of steps) depends on the specific problem and projection algorithm used. However, as our experiments empirically demonstrate (Section 5), projection-based training can achieve convergence rates that are competitive with first-order SGD methods on several tasks.

5 EXPERIMENTS

This section presents an empirical evaluation of our projection-based training method, outlined in Algorithm 1. We compare three projection methods (`ProjMethod`), Alternating Projections (AP), Douglas-Rachford (DR), and Cyclic Projections (CP), against standard gradient-based optimizers, Stochastic Gradient Descent (SGD) and Adam (Kingma and Ba, 2014), and non-backpropagation baselines, Feedback Alignment (FA) (Lillicrap et al., 2016) on MLPs and Forward–Forward (FF) (Hinton, 2022) on MLPs and CNNs. For AP and DR, we utilize the two-set feasibility problem derived from the bipartite graph formulation (Theorem 2). For CP, we apply the multi-set feasibility problem formulation directly, projecting cyclically onto node constraints and merging constraints within the same layer for efficiency (e.g., performing all ReLU projections in parallel). The projection order for CP follows a backward breadth-first search (BFS) from the target node.

5.1 Datasets and tasks

We evaluate our approach on diverse standard machine learning tasks. These include image classification using **MNIST** (28×28 grayscale images, 10 classes) and **CIFAR-10** (32×32 color images, 10 classes), binary classification on the **HIGGS** dataset (distinguishing signal from background noise in particle physics), and character-level language modeling with the **Shakespeare** dataset. We use the default train/test splits (approximately 80% train, 20% test) and reserve 10%

of training data for validation (hyperparameter tuning and early stopping). Across all tasks, we employ the standard cross-entropy loss and report accuracy as the primary evaluation metric. For the sequence modeling task, accuracy specifically refers to the fraction of correctly predicted next characters.

5.2 Neural network architectures

We test three representative architectures: A **Multilayer Perceptron (MLP)** with fully connected layers (linear + bias) and ReLU activations. A **Convolutional Neural Network (CNN)** with 3×3 convolutional layers (stride 1, padding 1) and ReLU activations; the output feature maps are spatially max-pooled into vectors before passing through a final linear layer. We implement a **Recurrent Neural Network (RNN)** with an MLP cell that processes the concatenation of a learned character embedding and the previous hidden state ($\boldsymbol{h}_{t-1}$) at each step t. The MLP outputs both logits for the next character and a raw hidden state ($\boldsymbol{h}'_t$). Activating this state via $\boldsymbol{h}_t = \mathrm{ReLU}(\boldsymbol{h}'_t)$ produces the input state for step $t+1$.

For each architecture, we evaluate both shallow (1 hidden layer) and deep (4 hidden layers) variants. For deep models, we also test a skip-connection design tailored to projection-based training: we concatenate the post-activation outputs of all hidden layers and feed the resulting vector to the final linear readout. This shortens the path from early layers to the output, which is important because projection-based training relies on local projection operators that only update adjacent variables in the computation graph. Without additional shortcuts, information from early layers reaches the output only after many iterations. Crucially, the design preserves the bipartite structure of the computation graph and thus our parallel projection scheme. It differs from the usual ResNet-style residual connection, which replaces a module $f(x)$ with $x + f(x)$ and does not limit the distance to the output as effectively.

Implementation and hyperparameters We implement projection-based methods (AP, DR, CP) using our PJAX framework (Section B) and all baselines with JAX and Flax (Heek et al., 2024), ensuring fair comparison, as PJAX uses JAX as its backend. Notably, defining models in PJAX's `pjax.nn` API closely mirrors standard JAX/Flax usage. For the output layer constraint, we use the proximal operator for cross-entropy (Theorem 11, with $\lambda = 5$), finding it more robust than margin constraints (Theorem 10), which require careful tuning of the margin parameter.

FA and FF are implemented in line with their original formulations (Lillicrap et al., 2016; Hinton, 2022). Neither method is applied to skip-connection architectures, which are not part of their original design. For FF, we report results with both SGD and Adam optimizers, since performance varies substantially with the choice of optimizer.

We use consistent parameters across experiments: learning rate 10^{-3} for SGD, Adam and FF, 10^{-4} for FA, batch size 256 for all methods, and $K = 50$ projection steps per batch for AP/DR/CP. Each projection step counts as one training step, so projection methods process 50 times fewer batches than baseline methods per reported step. Accordingly, the horizontal axes in Figs. 3 and 4 compare optimization steps rather than equal amounts of data processed.

5.3 Results

We present an overview of our experimental findings here. Section D provides detailed numerical results, including final test accuracies, convergence steps, and timings of all conducted experiments on a single NVIDIA H100 GPU with 96GB of memory.

Overall trends. Figure 3 displays representative test accuracy curves from our experiments, highlighting a general trend: while Adam consistently achieves the highest test accuracy and generally converges fastest, projection-based methods prove viable across tasks and architectures. Among the projection algorithms, DR often outperforms AP on MLPs and RNNs in accuracy, while AP is competitive on CNNs. CP yields almost identical accuracy to AP but is slower due to its sequential nature (hence we omit CP from plots for clarity; see Section D for results). We also find that AP/CP tend to exhibit higher run-to-run variance than DR, likely due to greater sensitivity to initialization. DR's reflection (overshoot) step enlarges the effective basin of attraction and reduces stalling in shallow basins, leading to more consistent outcomes across seeds. The specific characteristics of projection-based training, such as convergence to optimal solutions and step efficiency, vary with network architecture. For **MLPs** (Table 2), projection methods, particularly DR in shallow cases, approach SGD's accuracy with notable computational efficiency per step (often $\sim 10\times$ faster). For **CNNs** (Table 3), the accuracy gap relative to gradient-based methods widens, and step speed decreases due to higher memory requirements for shared parameters (see Section 4 for theoretical details). For **RNNs** (Table 4), projection methods possess a structural advantage, as no backpropagation through time is performed, thereby sidestepping issues with vanishing or exploding gradients (see Fig. 6). This results in significantly faster convergence in terms of training steps compared to SGD, though Adam still achieves the best overall results, managing training dynamics effectively.

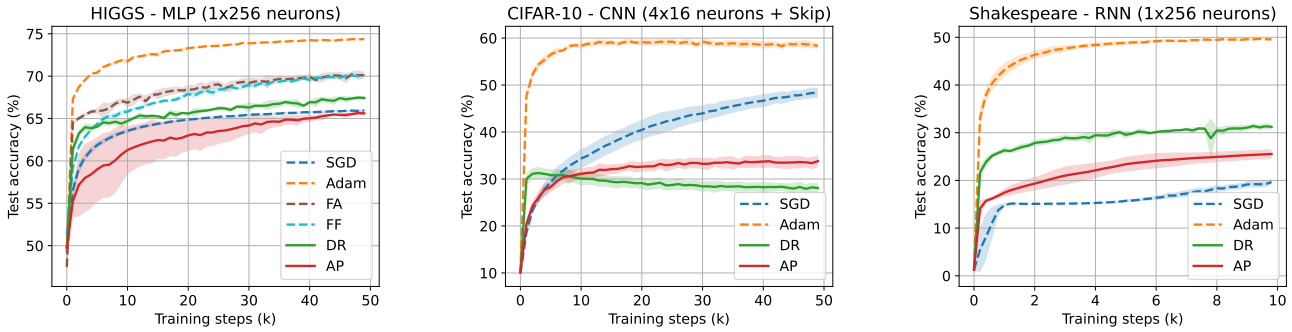

Figure 3: Test accuracy vs. training steps.

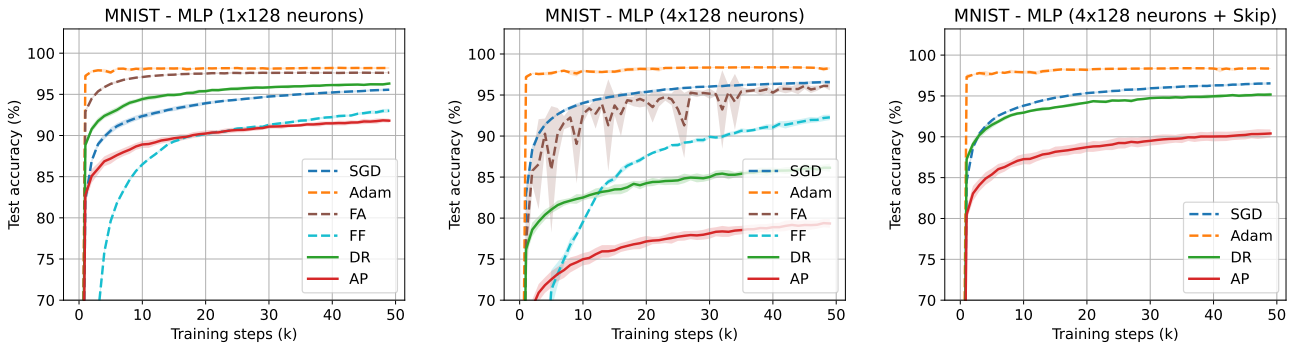

Figure 4: Impact of network depth and skip connections.

The memory demands for shared parameters in RNNs also impact step times, similar to CNNs.

FF and FA baselines. Our experiments do not reveal a consistent ranking between projection-based methods and FF/FA baselines for all tasks. On CIFAR-10 CNN, for example, the 4-layer projection-based model with skip connections outperforms both FF (even when trained with Adam) and FA (implemented without skips, as in prior work). In other settings, FF or FA achieve better performance. A clear pattern is that FF is highly optimizer-dependent: it performs well with Adam but degrades sharply with SGD. Figures 3 and 4 report FF results with Adam, while additional SGD results are provided in Section D.

Depth and skip connections. A key consideration for projection-based methods, which rely on local updates, is their effectiveness in deeper networks. Figure 4 explores this using an MLP trained on MNIST with the Douglas-Rachford (DR) algorithm. While shallow MLPs (left panel) train readily, the performance of a deep MLP without skip connections (center panel) degrades, highlighting challenges in propagating information through many layers via local projections alone. The introduction of skip connections (right panel) substantially improves training for the deep MLP. These connections provide shorter paths for information flow,

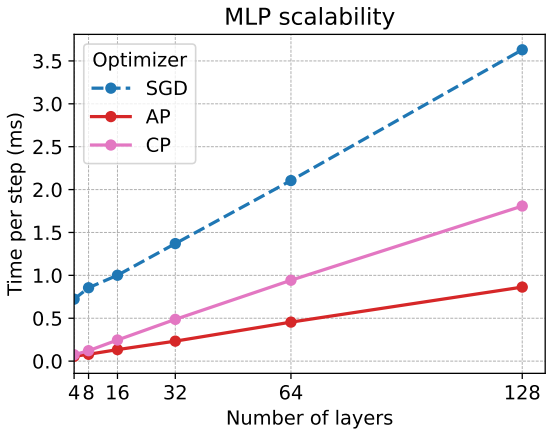

Figure 5: Scalability of projection vs. gradient-based methods with network depth.

proving crucial for effective learning in deeper architectures using projection methods, while maintaining the bipartite graph structure essential for parallelization.

Figure 5 further analyzes the computational efficiency of projection methods. It compares step times of SGD, CP, and AP for an MLP (16 hidden units per layer) on a single MNIST sample, across varying network depths. These algorithms serve as direct proxies for the gradient-based, sequential projection, and parallel

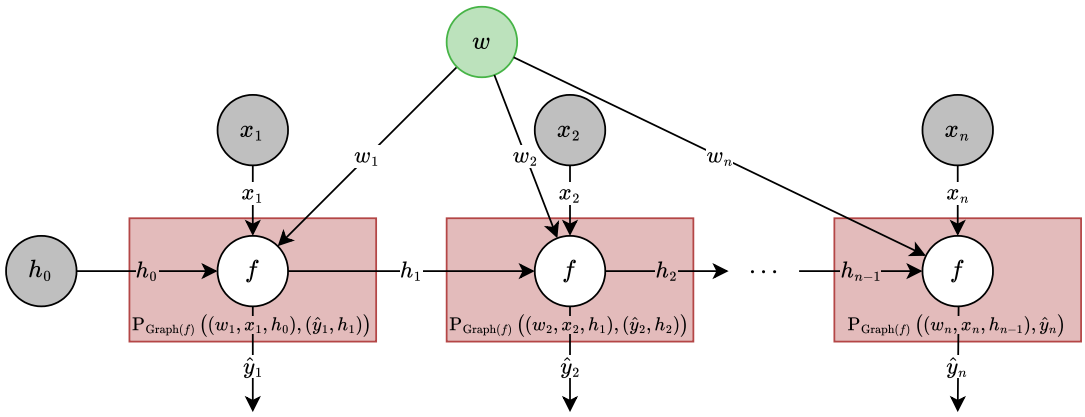

Figure 6: Computation graph for RNN: each unrolled cell f has a local parameter copy (w_t). Projections are performed separately (possibly concurrently) for each time step. Parameter consensus is enforced through the parameter node.

projection paradigms, respectively, avoiding the complexities of other optimizers like Adam or more involved projection schemes like DR. The analysis reveals two main trends. First, both CP and AP achieve lower step times than SGD, likely due to their intensive data reuse ($K = 50$ updates per sample). Second, AP's step time increases less steeply with network depth compared to SGD and CP. This favorable scaling stems from the parallelization of updates across all layers via the bipartite graph structure (Theorem 2), a concurrency efficiently harnessed by modern hardware like GPUs, in contrast to the sequential processing inherent in SGD (backpropagation) and CP (cyclic projections). Consequently, AP's parallel advantage becomes more pronounced for deeper networks.

6 DISCUSSION

We reframe neural network training as a feasibility problem and solve it with iterative projections. Because the method requires projections rather than derivatives, it naturally supports non-differentiable components such as quantization (Theorem 9) and logical constraints (e.g., the margin loss in Theorem 10). Updates are local to adjacent variables in the computation graph, which removes global error backpropagation, enables parallelization across network components, and aligns with biological learning principles. The PJAX framework (Section B) composes projection operators analogously to how autodiff composes derivatives via the chain rule, providing a practical tool for investigating and extending this paradigm. To the best of our knowledge, it is the only gradient-free training approach accompanied by a general-purpose framework rather than a single-task implementation.

Beyond these advantages, our formulation keeps the standard vector-to-vector interface with single-pass inference. In contrast, *Forward–Forward* (Hinton, 2022) learns a scalar "goodness" for each (input, label) pair, which implies a per-class forward pass at inference and presents challenges for scaling to large output spaces (as in language modeling) or extending to continuous targets.

Empirically, projection methods train diverse architectures (MLPs, CNNs, and RNNs) reliably. They achieve competitive step times and benefit from parallel updates; RNNs particularly benefit from avoiding backpropagation through time. Douglas–Rachford is often the most stable, though performance still falls short of highly optimized adaptive gradient methods such as Adam. For both gradient- and projection-based optimizers, the train-test gap is often modest (Section D), suggesting that the gap reflects limits in fitting the training data rather than a fundamental lack of generalization.

The main limitation is memory. The algorithm maintains distinct edge variables for each interaction between parameters and data, so requirements scale with batch size, sequence length, and the number of convolutional locations.

Promising directions include: (i) improved projection dynamics (adaptive damping/relaxation, preconditioning, acceleration, learned step sizes); (ii) hybrid schemes that interleave projections with occasional gradient steps; (iii) tailored architectures that shorten paths to the output and use structured sparsity (e.g., Mixture-of-Experts) to limit parameter–data interaction; (iv) memory-saving techniques (low-rank or quantized edge states); and (v) refined analyses of convergence and generalization for nonconvex, possibly set-valued projections. We hope PJAX lowers the barrier to exploring these directions.

ACKNOWLEDGMENTS

This project was funded by the Alexander von Humboldt Foundation.

References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

Heinz H. Bauschke and Patrick L. Combettes. *Convex Analysis and Monotone Operator Theory in Hilbert Spaces*. CMS Books in Mathematics. Springer, New York, NY, 2011. ISBN 978-1-4419-9466-0. doi: 10.1007/978-1-4419-9467-7.

Heinz H Bauschke, Manish Krishan Lal, and Xianfu Wang. Projecting onto rectangular hyperbolic paraboloids in hilbert space. *arXiv preprint arXiv:2206.04878*, 2022.

Yoshua Bengio. How auto-encoders could provide credit assignment in deep networks via target propagation. *arXiv preprint arXiv:1407.7906*, 2014.

Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

Dimitri P Bertsekas. Nonlinear programming. *Journal of the Operational Research Society*, 48(3):334–334, 1997.

Elie L Bienenstock, Leon N Cooper, and Paul W Munro. Theory for the development of neuron selectivity: orientation specificity and binocular interaction in visual cortex. *Journal of Neuroscience*, 2(1):32–48, 1982.

Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, Jonathan Eckstein, et al. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine learning*, 3(1):1–122, 2011.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/jax-ml/jax.

Lev M Bregman. The method of successive projections for finding a common point of convex sets. In *Soviet Math. Dokl.*, volume 6, pages 688–692, 1965.

Miguel Á. Carreira-Perpiñán and Weiran Wang. Distributed optimization of deeply nested systems. In *Artificial Intelligence and Statistics (AISTATS)*, 2014.

Anna Choromanska, Mikael Henaff, Michael Mathieu, Gérard Ben Arous, and Yann LeCun. The loss surfaces of multilayer networks. In *Proceedings of the 18th International Conference on Artificial Intelligence and Statistics (AISTATS 2015)*, pages 192–204, 2015.

Francis Crick. The recent excitement about neural networks. *Nature*, 337(6203):129–132, 1989.

Yann N. Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in Neural Information Processing Systems 27 (NeurIPS 2014)*, pages 2933–2941, 2014.

Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, et al. Loihi: A neuromorphic manycore processor with on-chip learning. *Ieee Micro*, 38(1):82–99, 2018.

Jim Douglas and Henry H Rachford. On the numerical solution of heat conduction problems in two and three space variables. *Transactions of the American mathematical Society*, 82(2):421–439, 1956.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

Veit Elser. Phase retrieval by iterated projections. *Journal of the Optical Society of America A*, 20(1):40–55, 2003.

Veit Elser. Learning without loss. *Fixed Point Theory and Algorithms for Sciences and Engineering*, 2021 (1):12, 2021.

Wulfram Gerstner and Werner M Kistler. *Spiking neuron models: Single neurons, populations, plasticity*. Cambridge university press, 2002.

Roland Glowinski and Americo Marroco. Sur l'approximation, par éléments finis d'ordre un, et la résolution, par pénalisation-dualité d'une classe de problèmes de dirichlet non linéaires. *Revue française d'automatique, informatique, recherche opérationnelle. Analyse numérique*, 9(R2):41–76, 1975.

LG Gubin, Boris T Polyak, and EV Raik. The method of projections for finding the common point of convex sets. *USSR Computational Mathematics and Mathematical Physics*, 7(6):1–24, 1967.

Donald Olding Hebb. *The organization of behavior: A neuropsychological theory*. Wiley, 1949.

Jonathan Heek, Anselm Levskaya, Avital Oliver, Marvin Ritter, Bertrand Rondepierre, Andreas Steiner, and Marc van Zee. Flax: A neural network library and ecosystem for JAX, 2024. URL http://github.com/google/flax.

Geoffrey Hinton. The forward-forward algorithm: Some preliminary investigations. *arXiv preprint arXiv:2212.13345*, 2022.

Geoffrey E Hinton. Training products of experts by minimizing contrastive divergence. *Neural computation*, 14(8):1771–1800, 2002.

Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. *Diploma, Technische Universität München*, 91(1):31, 1991.

John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79 (8):2554–2558, 1982.

Max Jaderberg, Wojciech M Czarnecki, Simon Osindero, Oriol Vinyals, Alex Graves, Koray Kavukcuoglu, and David Silver. Decoupled neural interfaces using synthetic gradients. In *International Conference on Machine Learning*, pages 1627–1635. PMLR, 2017.

James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95-international conference on neural networks*, volume 4, pages 1942–1948. ieee, 1995.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Vijay Anand Korthikanti, Jared Casper, Sang Lym, Mandy McAfee, Michael Subramanian, Sophia Li, Sandeep Kotha, William Hoover, Ganesh Ananthanarayanan, Pradeep Dubey, et al. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5:341–353, 2023.

Dong-Hyun Lee, Saizheng Zhang, Asja Fischer, and Yoshua Bengio. Difference target propagation. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2015, Porto, Portugal, September 7-11, 2015, Proceedings, Part I 15*, pages 498–515. Springer, 2015.

Timothy P Lillicrap, Daniel Cownden, Douglas B Tweed, and Colin J Akerman. Random synaptic feedback weights support error backpropagation for deep learning. *Nature communications*, 7(1):13276, 2016.

Pierre-Louis Lions and Bertrand Mercier. Splitting algorithms for the sum of two nonlinear operators. *SIAM Journal on Numerical Analysis*, 16(6):964–979, 1979.

Paul A Merolla, John V Arthur, Rodrigo Alvarez-Icaza, Andrew S Cassidy, Jun Sawada, Filipp Akopyan, Bryan L Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, et al. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, 2014.

Jean-Jacques Moreau. Proximité et dualité dans un espace hilbertien. *Bulletin de la Société mathématique de France*, 93:273–299, 1965.

Yurii Nesterov. A method for solving the convex programming problem with convergence rate o (1/k2). In *Dokl akad nauk Sssr*, volume 269, page 543, 1983.

Arild Nøkland. Direct feedback alignment provides learning in deep neural networks. *Advances in neural information processing systems*, 29, 2016.

Erkki Oja. Simplified neuron model as a principal component analyzer. *Journal of mathematical biology*, 15:267–273, 1982.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr computational mathematics and mathematical physics*, 4(5): 1–17, 1964.

Adityanarayanan Radhakrishnan, Daniel Beaglehole, Parthe Pandit, and Mikhail Belkin. Mechanism for feature learning in neural networks and backpropagation-free machine learning models. *Science*, 383(6690): 1461–1467, 2024. doi: 10.1126/science.adi5639.

Ingo Rechenberg. Evolutionsstrategien. In *Simulationsmethoden in der Medizin und Biologie: Workshop, Hannover, 29. Sept.–1. Okt. 1977*, pages 83–114. Springer, 1978.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

Benjamin Scellier and Yoshua Bengio. Equilibrium propagation: Bridging the gap between energy-based models and backpropagation. In *International Conference on Learning Representations*, 2017.

Jürgen Schmidhuber. *Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook*. PhD thesis, Technische Universität München, 1987.

Hans-Paul Schwefel and Hans-Paul Schwefel. *Evolutionsstrategien für die numerische optimierung*. Springer, 1977.

Sen Song, Kenneth D Miller, and Larry F Abbott. Competitive hebbian learning through spike-timing-dependent synaptic plasticity. *Nature neuroscience*, 3 (9):919–926, 2000.

Gavin Taylor, Ryan Burmeister, Zheng Xu, Bharat Singh, Ankit Patel, and Tom Goldstein. Training neural networks without gradients: A scalable ADMM approach. In *International conference on machine learning*, pages 2722–2731. PMLR, 2016.

The Theano Development Team, Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, et al. Theano: A Python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688*, 2016.

Tijmen Tieleman. Lecture 6.5-RMSProp: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26, 2012.

Xingzi Xu et al. Activation sharding for scalable training of large models. *Transactions on Machine Learning Research*, 2025.

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]
   (b) An analysis of the properties and complexity (time, space, sample size) of any algorithm. [Yes]
   (c) (Optional) Anonymized source code, with specification of all dependencies, including external libraries. [Yes]
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]
   (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]
   (b) All the training details (e.g., data splits, hyperparameters, how they were chosen). [Yes]
   (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). [Yes]
   (d) A description of the computing infrastructure used. (e.g., type of GPUs, internal cluster, or cloud provider). [Yes]
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]
   (b) The license information of the assets, if applicable. [Yes]
   (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 Projection-Based Framework for Gradient-Free and Parallel Learning: Supplementary Materials

A RELATED WORK

Deep neural network training is overwhelmingly dominated by gradient-based optimization methods. However, the limitations of gradient-based methods and inspiration from neuroscience have driven research into non-gradient-based learning. This section reviews prominent families of these methods.

A.1 Gradient-based optimization: The standard paradigm

Gradient-based methods are foundational in deep learning. The backpropagation algorithm, as popularized in the seminal work by Rumelhart et al. (1986), remains the dominant paradigm for training neural networks. Backpropagation leverages the chain rule of differentiation to compute the gradient of a global loss function with respect to all network parameters efficiently. These gradients are then typically used within variants of stochastic gradient descent (SGD) to iteratively update the parameters and minimize the loss.

To address issues such as slow convergence, navigation of complex loss landscapes, and sensitivity to learning rates, several improvements to vanilla SGD have been proposed. Momentum-based methods, including Polyak's momentum (Polyak, 1964) and Nesterov accelerated gradient (Nesterov, 1983), incorporate a history of past updates to accelerate progress along consistent descent directions and dampen oscillations. Adaptive methods such as AdaGrad (Duchi et al., 2011), RMSProp (Tieleman, 2012), and Adam (Kingma and Ba, 2014) adjust learning rates on a per-parameter basis, adapting to the geometry of the loss landscape and helping networks train more robustly, especially with sparse gradients. For large models, memory-efficient variants of backpropagation have also been developed, e.g., reducing activation storage or recomputation overheads (Korthikanti et al., 2023; Xu et al., 2025).

Despite their success, gradient-based methods face several challenges. They can converge to suboptimal local minima or encounter prevalent high-error saddle points, particularly in high-dimensional non-convex optimization problems (Dauphin et al., 2014; Choromanska et al., 2015). Furthermore, they can suffer from vanishing or exploding gradients in deep networks (Hochreiter, 1991; Bengio et al., 1994), require the objective function and network components to be differentiable (or sub-differentiable), and can be sensitive to hyperparameter choices. The computational cost and sequential nature of computing and propagating gradients through deep networks can also be significant bottlenecks for parallelization across layers. These limitations motivate the exploration of alternative training schemes.

A.2 Derivative-free and zeroth-order methods

A direct alternative is to forgo gradient calculation entirely and use derivative-free optimization (DFO) or zeroth-order methods. These methods typically rely only on evaluations of the objective function (the global loss). Evolution Strategies (ES) (Rechenberg, 1978; Schwefel and Schwefel, 1977) are a prominent example. ES optimizes parameters by sampling points in the parameter space around the current estimate, evaluating the loss at these points, and moving the estimate in a direction informed by these evaluations, effectively approximating a gradient direction or directly seeking improvement (Salimans et al., 2017). While computationally intensive due to the need for multiple forward passes per update, ES has shown surprising effectiveness, particularly in reinforcement learning domains (Salimans et al., 2017). Other population-based methods like Genetic Algorithms (GAs) (Holland, 1992) and Particle Swarm Optimization (PSO) (Kennedy and Eberhart, 1995) maintain a population of candidate solutions (network parameters) and iteratively refine them using operators inspired by biological evolution (selection, crossover, mutation) or social behavior (particle movement). The objective is

typically to maximize a fitness function directly related to the global loss. Evolutionary approaches have also been applied to meta-learning problems, such as optimizing learning rules or hyperparameters (Schmidhuber, 1987). The primary objective of these DFO methods is the minimization of the global loss function. However, they often exhibit high sample complexity compared to gradient-based methods, requiring many loss evaluations, which can be prohibitive for large networks and datasets. Scalability to extremely high-dimensional parameter spaces remains a challenge.

A.3 Biologically inspired local learning rules

Drawing inspiration from neuroscience, another class of methods employs local learning rules that do not require global error backpropagation. Rooted in neuroscientific theory, Hebbian learning follows the principle "cells that fire together, wire together." This principle, introduced by Donald Hebb in his 1949 book (Hebb, 1949), states that the synaptic strength between two neurons increases if they are co-activated. Early neural network research used simplified Hebbian rules for unsupervised representation learning. Modern variants, such as Oja's rule (Oja, 1982) and the Bienenstock–Cooper–Munro (BCM) rule (Bienenstock et al., 1982), build on the classical Hebbian framework to ensure weight stabilization and learn principal components or other low-dimensional representations. Relatedly, attractor networks like the Hopfield network (Hopfield, 1982) also utilize Hebbian-style rules to store patterns as stable states (attractors) within an energy function, suitable for tasks like associative memory and pattern completion. Another important family of models leveraging local updates includes energy-based models like Boltzmann Machines (BMs) and particularly their simplified variant, Restricted Boltzmann Machines (RBMs) (Hinton, 2002). RBMs are stochastic networks trained to model a probability distribution over their inputs, typically via Maximum Likelihood. While exact gradient calculation is intractable for general BMs, RBMs can be effectively trained using algorithms like Contrastive Divergence (CD) (Hinton, 2002), which relies on Gibbs sampling and local computations between connected layers to approximate the gradient. This made RBMs instrumental as building blocks for Deep Belief Networks (Hinton et al., 2006), offering a biologically more plausible route to unsupervised feature learning and generative modeling. A common appealing property of these various local learning rules, whether Hebbian or energy-based, is their independence from global gradient signals. Instead, they rely on updates based on local neuronal activity (e.g., correlations, timing, sampling statistics) or principles like energy minimization. The objective is often related to maximizing correlation, capturing variance, achieving representational stability, minimizing a network energy function, or modeling data distributions, rather than directly minimizing a global supervised loss function for input-output mappings. This locality can be advantageous for large-scale or biologically plausible architectures where global signals are unavailable or costly. However, directly applying or adapting these principles to match the performance of gradient-based methods on complex supervised tasks remains challenging.

Inspired by the brain's event-driven architecture, neuromorphic computing focuses on spiking neural networks (SNNs) that communicate via discrete spike events instead of continuous-valued activations. Conventional backpropagation becomes difficult in spiking systems due to the non-differentiable nature of spike functions. As a result, a variety of alternative learning rules, such as Spike-Timing-Dependent Plasticity (STDP) in spiking neural networks (Song et al., 2000; Gerstner and Kistler, 2002), have emerged. STDP aligns closely with Hebbian-like principles, adjusting synaptic strengths based on the relative timing of pre- and post-synaptic spikes. The objective here is local synaptic modification driven by temporal correlations. Neuromorphic chips like IBM's TrueNorth (Merolla et al., 2014) and Intel's Loihi (Davies et al., 2018) have been designed to implement such models efficiently, potentially offering significant energy savings. Because learning in SNNs often occurs locally within each synapse, neuromorphic approaches can circumvent the need for global gradient signals. However, training SNNs for complex, real-world tasks comparable to those solved by deep learning remains an active research area, and performance often lags behind conventional networks. Such methods also inherently handle non-differentiable spike events, a property shared with the projection-based methods discussed in this work.

A.4 Gradient approximation and decoupled training methods

Several approaches attempt to retain some benefits of gradient-based learning while avoiding full backpropagation, often motivated by biological plausibility (e.g., the weight transport problem) or computational considerations. Target Propagation (TP) (Bengio, 2014; Lee et al., 2015) aims to compute layer-specific targets instead of gradients. An inverse mapping or autoencoder associated with each layer generates target activations that, if achieved, would reduce the global loss. The layer then updates its weights locally to better map its input to

these targets. The objective is local: minimize the mismatch between layer outputs and computed targets, serving as a proxy for the global loss. TP can avoid propagating precise gradients but requires learning or defining inverse mappings, which can be challenging and potentially unstable. Equilibrium Propagation (Scellier and Bengio, 2017) is another line of work in this direction: it computes updates from differences between free and weakly clamped network states in energy-based models, avoiding explicit reverse-mode backpropagation through layers. Feedback Alignment (FA) (Lillicrap et al., 2016) replaces the transposed weight matrices used in backpropagation's backward pass with fixed, random feedback matrices. Surprisingly, the network can learn by aligning its forward weights to leverage these random pathways for error signaling. The objective remains the global loss, but the gradient information is approximated. FA solves the weight transport problem but typically learns slower and may achieve lower final performance compared to backpropagation. Direct Feedback Alignment (DFA) (Nøkland, 2016) further simplifies this by sending the error signal directly from the output layer to each hidden layer via fixed random matrices. Decoupled Neural Interfaces (synthetic gradients) (Jaderberg et al., 2017) similarly reduce strict sequential dependencies by learning gradient predictors for intermediate modules. Layer-wise training provides another way to decouple learning. Initially popular for pre-training Deep Belief Networks (Hinton et al., 2006), this involves training layers sequentially, often using unsupervised objectives like reconstruction error before potential end-to-end fine-tuning. The objective is local per layer/stage. While useful for initialization, purely greedy layer-wise training may not yield globally optimal solutions for the final task. More recently, the Forward-Forward algorithm (Hinton, 2022) was proposed as a potential alternative inspired by biology. It discards backpropagation entirely, using two forward passes—one with positive (real) data and one with negative data—and updating weights based on a local goodness metric specific to each layer. The objective is layer-local: maximizing goodness for positive samples and minimizing it for negative samples. This avoids backpropagation but requires generating negative data and doubles the computation per update compared to a single forward pass. Its scalability and performance across diverse tasks are still under investigation. Further exploring alternatives to full backpropagation, Radhakrishnan et al. (2024) recently introduced the Average Gradient Outer Product (AGOP) as a backpropagation-free mathematical mechanism to characterize and enable feature learning. Their work demonstrates that AGOP captures learned features across diverse architectures and can instill feature learning capabilities in models like kernel machines, notably through their Recursive Feature Machine (RFM) algorithm.

A.5 Alternative mathematical optimization frameworks

Beyond heuristic or bio-inspired approaches, alternative mathematical optimization frameworks have been applied to neural network training, often by reformulating the learning problem. The Alternating Direction Method of Multipliers (ADMM) (Boyd et al., 2011) is a general framework for constrained optimization that decomposes a large problem into smaller, potentially easier subproblems that are solved iteratively. It has been explored for training neural networks, sometimes by introducing auxiliary variables and constraints to decouple layers or enforce structure (Glowinski and Marroco, 1975; Taylor et al., 2016). The Method of Auxiliary Coordinates (MAC) represents another such decomposition strategy for deeply nested systems (Carreira-Perpiñán and Wang, 2014). The objective in these frameworks is typically the original global loss, subject to reformulations. ADMM can handle non-differentiable regularizers and constraints but requires careful problem formulation, and solving the subproblems efficiently can be challenging, often involving overhead from dual variable updates or the solution of complex subproblems like large matrix inversions (Taylor et al., 2016). Block Coordinate Descent (BCD) methods optimize the network parameters block by block (e.g., layer by layer) while keeping other parameters fixed (Bertsekas, 1997). Each subproblem optimizes the global loss with respect to a subset of variables. BCD can be simpler to implement and potentially more memory-efficient than SGD for certain structures, but its convergence can be slow, dependent on the block partitioning strategy, and it may struggle with highly correlated parameters.

Crucially, the perspective of training as a **feasibility problem** (finding parameters $\boldsymbol{\theta}$ such that $\boldsymbol{f}(\boldsymbol{x}_i, \boldsymbol{\theta}) = \boldsymbol{y}_i$ for all i) provides a distinct reformulation suitable for iterative projection methods like AP or DR. This "learning without loss" approach was conceptually explored by Elser (2021), who proposed using the Difference Map algorithm (related to DR) to find feasible points in the intersection of constraints derived from individual data samples. Elser demonstrated the concept with illustrative examples. The projection-based method detailed in our work builds directly on this feasibility perspective, leveraging a fine-grained decomposition based on the computation graph's primitive functions, which allows for efficient, parallelizable projection steps.

A.6 Symbolic, logic-based, and combinatorial optimization approaches

A less common direction treats network training or design as a discrete or symbolic problem. By formulating aspects of a neural network's parameters or structure (e.g., activation functions, connectivity) as a discrete optimization problem, one can sometimes leverage mature combinatorial solvers like SAT solvers or integer programming techniques. These approaches often arise in the context of "neuro-symbolic AI," aiming to integrate neural learning with symbolic reasoning. The objective might be to find a network satisfying certain logical constraints or optimizing a discrete objective. However, these methods often suffer from severe scalability challenges as network dimensions grow and are typically applicable only to specific problem types or network architectures.

B PJAX: A PROJECTION-BASED NUMERICAL COMPUTATION FRAMEWORK

Building upon the reformulation of neural network training as a feasibility problem solvable via projection methods (Section 3), we introduce PJAX, a numerical computation framework designed to implement and solve such problems efficiently. PJAX aims to be an analogue of modern automatic differentiation (autodiff) libraries like Theano (Team et al., 2016), TensorFlow (Abadi et al., 2016), PyTorch (Paszke et al., 2019), and JAX (Bradbury et al., 2018). However, unlike autodiff libraries that compute gradients, PJAX's core computational mechanism revolves around projection operators. By leveraging JAX as its backend (hence the name PJAX, with "P" signifying Projection), our framework inherits JAX's just-in-time (JIT) compilation capabilities and seamless execution across different hardware accelerators, including CPUs, GPUs, and TPUs.

The core design principle of PJAX mirrors that of autograd frameworks: users write numerical code using functions provided by the library, and the framework automatically tracks these computations to build an underlying representation—in our case, a computation graph suitable for projection-based methods rather than gradient calculation. An optimizer module then utilizes this graph and the associated projection operators to find feasible solutions to the defined problem. The primary goal is to offer the core functionality of a projection-based computation framework with an interface familiar to users of JAX or similar libraries, complemented by a high-level API for streamlined machine learning model definition.

B.1 Core components

PJAX is built upon core components implemented using JAX primitives. Accessed only indirectly via the high-level API, these components form the foundational building blocks, implementing the essential primitive functions and projection operators required for the optimization process:

Primitive functions correspond to the hidden function nodes in the computation graph described in Section 3.1. Examples include fundamental operations like `identity`, `add`, `dot`, `sum_relu`, `max`, and `quantize`. For each primitive function f, we implement both its standard forward evaluation and the corresponding projection operator $\mathrm{P}_{\mathrm{Graph}(f)}$. Detailed definitions and derivations of these projection operators are included in Section C.1.

Loss functions correspond to the target nodes in the computation graph. For now, we provide a `cross_entropy` and `margin_loss` loss for classification tasks. Further details can be found in Section C.2.

Shape transformations, such as `index`, `reshape`, `transpose`, `repeat`, `concat`, `padding`, and `conv_patch`, are treated as special *no-ops* nodes that manipulate the shape or layout of data without altering its numerical values. These transformations must be invertible, and PJAX implements both the forward and inverse operations. These transformations do not impose feasibility constraints themselves (i.e., they do not define sets $\mathcal{C}_v$ for projection). Instead, when a projection operator $\mathrm{P}_{\mathcal{C}_v}$ associated with a computational node v needs to access values from incoming edges or distribute values to outgoing edges, these values are automatically passed through any intermediate shape transformation nodes using their forward or inverse implementations as needed. This design allows users to incorporate complex tensor manipulations common in neural networks without requiring additional projection operators.

B.2 User API

Users interact with PJAX through a high-level API designed to closely resemble the JAX / NumPy interface, facilitating adoption for those already familiar with these libraries.

Computation container The `Computation` class is fundamental for objects managed by PJAX. A `Computation` instance can hold input data (as an `Array` or `Parameter`) or represent the symbolic output of a PJAX operation, maintaining references to the operation and its inputs. This mechanism allows PJAX to trace the sequence of operations and construct the computation graph implicitly.

Data containers include `Array` (for constant inputs) and `Parameter` (for variables to be optimized). Both are subclasses of `Computation` and act as wrappers around standard JAX arrays (`jax.numpy.ndarray`).

API functions provided by PJAX operate on `Computation` objects and return new `Computation` objects, thereby extending the computation graph. This includes wrappers for the core primitive functions, output constraints, and shape transformations. Furthermore, PJAX offers functions designed to mirror the `jax.numpy` API, such as `dot` (with batching semantics), `matmul`, `swapaxes`, `moveaxis`, `expand_dims`, `squeeze`, `stack`, etc. These higher-level functions are implemented internally using combinations of PJAX core functions and the `vmap` utility, obviating the need to define unique projection operators for each one.

Vectorization (`vmap`) is supported through a utility, `pjax.vmap`, with a signature and semantics analogous to `jax.vmap`. It is used internally to implement batch-aware operations (e.g., `matmul` from `dot`) and is exposed to the user, enabling automatic vectorization of user-defined functions composed of PJAX operations. This is critical for achieving high performance on modern hardware.

B.3 Optimizer module (`pjax.optim`)

The `pjax.optim` module contains the algorithms that solve the feasibility problem defined by the computation graph. Currently available optimizers include Alternating Projections (AP), Cyclic Projections (CP), Douglas–Rachford (DR), and the Difference Map algorithm (Elser, 2003). The optimizer's `update` function accepts:

- A user-defined Python function that computes the desired output using PJAX operations. This function implicitly defines the computation graph and constraints. It is analogous to the function one might pass to `jax.grad`, often representing the forward pass of a model, $\boldsymbol{\theta} \mapsto \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{\theta})$, whose output is then constrained according to the graph (e.g., via Output Constraint nodes).
- A dictionary (or other *pytree* structure) holding the current state of the `Parameter` objects (e.g., network weights).

The optimizer then applies the function to the parameters, computes and stores intermediate outputs, forms a bipartition of the graph, and performs a specified number of update steps using the chosen projection algorithm. Finally, it returns a dictionary containing the updated consensus values for the parameters.

B.4 High-level neural network API (`pjax.nn`)

Inspired by Flax, `pjax.nn` simplifies model definition and training via a compositional `Module` interface for reusable components (e.g., layers, blocks). It automates parameter handling (initialization, naming, sharing), enabling users to define complex architectures with familiar concepts, which are readily compatible with the `pjax.optim` module for projection-based training.

B.5 Example: MLP definition and computation graph

To illustrate how a neural network is defined using the `pjax.nn` API and how PJAX subsequently constructs a detailed computation graph, we consider an MLP with a single hidden layer and ReLU activation. Mathematically,

this MLP is defined as

$$\boldsymbol{f}(\boldsymbol{x}; \boldsymbol{W}^{\text{hidden}}, \boldsymbol{b}^{\text{hidden}}, \boldsymbol{W}^{\text{out}}) = \boldsymbol{W}^{\text{out}} \operatorname{ReLU}(\boldsymbol{W}^{\text{hidden}} \boldsymbol{x} + \boldsymbol{b}^{\text{hidden}}) \tag{13}$$

where $\boldsymbol{x}$ is the input, $\boldsymbol{W}^{\text{hidden}}$ and $\boldsymbol{b}^{\text{hidden}}$ are the weights and biases for the hidden layer, respectively, and $\boldsymbol{W}^{\text{out}}$ represents the weights for the output layer. The corresponding PJAX code for this MLP is shown in Listing 1.

```
import jax
import pjax
from pjax import nn, optim

class MLP(nn.Module):
    def __init__(self, in_features, hidden_features, num_classes):
        super().__init__()
        self.hidden = nn.Linear(in_features, hidden_features)
        self.relu = nn.ReLU(hidden_features)
        self.out = nn.Linear(hidden_features, num_classes)

    def __call__(self, x):
        x = self.hidden(x)
        x = self.relu(x)
        x = self.out(x)
        return x


model = MLP(in_features=784, hidden_features=256, num_classes=10)
params = model.init(jax.random.key(0))
optimizer = optim.DouglasRachford(steps_per_update=50)

for (x, y) in dataloader:
    def loss_fn(params):
        logits = model.apply(params, x)
        return nn.cross_entropy(logits, y)

    params, loss = optimizer.update(loss_fn, params)
```

Listing 1: Python code for an MLP using the `pjax` API, including training loop

When this `MLP` model is instantiated and applied to an input batch (e.g., 32 MNIST vectors, each 784-dimensional, with a hidden layer of 16 features and 10 output classes), PJAX traces the operations. The resulting computation graph, shown in Fig. 7, visualizes this trace.

In this graph, nodes represent PJAX components. `Array` nodes hold constant data like the input batch and target data for the `cross_entropy` loss. Learnable parameters, whose names are assigned by our `pjax.nn` API, are represented as `Parameter` nodes; for this MLP, these include `hidden.weight` (corresponding to $\boldsymbol{W}^{\text{hidden}}$), `relu.bias` (corresponding to $\boldsymbol{b}^{\text{hidden}}$ and applied at the `sum_relu` stage), and `out.weight` (corresponding to $\boldsymbol{W}^{\text{out}}$). The primitive functions shown are `dot` (scalar dot product) and `sum_relu`. Shape transformation nodes, specifically `reshape` and `repeat`, prepare input tensors for the batched scalar operations. For instance, to implement the batched matrix multiplication in the first linear layer ($\boldsymbol{W}^{\text{hidden}}\boldsymbol{x}$), the input `Array (32,784)` and the `hidden.weight (784,16)` parameter are reshaped and repeated to match dimensions for the `dot` operation.

This graph details the concrete sequence of operations as traced by PJAX. In contrast, a conceptual illustration (as in Fig. 2) would represent each neuron's compound operations (dot product and activation) as distinct explicit nodes with edges fanning out to all neurons in the subsequent layer, rather than utilizing PJAX's explicit shape transformations for vectorized scalar primitives. Another minor technical difference is that the conceptual graph does not include a node for the target data, rather, it's implicitly represented in the loss function.

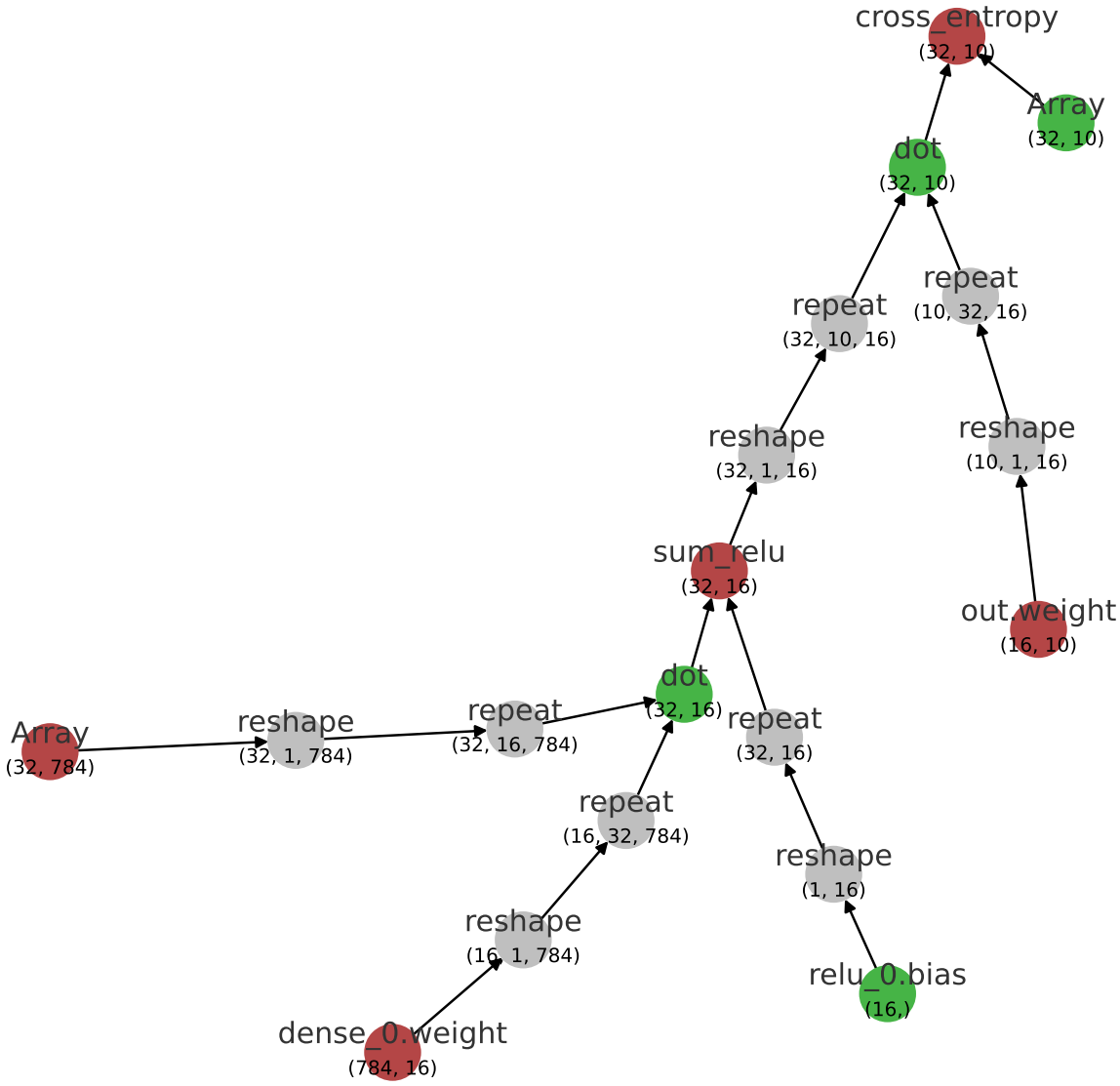

Figure 7: PJAX computation graph for the 16-neuron hidden layer MLP (Listing 1), processing a batch of 32 MNIST samples. Node sublabels indicate tensor shapes.

C PROJECTION OPERATORS

Here we provide the mathematical details underpinning the projection steps central to our method (Section 3). These steps form the computational core of each iteration within the projection algorithms used for training (Algorithm 1). We begin by presenting the theorem that justifies how projections onto hidden function node constraints are computed by leveraging projections onto the graphs of the underlying primitive functions (Theorem 3). Subsequently, in Section C.1, we detail the specific orthogonal projection operators onto the graphs (Graph(f)) for various primitive functions commonly used in neural networks, such as linear operations, activations, and pooling. Finally, Section C.2 describes the operators employed at the output nodes to enforce conditions derived from the learning objective.

Theorem 3 (Projection onto consensus sets)**.** *Let $\bar{\mathcal{C}} \subseteq \mathbb{R}^{d_x} \times \mathbb{R}^{d_y}$ be a non-empty closed set (e.g., the graph of a primitive function). The associated consensus set $\mathcal{C} \subseteq \mathbb{R}^{d_x} \times (\mathbb{R}^{d_y})^n$ is*

$$\mathcal{C} = \left\{(\boldsymbol{w}, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_n) \mid (\boldsymbol{w}, \boldsymbol{u}_i) \in \bar{\mathcal{C}} \forall i = 1, \ldots, n\right\}. \tag{14}$$

Given a point $\boldsymbol{z}_0 = (\boldsymbol{x}_0, \boldsymbol{y}_{0,1}, \ldots, \boldsymbol{y}_{0,n})$, a projection onto $\mathcal{C}$ can be computed as follows:

1. *Compute the average $\boldsymbol{y}$ component: $\bar{\boldsymbol{y}}_0 = \frac{1}{n}\sum_{i=1}^n \boldsymbol{y}_{0,i}$.*
2. *Choose a minimizer of the weighted projection problem for the pair $(\boldsymbol{x}_0, \bar{\boldsymbol{y}}_0)$ onto the base set $\bar{\mathcal{C}}$:*

$$(\boldsymbol{x}, \bar{\boldsymbol{y}}) \in \operatorname*{arg\,min}_{(\boldsymbol{w}, \bar{\boldsymbol{u}}) \in \bar{\mathcal{C}}} \left(\|\boldsymbol{w} - \boldsymbol{x}_0\|^2 + n \|\bar{\boldsymbol{u}} - \bar{\boldsymbol{y}}_0\|^2 \right). \tag{15}$$

3. *A projection onto the consensus set $\mathcal{C}$ is formed by replicating the $\bar{\boldsymbol{y}}$ component:*

$$(\boldsymbol{x}, \underbrace{\bar{\boldsymbol{y}}, \ldots, \bar{\boldsymbol{y}}}_{n \text{ times}}). \tag{16}$$

This theorem justifies the projection procedure for function nodes described in Section 3. In that context, $\bar{\mathcal{C}} = \text{Graph}(f_v)$, $\boldsymbol{x}_0$ holds incoming edge values (z_{uv}), $\boldsymbol{y}_{0,i}$ hold outgoing values z_{vw}, and $n = |\mathcal{N}^+(v)|$. Step 2 yields the updated incoming values $\boldsymbol{x}$ (new (z_{uv})) and the single consensus outgoing value $\bar{\boldsymbol{y}}$ (new z_{out}). Step 3 constructs the state update for $\boldsymbol{z}$.
For simplicity, our implementation uses a standard projection onto $\bar{\mathcal{C}}$ in Step 2, rather than the theoretically derived weighted projection. While this alters the exact projection dynamics, any resulting fixed point still satisfies both the consensus requirement and the base constraint $(\boldsymbol{x}, \bar{\boldsymbol{y}}) \in \bar{\mathcal{C}}$.

Proof. Let $\boldsymbol{z}_0 = (\boldsymbol{x}_0, \boldsymbol{y}_{0,1}, \ldots, \boldsymbol{y}_{0,n})$. We seek a projection of $\boldsymbol{z}_0$ onto $\mathcal{C}$, i.e. a point $(\boldsymbol{x}, \boldsymbol{y}_1, \ldots, \boldsymbol{y}_n) \in \mathcal{C}$ minimizing the squared Euclidean distance f to $\boldsymbol{z}_0$. We use $(\boldsymbol{w}, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_n)$ as dummy variables for points in $\mathcal{C}$

$$\min_{(\boldsymbol{w}, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_n) \in \mathcal{C}} \left(\|\boldsymbol{w} - \boldsymbol{x}_0\|^2 + \sum_{i=1}^n \|\boldsymbol{u}_i - \boldsymbol{y}_{0,i}\|^2 \right). \tag{17}$$

The constraint $(\boldsymbol{w}, \boldsymbol{u}_i) \in \bar{\mathcal{C}}$ for all i implies that any feasible point must satisfy $\boldsymbol{u}_1 = \cdots = \boldsymbol{u}_n = \bar{\boldsymbol{u}}$ for some $\bar{\boldsymbol{u}}$ where $(\boldsymbol{w}, \bar{\boldsymbol{u}}) \in \bar{\mathcal{C}}$. The problem reduces to finding $(\boldsymbol{w}, \bar{\boldsymbol{u}}) \in \bar{\mathcal{C}}$ that minimizes

$$f(\boldsymbol{w}, \bar{\boldsymbol{u}}) = \|\boldsymbol{w} - \boldsymbol{x}_0\|^2 + \sum_{i=1}^n \|\bar{\boldsymbol{u}} - \boldsymbol{y}_{0,i}\|^2. \tag{18}$$

Let $\bar{\boldsymbol{y}}_0 = \frac{1}{n}\sum_{i=1}^n \boldsymbol{y}_{0,i}$. Using the identity $\sum_{i=1}^n \|a - b_i\|^2 = n\left\|a - \bar{b}\right\|^2 + \sum_{i=1}^n \left\|\bar{b} - b_i\right\|^2$, the sum term becomes

$$\sum_{i=1}^n \|\bar{\boldsymbol{u}} - \boldsymbol{y}_{0,i}\|^2 = n\|\bar{\boldsymbol{u}} - \bar{\boldsymbol{y}}_0\|^2 + \sum_{i=1}^n \|\bar{\boldsymbol{y}}_0 - \boldsymbol{y}_{0,i}\|^2. \tag{19}$$

Substituting this into $f(\boldsymbol{w}, \bar{\boldsymbol{u}})$, the minimization problem is equivalent (since the last term is constant w.r.t. $\boldsymbol{w}, \bar{\boldsymbol{u}}$) to

$$\min_{(\boldsymbol{w},\bar{\boldsymbol{u}})\in\bar{\mathcal{C}}} \left(\|\boldsymbol{w} - \boldsymbol{x}_0\|^2 + n \|\bar{\boldsymbol{u}} - \bar{\boldsymbol{y}}_0\|^2 \right). \tag{20}$$

Any minimizer $(\boldsymbol{x}, \bar{\boldsymbol{y}})$ of Eq. (15) yields a projection onto $\mathcal{C}$. Such a projection is constructed by replicating $\bar{\boldsymbol{y}}$, yielding $(\boldsymbol{x}, \underbrace{\bar{\boldsymbol{y}}, \ldots, \bar{\boldsymbol{y}}}_{n \text{ times}})$, as stated in step 3. □

C.1 Projection operators for primitive functions

Below, we detail and derive the projection operators onto the graphs of several primitive functions used within the PJAX framework. Recall that for a function $f : \mathbb{R}^n \to \mathbb{R}^m$, its graph is defined as $\text{Graph}(f) = \{(\boldsymbol{x}, f(\boldsymbol{x})) \mid \boldsymbol{x} \in \mathbb{R}^n\}$. The projection onto this graph, $\text{P}_{\text{Graph}(f)}(\boldsymbol{x}_0, \boldsymbol{y}_0)$, finds the point on the graph closest to $(\boldsymbol{x}_0, \boldsymbol{y}_0)$.

Table 1: Summary of primitive functions presented.

Function	Definition	Theorem Reference
Identity	$f(x) = x$	Theorem 4
Sum	$f(\boldsymbol{x}) = \mathbf{1}^\top \boldsymbol{x}$	Theorem 5
ReLU of summation	$f(\boldsymbol{x}) = \max\{0, \mathbf{1}^\top \boldsymbol{x}\}$	Theorem 6
Dot Product	$f(\boldsymbol{x}, \boldsymbol{y}) = \langle \boldsymbol{x}, \boldsymbol{y} \rangle$	Theorem 7
Maximum	$f(\boldsymbol{x}) = \max_i\{x_i\}$	Theorem 8
Quantization	$f(x)$ maps x to nearest in Z	Theorem 9

Theorem 4 (Identity)**.** *The projection operator onto the graph of the identity function* $\text{id}(x) = x$ *($x \in \mathbb{R}$) is given by an average*

$$\text{P}_{\text{Graph(id)}}(x_0, y_0) = \left(\frac{x_0 + y_0}{2}, \frac{x_0 + y_0}{2} \right). \tag{21}$$

Proof. We seek the point (x, x) on the line $y = x$ that is closest to (x_0, y_0). This involves minimizing the squared distance $(x - x_0)^2 + (x - y_0)^2$. Standard calculus techniques (setting the derivative with respect to x to zero) or geometric reasoning show that the minimum occurs when $x = (x_0 + y_0)/2$. Since the graph is a line (closed and convex), the projection is unique. □

Theorem 5 (Sum)**.** *Let* $\text{sum} : \mathbb{R}^n \to \mathbb{R}$ *be the summation function* $\text{sum}(\boldsymbol{x}) = \mathbf{1}^\top \boldsymbol{x}$*. The projection of* $(\boldsymbol{x}_0, y_0) \in \mathbb{R}^n \times \mathbb{R}$ *onto its graph is*

$$\text{P}_{\text{Graph(sum)}}(\boldsymbol{x}_0, y_0) = (\boldsymbol{x}_0 + \lambda \mathbf{1}, y_0 - \lambda), \quad \textit{where} \quad \lambda = \frac{y_0 - \mathbf{1}^\top \boldsymbol{x}_0}{n + 1}. \tag{22}$$

Proof. We seek the point $(\boldsymbol{x}, y)$ on the hyperplane $y = \mathbf{1}^\top \boldsymbol{x}$ that minimizes the squared distance $\|\boldsymbol{x} - \boldsymbol{x}_0\|^2 + (y - y_0)^2$. Geometrically, the vector connecting $(\boldsymbol{x}_0, y_0)$ to its projection $(\boldsymbol{x}, y)$ must be orthogonal to the hyperplane. The normal vector to the hyperplane $y - \mathbf{1}^\top \boldsymbol{x} = 0$ is $(-\mathbf{1}, 1)$. Thus, $(\boldsymbol{x} - \boldsymbol{x}_0, y - y_0)$ must be proportional to $(-\mathbf{1}, 1)$. Setting $(\boldsymbol{x} - \boldsymbol{x}_0, y - y_0) = -\lambda(-\mathbf{1}, 1)$ gives $\boldsymbol{x} = \boldsymbol{x}_0 + \lambda \mathbf{1}$ and $y = y_0 - \lambda$. Substituting into the hyperplane equation $y = \mathbf{1}^\top \boldsymbol{x}$ allows solving for $\lambda = (y_0 - \mathbf{1}^\top \boldsymbol{x}_0)/(n + 1)$. The graph is an affine subspace (closed and convex), guaranteeing a unique projection. □

Theorem 6 (ReLU of summation). *Define* $\mathrm{SumReLU}(\boldsymbol{x}) = \max\{0, \mathbf{1}^\top \boldsymbol{x}\}$ *for* $\boldsymbol{x} \in \mathbb{R}^n$. *The graph* $\mathcal{C} = \mathrm{Graph}(\mathrm{SumReLU})$ *consists of two parts* ($\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2$)*:*

1. *The "flat" region* $\mathcal{C}_1 = \{(\boldsymbol{x}, y) \mid \mathbf{1}^\top \boldsymbol{x} \leq 0, y = 0\}$.
2. *The "sloped" region* $\mathcal{C}_2 = \{(\boldsymbol{x}, y) \mid \mathbf{1}^\top \boldsymbol{x} \geq 0, y = \mathbf{1}^\top \boldsymbol{x}\}$.

The projection of a point $(\boldsymbol{x}_0, y_0) \in \mathbb{R}^n \times \mathbb{R}$ *onto the graph is found by projecting onto these two regions and selecting the candidate closest to* $(\boldsymbol{x}_0, y_0)$*:*

1. *Candidate 1,* $(\boldsymbol{x}^{(1)}, y^{(1)}) = \mathrm{P}_{\mathcal{C}_1}(\boldsymbol{x}_0, y_0)$*, is given by*

$$\boldsymbol{x}^{(1)} = \boldsymbol{x}_0 - \max\left\{0, \frac{\mathbf{1}^\top \boldsymbol{x}_0}{n}\right\} \mathbf{1}, \qquad y^{(1)} = 0. \tag{23}$$

2. *Candidate 2,* $(\boldsymbol{x}^{(2)}, y^{(2)}) = \mathrm{P}_{\mathcal{C}_2}(\boldsymbol{x}_0, y_0)$*, is computed by first projecting onto the hyperplane* $y = \mathbf{1}^\top \boldsymbol{x}$*:*

$$(\hat{\boldsymbol{x}}, \hat{y}) = (\boldsymbol{x}_0 + \lambda \mathbf{1}, y_0 - \lambda), \quad \text{where } \lambda = \frac{y_0 - \mathbf{1}^\top \boldsymbol{x}_0}{n+1}. \tag{24}$$

Then, the projection onto $\mathcal{C}_2$ *is determined by whether* $(\hat{\boldsymbol{x}}, \hat{y})$ *satisfies the non-negativity constraint*

$$(\boldsymbol{x}^{(2)}, y^{(2)}) = \begin{cases} (\hat{\boldsymbol{x}}, \hat{y}) & \text{if } \mathbf{1}^\top \hat{\boldsymbol{x}} \geq 0 \\ \left(\boldsymbol{x}_0 - \frac{\mathbf{1}^\top \boldsymbol{x}_0}{n} \mathbf{1}, 0\right) & \text{otherwise.} \end{cases} \tag{25}$$

Finally, select

$$\mathrm{P}_{\mathrm{Graph(SumReLU)}}(\boldsymbol{x}_0, y_0) = \underset{(\boldsymbol{x}, y) \in \{(\boldsymbol{x}^{(1)}, y^{(1)}), (\boldsymbol{x}^{(2)}, y^{(2)})\}}{\arg\min} \left\| (\boldsymbol{x}, y) - (\boldsymbol{x}_0, y_0) \right\|^2. \tag{26}$$

Proof. The graph $\mathcal{C} = \mathrm{Graph(SumReLU)}$ is the union of two closed convex sets, $\mathcal{C}_1$ and $\mathcal{C}_2$. The projection $\mathrm{P}_{\mathcal{C}}(\boldsymbol{x}_0, y_0)$ is therefore the point closer to $(\boldsymbol{x}_0, y_0)$ among the projections onto $\mathcal{C}_1$ and $\mathcal{C}_2$. These projections minimize the squared Euclidean distance from $(\mathbf{x}_0, y_0)$ subject to the constraints defining each set. Solving this minimization, for instance using the method of Lagrange multipliers, yields the formulas presented in the theorem. □

Theorem 7 (Dot product). *Let* $\mathrm{dot}(\boldsymbol{x}, \boldsymbol{y}) = \langle \boldsymbol{x}, \boldsymbol{y} \rangle$ *for* $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$. *For a point* $(\boldsymbol{x}_0, \boldsymbol{y}_0, z_0) \in \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}$ *with* $\boldsymbol{x}_0 \neq \pm \boldsymbol{y}_0$*, the orthogonal projection onto the graph* $\mathrm{Graph(dot)}$ *is unique and given by*

$$\mathrm{P}_{\mathrm{Graph(dot)}}(\boldsymbol{x}_0, \boldsymbol{y}_0, z_0) = (\boldsymbol{x}(\lambda), \boldsymbol{y}(\lambda), z_0 - \lambda), \tag{27}$$

where

$$\boldsymbol{x}(\lambda) = \frac{\boldsymbol{x}_0 + \lambda \boldsymbol{y}_0}{1 - \lambda^2}, \quad \boldsymbol{y}(\lambda) = \frac{\boldsymbol{y}_0 + \lambda \boldsymbol{x}_0}{1 - \lambda^2}. \tag{28}$$

The scalar parameter λ *is the unique root in* $]-1, 1[$ *of the function*

$$f(\lambda) = \frac{(1 + \lambda^2)p + \lambda q}{(1 - \lambda^2)^2} - z_0 + \lambda = 0, \tag{29}$$

with $p = \langle \boldsymbol{x}_0, \boldsymbol{y}_0 \rangle$ *and* $q = \|\boldsymbol{x}_0\|^2 + \|\boldsymbol{y}_0\|^2$*. This root can be efficiently found using Newton's method. Practically, 5 to 10 iterations starting from* $\lambda = 0$ *are sufficient to converge to a solution with high accuracy.*

Proof. We seek the point $(\boldsymbol{x}, \boldsymbol{y}, z)$ on the graph $z = \langle \boldsymbol{x}, \boldsymbol{y} \rangle$ that minimizes the squared Euclidean distance

$\|\boldsymbol{x} - \boldsymbol{x}_0\|^2 + \|\boldsymbol{y} - \boldsymbol{y}_0\|^2 + (z - z_0)^2$. This is equivalent to minimizing the unconstrained function

$$g(\boldsymbol{x}, \boldsymbol{y}) = \|\boldsymbol{x} - \boldsymbol{x}_0\|^2 + \|\boldsymbol{y} - \boldsymbol{y}_0\|^2 + (\langle \boldsymbol{x}, \boldsymbol{y} \rangle - z_0)^2 \tag{30}$$

over $(\boldsymbol{x}, \boldsymbol{y}) \in \mathbb{R}^n \times \mathbb{R}^n$. Existence of a minimum is guaranteed as g is continuous and coercive. The first-order optimality conditions are $\nabla_{\boldsymbol{x}} g = \boldsymbol{0}$ and $\nabla_{\boldsymbol{y}} g = \boldsymbol{0}$. Let $\lambda = z_0 - \langle \boldsymbol{x}, \boldsymbol{y} \rangle$. The conditions simplify to

$$\boldsymbol{x} = \boldsymbol{x}_0 + \lambda \boldsymbol{y} \quad \text{and} \quad \boldsymbol{y} = \boldsymbol{y}_0 + \lambda \boldsymbol{x}. \tag{31}$$

Solving this linear system for $\boldsymbol{x}$ and $\boldsymbol{y}$ (assuming $\lambda^2 \neq 1$, which holds for the minimum when $\boldsymbol{x}_0 \neq \pm \boldsymbol{y}_0$) yields

$$\boldsymbol{x}(\lambda) = \frac{\boldsymbol{x}_0 + \lambda \boldsymbol{y}_0}{1 - \lambda^2}, \quad \boldsymbol{y}(\lambda) = \frac{\boldsymbol{y}_0 + \lambda \boldsymbol{x}_0}{1 - \lambda^2}. \tag{32}$$

Substituting these back into the definition $\lambda = z_0 - \langle \boldsymbol{x}(\lambda), \boldsymbol{y}(\lambda) \rangle$ and simplifying leads to the condition $f(\lambda) = 0$ as defined in Eq. (29), where $p = \langle \boldsymbol{x}_0, \boldsymbol{y}_0 \rangle$ and $q = \|\boldsymbol{x}_0\|^2 + \|\boldsymbol{y}_0\|^2$. Analogous to the analysis for projections onto hyperbolas, the function $f(\lambda)$ has a unique root in $]-1, 1[$ when $\boldsymbol{x}_0 \neq \pm \boldsymbol{y}_0$, ensuring a unique projection (Bauschke et al., 2022). The z component of the projection is $z = \langle \boldsymbol{x}(\lambda), \boldsymbol{y}(\lambda) \rangle = z_0 - \lambda$. □

Theorem 8 (Maximum)**.** *Let* $\max : \mathbb{R}^n \to \mathbb{R}$ *be the maximum function* $\max(\boldsymbol{x}) = \max_i\{x_i\}$*. To project* $(\boldsymbol{x}_0, y_0)$ *onto the graph* $y = \max(\boldsymbol{x})$*, first sort* $\boldsymbol{x}_0$ *such that* $x_{0,1} \leq \ldots \leq x_{0,n}$ *(and note the permutation used). Then, for* $k = 1, \ldots, n$*, generate candidate points* $(\boldsymbol{x}^{(k)}, y^{(k)})$ *based on the hypothesis that the maximum* y *is achieved by components* $x_k, \ldots, x_n$*:*

$$y^{(k)} = \frac{(\sum_{i=k}^{n} x_{0,i}) + y_0}{n - k + 2} \qquad \textit{and} \qquad \boldsymbol{x}_i^{(k)} = \begin{cases} y^{(k)}, & \textit{if } i \geq k \\ x_{0,i}, & \textit{if } i < k \end{cases} \tag{33}$$

A candidate $(\boldsymbol{x}^{(k)}, y^{(k)})$ *is* valid *if* $\max(\boldsymbol{x}^{(k)}) = y^{(k)}$*. Given the sorted input, this simplifies to checking* $x_{0,k-1} \leq y^{(k)}$ *(for* $k > 1$*;* $k = 1$ *is always valid). A projection is found by selecting a valid candidate* $(\boldsymbol{x}^{(k)}, y^{(k)})$ *that minimizes the distance* $\left\|(\boldsymbol{x}^{(k)}, y^{(k)}) - (\boldsymbol{x}_0, y_0)\right\|^2$*. The final vector* $\boldsymbol{x}$ *must be permuted back to the original order.*

Proof. We minimize the squared distance $f(\boldsymbol{x}, y) = \|\boldsymbol{x} - \boldsymbol{x}_0\|^2 + (y - y_0)^2$ subject to $y = \max_i x_i$. The graph of the maximum function is closed, so a projection exists. Let $(\boldsymbol{x}, y)$ be a projection, and let $I = \{i \mid x_i = y\}$ be the non-empty set of indices achieving the maximum. For $i \notin I$, optimality forces $x_i = x_{0,i}$; otherwise one could move x_i toward $x_{0,i}$ while keeping $x_i \leq y$ and decrease f. Hence $x_{0,i} \leq y$ for all $i \notin I$. After sorting $\boldsymbol{x}_0$, if there exist indices $i < j$ with $i \in I$ and $j \notin I$, then replacing $(x_i, x_j) = (y, x_{0,j})$ by $(x_{0,i}, y)$ preserves feasibility and does not increase the objective, since $x_{0,i} \leq x_{0,j} \leq y$. Repeating this exchange yields a projection whose active set is of the form $I = \{k, \ldots, n\}$ for some $k \in \{1, \ldots, n\}$. For such a suffix active set, we have $x_i = y$ for $i \geq k$ and $x_i = x_{0,i}$ for $i < k$. The objective thus reduces to minimizing

$$g(y) = \sum_{i=k}^{n} (x_{0,i} - y)^2 + (y_0 - y)^2 \tag{34}$$

with respect to y. Setting the derivative $g'(y) = 0$ gives the value $y = y^{(k)}$ defined in the theorem, with corresponding candidate $\boldsymbol{x}^{(k)}$. This candidate assumes the structure holds, which requires the validity check $x_{0,k-1} \leq y^{(k)}$ (for $k > 1$). These n candidates cover all possible active sets of the form $I = \{k, \ldots, n\}$ for a projection, so any valid candidate minimizing f is a projection. The $k = 1$ candidate is always valid, ensuring the candidate list is non-empty. □

Theorem 9 (Quantization). *Let* $\text{quant} : \mathbb{R} \to \mathbb{R}$ *be the quantization function that maps a real number* x *to the nearest point in a set* $Z = \{z_1, z_2, \ldots, z_k\}$ *of* $k \geq 2$ *equidistant points within* $[-\alpha, \alpha]$*, where* $\alpha > 0$*. Specifically, the points* z_i *are given by*

$$z_i = -\alpha + (i-1)\frac{2\alpha}{k-1} \quad \text{for } i = 1, \ldots, k. \tag{35}$$

The function partitions the real line into intervals I_i *such that* $\text{quant}(x) = z_i$ *for* $x \in I_i$*. These intervals are defined by the* $k-1$ *midpoints* $m_i = (z_i + z_{i+1})/2$ *for* $i = 1, \ldots, k-1$*:*

$$I_1 = (-\infty, m_1] \tag{36}$$

$$I_i = (m_{i-1}, m_i] \quad \text{for } i = 2, \ldots, k-1 \tag{37}$$

$$I_k = (m_{k-1}, \infty) \tag{38}$$

The graph of the quantization function is the union of horizontal line segments and rays

$$\text{Graph}(\text{quant}) = \bigcup_{i=1}^{k} (I_i \times \{z_i\}) \tag{39}$$

This set is generally non-convex.
To project a point $(x_0, y_0) \in \mathbb{R} \times \mathbb{R}$ *onto this graph, first compute* k *candidate points* $(x^{(i)}, y^{(i)})$ *by projecting* x_0 *onto each interval* I_i *and keeping the corresponding* y *value* z_i*:*

$$(x^{(i)}, y^{(i)}) = (\mathrm{P}_{I_i}(x_0), z_i) \qquad \text{for } i = 1, \ldots, k. \tag{40}$$

The projection $\mathrm{P}_{\text{Graph(quant)}}(x_0, y_0)$ *is then given by the candidate* $(x^{(i)}, y^{(i)})$ *that is closest to* (x_0, y_0)*:*

$$\mathrm{P}_{\text{Graph(quant)}}(x_0, y_0) = \operatorname*{arg\,min}_{i \in \{1, \ldots, k\}} \left\| (x^{(i)}, y^{(i)}) - (x_0, y_0) \right\|^2. \tag{41}$$

Note that since the graph is non-convex, the minimum distance might be achieved by multiple candidates; the arg min *selects one such point (consistent with Eq. (1)).*

Proof. We seek $(x, y) \in \text{Graph(quant)}$ that minimizes the squared distance $\|(x, y) - (x_0, y_0)\|^2$. The graph is the union of closed, non-convex pieces, hence a minimum distance exists, but the projection is not necessarily unique.

The minimizing point (x, y) must belong to some piece $\mathcal{C}_j = I_j \times \{z_j\}$. Consider the minimization restricted to an arbitrary but fixed piece $\mathcal{C}_i = I_i \times \{z_i\}$. A point (x, z_i) on this piece minimizes $\|(x, z_i) - (x_0, y_0)\|^2$ subject to $x \in I_i$. This is equivalent to finding $x \in I_i$ that minimizes $(x - x_0)^2$, whose solution is $x = \mathrm{P}_{I_i}(x_0)$. This identifies the candidate $(x^{(i)}, y^{(i)}) = (\mathrm{P}_{I_i}(x_0), z_i)$ as the closest point on $\mathcal{C}_i$ to (x_0, y_0).

The overall projection is the candidate $(x^{(j)}, y^{(j)})$ that yields the minimum squared distance $\left\| (x^{(i)}, y^{(i)}) - (x_0, y_0) \right\|^2$ among all $i \in \{1, \ldots, k\}$, as stated in the theorem. □

C.2 Output operators

This subsection describes operators used at the output nodes (Section 3) to enforce conditions derived from the task's loss function. We provide the projection operators for the margin loss constraint and the proximal operator for the cross-entropy loss function.

Theorem 10 (Margin loss constraint)**.** *Consider a classification setting where the goal is to enforce a condition on a single logit output $x \in \mathbb{R}$ based on a label $y \in \mathbb{R}$. The margin loss constraint requires the logit to be non-positive for negative labels and greater than or equal to a positive margin $m > 0$ for positive labels. Let the constraint set $\mathcal{C}_{y,m}$ be defined as*

$$\mathcal{C}_{y,m} = \begin{cases} (-\infty, 0] & \text{if } y \leq 0 \\ [m, \infty) & \text{otherwise.} \end{cases} \tag{42}$$

The orthogonal projection of a point $x_0 \in \mathbb{R}$ onto this set is given by

$$\mathrm{P}_{\mathcal{C}_{y,m}}(x_0) = \begin{cases} \min(x_0, 0) & \text{if } y \leq 0 \\ \max(x_0, m) & \text{otherwise.} \end{cases} \tag{43}$$

Proof. We seek $x \in \mathcal{C}_{y,m}$ that minimizes $(x - x_0)^2$. Case 1: $y \leq 0$. We need $x \in (-\infty, 0]$. If $x_0 \leq 0$, then x_0 is already in the set, and the minimum distance (zero) is achieved at $x = x_0$. If $x_0 > 0$, the closest point in $(-\infty, 0]$ is $x = 0$. Thus, the projection is $\min(x_0, 0)$. Case 2: $y > 0$. We need $x \in [m, \infty)$. If $x_0 \geq m$, then x_0 is in the set, and the minimum distance is achieved at $x = x_0$. If $x_0 < m$, the closest point in $[m, \infty)$ is $x = m$. Thus, the projection is $\max(x_0, m)$. Combining both cases yields the stated formula. □

Theorem 11 (Proximal operator for cross-entropy loss)**.** *Let $\ell_{CE} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ be the standard cross-entropy loss function for multi-class classification, defined for logits $\boldsymbol{x} \in \mathbb{R}^d$ and a one-hot encoded target label vector $\boldsymbol{y} \in \{0, 1\}^d$ (with $\mathbf{1}^\top \boldsymbol{y} = 1$) as*

$$\ell_{CE}(\boldsymbol{x}, \boldsymbol{y}) = \log\left(\sum_{j=1}^{d} e^{x_j}\right) - \langle \boldsymbol{y}, \boldsymbol{x} \rangle. \tag{44}$$

The proximal operator of this loss function, scaled by $\lambda > 0$, applied to a point $\boldsymbol{x}_0 \in \mathbb{R}^d$ is defined as

$$\mathrm{prox}_{\lambda \ell_{CE}(\cdot, \boldsymbol{y})}(\boldsymbol{x}_0) = \underset{\boldsymbol{x} \in \mathbb{R}^d}{\arg\min} \left(\lambda \ell_{CE}(\boldsymbol{x}, \boldsymbol{y}) + \frac{1}{2} \|\boldsymbol{x} - \boldsymbol{x}_0\|^2 \right). \tag{45}$$

The unique minimizer $\boldsymbol{x}^* = \mathrm{prox}_{\lambda \ell_{CE}(\cdot, \boldsymbol{y})}(\boldsymbol{x}_0)$ is characterized by the condition

$$\boldsymbol{x}^* = \boldsymbol{x}_0 + \lambda(\boldsymbol{y} - \mathrm{softmax}(\boldsymbol{x}^*)). \tag{46}$$

This point $\boldsymbol{x}^*$ can be found efficiently using iterative methods targeting this fixed-point equation, such as fixed-point iteration or Newton's method applied to the equivalent root-finding problem $\boldsymbol{x} - \boldsymbol{x}_0 - \lambda(\boldsymbol{y} - \mathrm{softmax}(\boldsymbol{x})) = \mathbf{0}$. Practically, we found that a fixed-point iteration with 10 iterations provides a stable and efficient solution.

Proof. The function $g(\boldsymbol{x}) = \lambda \ell_{CE}(\boldsymbol{x}, \boldsymbol{y}) + \frac{1}{2} \|\boldsymbol{x} - \boldsymbol{x}_0\|^2$ is strictly convex because $\ell_{CE}(\boldsymbol{x}, \boldsymbol{y})$ is convex and $\frac{1}{2} \|\boldsymbol{x} - \boldsymbol{x}_0\|^2$ is strictly convex. Therefore, a unique minimizer $\boldsymbol{x}^*$ exists. The minimizer is characterized by the first-order optimality condition $\nabla g(\boldsymbol{x}^*) = \mathbf{0}$. The gradient is

$$\nabla g(\boldsymbol{x}) = \lambda \nabla_{\boldsymbol{x}} \ell_{CE}(\boldsymbol{x}, \boldsymbol{y}) + \nabla_{\boldsymbol{x}} \left(\frac{1}{2} \|\boldsymbol{x} - \boldsymbol{x}_0\|^2 \right). \tag{47}$$

The gradient of the cross-entropy loss is $\nabla_{\boldsymbol{x}} \ell_{CE}(\boldsymbol{x}, \boldsymbol{y}) = \mathrm{softmax}(\boldsymbol{x}) - \boldsymbol{y}$. The gradient of the quadratic term is $\boldsymbol{x} - \boldsymbol{x}_0$. Setting the total gradient to zero at $\boldsymbol{x}^*$ gives

$$\lambda(\mathrm{softmax}(\boldsymbol{x}^*) - \boldsymbol{y}) + (\boldsymbol{x}^* - \boldsymbol{x}_0) = \mathbf{0}. \tag{48}$$

Rearranging this equation yields the characterization

$$\boldsymbol{x}^* = \boldsymbol{x}_0 - \lambda(\mathrm{softmax}(\boldsymbol{x}^*) - \boldsymbol{y}) = \boldsymbol{x}_0 + \lambda(\boldsymbol{y} - \mathrm{softmax}(\boldsymbol{x}^*)). \tag{49}$$

This confirms Eq. (46). □

D DETAILED EXPERIMENTAL RESULTS

This section provides the detailed numerical results from the experiments described in Section 5. We compare the performance of the projection-based methods (Douglas-Rachford (DR), Alternating Projections (AP), Cyclic Projections (CP)) against gradient-based baselines (Stochastic Gradient Descent (SGD), Adam) and gradient-free baselines (Feedback Alignment (FA) for MLPs and Forward–Forward (FF) for MLPs and CNNs). The results are presented in the tables below (Table 2 for MLPs, Table 3 for CNNs, and Table 4 for RNNs).

These results are obtained using default train/test splits (approximately 80% train, 20% test), with 10% of training data reserved for validation. Early stopping is applied based on validation accuracy, with a patience of 5 checks (5000 steps) if no improvement is observed. Metrics are averaged over 5 independent runs with different random seeds for initialization and data shuffling; standard deviations ($\pm$) are reported where appropriate.

- **Test Accuracy (%):** The final classification accuracy achieved by the best model (selected based on validation performance) on the held-out test set. For the Shakespeare dataset, this refers to the accuracy of predicting the next character.
- **Train Accuracy (%):** The final classification accuracy on the training set, reported for comparison with test accuracy.
- **Steps (k):** The number of training steps (parameter updates), reported in thousands ('k'), required to reach 99% of the maximum validation accuracy achieved during that run. This metric indicates convergence speed in terms of updates.
- **Time (s):** The wall-clock time in seconds required to reach 99% of the maximum validation accuracy. This measurement excludes time spent on data loading, validation checks, and initial JIT compilation, focusing on the core training loop execution time.
- **Time/Step (ms):** The average wall-clock time in milliseconds per single training step (parameter update step).

Table 2: MLP Performance Comparison.

Dataset	Architecture	Method	Test Acc. (%)	Train Acc. (%)	Steps (k)	Time (s)	Time/Step (ms)
MNIST	1 x 128	SGD	96.8 ± 0.1	97.6 ± 0.1	58.6 ± 7.2	171.4 ± 21.4	2.92 ± 0.01
		Adam	98.1 ± 0.1	100.0 ± 0.0	1.6 ± 0.5	4.7 ± 1.4	2.95 ± 0.01
		FA	97.6 ± 0.1	99.5 ± 0.2	8.0 ± 1.1	23.3 ± 3.1	2.92 ± 0.02
		FF (SGD)	12.7 ± 1.9	12.8 ± 2.2	1.6 ± 1.7	5.1 ± 5.6	3.12 ± 0.08
		FF (Adam)	95.6 ± 0.5	96.2 ± 0.6	82.4 ± 11.8	256.2 ± 35.2	3.12 ± 0.04
		DR	95.8 ± 0.6	96.9 ± 0.7	16.6 ± 3.3	3.7 ± 0.7	0.22 ± 0.00
		AP	91.5 ± 0.7	91.4 ± 0.8	25.8 ± 5.4	2.3 ± 0.5	0.09 ± 0.00
		CP	92.4 ± 0.6	92.5 ± 0.7	41.6 ± 11.4	8.6 ± 2.3	0.21 ± 0.00
	4 x 128	SGD	96.9 ± 0.2	98.8 ± 0.2	32.4 ± 2.3	96.2 ± 7.2	2.97 ± 0.03
		Adam	98.1 ± 0.1	99.9 ± 0.1	1.8 ± 0.7	5.7 ± 2.3	3.09 ± 0.05
		FA	95.4 ± 0.8	96.1 ± 1.0	20.0 ± 6.2	60.2 ± 18.3	3.02 ± 0.03
		FF (SGD)	11.9 ± 0.8	11.8 ± 0.7	2.8 ± 1.7	9.0 ± 5.4	3.24 ± 0.05
		FF (Adam)	94.3 ± 0.4	94.2 ± 0.6	66.0 ± 9.3	220.3 ± 32.1	3.35 ± 0.06
		DR	85.5 ± 0.6	86.2 ± 0.9	29.6 ± 7.3	10.6 ± 2.6	0.36 ± 0.00
		AP	79.6 ± 0.5	79.2 ± 0.8	46.2 ± 4.2	9.4 ± 0.8	0.20 ± 0.00
		CP	79.9 ± 0.6	79.3 ± 0.8	47.8 ± 9.5	17.1 ± 3.5	0.36 ± 0.00
	4 x 128 + Skip	SGD	96.6 ± 0.3	98.1 ± 0.6	28.2 ± 7.2	84.9 ± 21.9	3.01 ± 0.01
		Adam	98.1 ± 0.2	100.0 ± 0.1	1.4 ± 0.5	4.3 ± 1.6	3.08 ± 0.01
		DR	94.9 ± 0.4	96.1 ± 0.7	21.8 ± 6.0	9.3 ± 2.6	0.43 ± 0.00
		AP	90.3 ± 0.8	90.3 ± 0.8	32.6 ± 9.3	6.9 ± 2.0	0.21 ± 0.00
		CP	91.1 ± 0.5	91.1 ± 0.5	34.2 ± 3.5	18.7 ± 1.9	0.55 ± 0.00
CIFAR-10	1 x 256	SGD	50.5 ± 0.1	67.0 ± 2.0	34.6 ± 4.8	307.1 ± 42.1	8.88 ± 0.06
		Adam	50.7 ± 0.8	80.9 ± 7.6	2.2 ± 0.4	20.2 ± 3.6	9.14 ± 0.09
		FA	49.9 ± 0.9	73.4 ± 2.9	5.8 ± 1.2	52.3 ± 10.6	9.08 ± 0.12
		FF (SGD)	9.9 ± 0.2	10.2 ± 0.4	2.6 ± 2.7	24.5 ± 25.3	9.29 ± 0.10
		FF (Adam)	45.4 ± 0.7	50.7 ± 1.3	100.0 ± 8.9	919.3 ± 83.5	9.19 ± 0.03
		DR	42.1 ± 1.1	55.6 ± 4.8	27.2 ± 10.3	62.3 ± 23.5	2.29 ± 0.00
		AP	38.5 ± 0.5	40.3 ± 0.5	11.0 ± 3.7	4.8 ± 1.6	0.43 ± 0.00
		CP	38.5 ± 0.5	40.3 ± 0.5	9.8 ± 3.1	14.4 ± 4.6	1.47 ± 0.00
	4 x 256	SGD	47.8 ± 0.3	67.6 ± 2.6	21.6 ± 3.1	198.3 ± 27.5	9.16 ± 0.10
		Adam	50.0 ± 0.2	79.3 ± 10.3	2.6 ± 1.4	24.4 ± 12.4	9.37 ± 0.05
		FA	29.3 ± 3.6	29.5 ± 3.4	9.0 ± 2.3	84.6 ± 21.1	9.34 ± 0.09
		FF (SGD)	11.0 ± 0.6	11.2 ± 0.6	1.2 ± 1.0	11.7 ± 9.2	9.48 ± 0.18
		FF (Adam)	43.6 ± 1.0	48.0 ± 2.5	92.6 ± 16.6	870.8 ± 155.3	9.40 ± 0.03
		DR	32.3 ± 0.9	35.4 ± 1.1	13.2 ± 5.5	39.8 ± 16.7	3.01 ± 0.00
		AP	32.5 ± 0.6	33.7 ± 0.9	13.6 ± 7.4	9.4 ± 5.1	0.69 ± 0.00
		CP	32.5 ± 0.5	33.8 ± 0.8	13.6 ± 7.4	27.0 ± 14.6	1.98 ± 0.00
	4 x 256 + Skip	SGD	49.4 ± 0.3	68.7 ± 1.7	21.4 ± 2.4	200.5 ± 23.6	9.32 ± 0.08
		Adam	50.6 ± 0.2	89.2 ± 6.0	3.2 ± 1.0	30.0 ± 8.9	9.33 ± 0.20
		DR	39.2 ± 0.8	45.3 ± 1.9	20.8 ± 6.5	61.2 ± 19.1	2.94 ± 0.00
		AP	38.9 ± 0.6	41.0 ± 0.8	14.6 ± 3.9	10.2 ± 2.7	0.70 ± 0.00
		CP	38.9 ± 0.7	41.0 ± 0.7	14.6 ± 3.9	42.4 ± 11.3	2.91 ± 0.00
Higgs	1 x 256	SGD	66.9 ± 0.3	66.9 ± 0.3	63.6 ± 11.4	55.5 ± 10.3	0.88 ± 0.01
		Adam	74.6 ± 0.1	74.6 ± 0.1	29.0 ± 3.9	27.3 ± 3.8	0.97 ± 0.01
		FA	71.3 ± 0.5	71.3 ± 0.5	59.0 ± 18.9	52.1 ± 17.1	0.89 ± 0.01
		FF (SGD)	57.6 ± 5.8	57.7 ± 5.7	152.8 ± 191.6	168.1 ± 210.9	1.07 ± 0.03
		FF (Adam)	71.1 ± 0.3	71.0 ± 0.4	55.6 ± 12.6	66.0 ± 15.5	1.18 ± 0.03
		DR	67.1 ± 0.4	67.0 ± 0.4	24.8 ± 5.1	0.9 ± 0.2	0.04 ± 0.00
		AP	65.4 ± 0.3	65.3 ± 0.3	27.4 ± 13.5	0.6 ± 0.3	0.02 ± 0.00
		CP	65.2 ± 0.4	65.2 ± 0.5	25.4 ± 10.8	0.7 ± 0.3	0.03 ± 0.00
	4 x 256	SGD	71.1 ± 0.6	71.1 ± 0.6	178.8 ± 37.7	176.0 ± 38.7	0.98 ± 0.01
		Adam	76.6 ± 0.2	76.6 ± 0.2	26.0 ± 4.9	26.8 ± 4.7	1.06 ± 0.02
		FA	62.4 ± 3.1	62.4 ± 3.2	6.6 ± 2.0	6.1 ± 1.8	0.92 ± 0.02
		FF (SGD)	57.3 ± 2.3	57.3 ± 2.2	35.4 ± 19.6	41.4 ± 23.2	1.16 ± 0.02
		FF (Adam)	69.4 ± 0.3	69.4 ± 0.3	33.6 ± 5.1	44.3 ± 7.5	1.33 ± 0.01
		DR	61.1 ± 0.3	61.1 ± 0.3	10.8 ± 5.8	5.4 ± 2.9	0.50 ± 0.00
		AP	51.7 ± 5.4	51.7 ± 5.4	3.4 ± 6.8	1.0 ± 2.0	0.29 ± 0.00
		CP	51.7 ± 5.4	51.7 ± 5.4	3.4 ± 6.8	1.9 ± 3.9	0.57 ± 0.00
	4 x 256 + Skip	SGD	68.9 ± 0.4	68.9 ± 0.4	73.0 ± 10.3	72.4 ± 10.9	0.99 ± 0.01
		Adam	76.2 ± 0.1	76.2 ± 0.1	22.6 ± 1.7	25.2 ± 1.9	1.15 ± 0.02
		DR	62.9 ± 0.6	62.9 ± 0.5	10.0 ± 3.6	6.4 ± 2.3	0.64 ± 0.00
		AP	58.2 ± 5.7	58.3 ± 5.8	13.0 ± 10.4	3.9 ± 3.1	0.30 ± 0.00
		CP	58.4 ± 5.8	58.4 ± 5.8	14.0 ± 10.4	6.7 ± 5.0	0.48 ± 0.00

Table 3: CNN Performance Comparison.

Dataset	Architecture	Method	Test Acc. (%)	Train Acc. (%)	Steps (k)	Time (s)	Time/Step (ms)
MNIST	1 x 32	SGD	71.7 ± 1.1	70.8 ± 1.5	133.0 ± 23.9	402.8 ± 72.6	3.03 ± 0.00
		Adam	77.8 ± 1.2	77.1 ± 1.4	26.2 ± 3.4	81.7 ± 11.1	3.11 ± 0.02
		FF (SGD)	24.5 ± 4.0	24.1 ± 4.0	4.0 ± 2.8	13.3 ± 9.1	3.32 ± 0.04
		FF (Adam)	70.6 ± 3.4	69.9 ± 3.1	19.6 ± 6.0	65.1 ± 18.1	3.35 ± 0.11
		DR	51.7 ± 1.2	51.3 ± 1.5	12.4 ± 5.0	31.7 ± 12.8	2.55 ± 0.00
		AP	49.8 ± 1.6	49.4 ± 1.8	25.8 ± 2.3	72.2 ± 6.5	2.80 ± 0.00
		CP	49.9 ± 1.6	49.4 ± 1.8	25.6 ± 2.0	72.1 ± 5.6	2.82 ± 0.00
	4 x 16	SGD	96.0 ± 0.2	96.3 ± 0.2	50.4 ± 8.0	161.0 ± 25.6	3.20 ± 0.03
		Adam	97.4 ± 0.3	99.0 ± 0.4	5.4 ± 1.0	18.4 ± 3.5	3.41 ± 0.02
		FF (SGD)	20.0 ± 3.0	19.5 ± 2.8	2.2 ± 1.9	8.1 ± 7.3	3.58 ± 0.10
		FF (Adam)	88.3 ± 1.2	87.7 ± 1.1	29.0 ± 6.0	108.3 ± 22.1	3.73 ± 0.03
		DR	55.4 ± 4.1	55.1 ± 3.8	99.4 ± 51.0	7713.7 ± 3959.3	77.60 ± 0.01
		AP	46.4 ± 4.8	46.0 ± 4.5	20.8 ± 3.3	1358.7 ± 216.8	65.32 ± 0.04
		CP	46.4 ± 4.8	46.1 ± 4.5	20.6 ± 3.0	1426.7 ± 208.6	69.24 ± 0.09
	4 x 16 + Skip	SGD	96.7 ± 0.3	97.1 ± 0.3	51.4 ± 10.2	166.7 ± 33.4	3.25 ± 0.01
		Adam	97.9 ± 0.2	99.1 ± 0.3	4.0 ± 0.6	13.7 ± 2.2	3.40 ± 0.03
		DR	71.3 ± 1.6	70.8 ± 1.4	16.0 ± 6.4	1303.8 ± 518.2	81.47 ± 0.03
		AP	69.4 ± 1.4	68.8 ± 1.4	46.2 ± 15.2	3240.5 ± 1068.3	70.14 ± 0.02
		CP	69.6 ± 1.3	68.9 ± 1.2	44.2 ± 10.5	1609.5 ± 383.7	36.42 ± 0.03
CIFAR-10	1 x 32	SGD	39.6 ± 1.1	40.0 ± 1.4	91.0 ± 19.2	821.3 ± 172.9	9.03 ± 0.03
		Adam	45.4 ± 0.6	45.9 ± 0.6	22.8 ± 6.3	210.7 ± 57.8	9.23 ± 0.03
		FF (SGD)	10.1 ± 0.0	10.0 ± 0.0	1.0 ± 0.0	9.7 ± 0.4	9.48 ± 0.13
		FF (Adam)	26.9 ± 2.1	27.0 ± 2.5	8.8 ± 4.0	84.3 ± 38.2	9.56 ± 0.05
		DR	27.6 ± 0.6	27.5 ± 0.9	2.4 ± 1.0	27.4 ± 11.6	11.41 ± 0.00
		AP	29.1 ± 0.9	29.1 ± 0.8	19.0 ± 5.4	110.6 ± 31.5	5.82 ± 0.00
		CP	29.2 ± 0.9	29.2 ± 0.9	18.8 ± 5.6	109.8 ± 32.7	5.84 ± 0.00
	4 x 16	SGD	49.2 ± 1.8	51.2 ± 2.3	83.6 ± 25.9	786.5 ± 245.4	9.40 ± 0.05
		Adam	54.9 ± 0.8	62.8 ± 1.8	12.0 ± 4.0	114.2 ± 38.8	9.49 ± 0.09
		FF (SGD)	10.0 ± 0.1	10.1 ± 0.1	0.6 ± 0.5	6.0 ± 4.9	9.96 ± 0.33
		FF (Adam)	28.3 ± 3.3	28.4 ± 3.2	10.4 ± 6.3	102.3 ± 61.7	9.81 ± 0.09
		DR	20.3 ± 0.9	20.2 ± 0.6	2.8 ± 2.2	301.7 ± 240.1	107.78 ± 0.07
		AP	22.8 ± 1.4	22.5 ± 1.2	9.2 ± 5.1	827.1 ± 457.4	89.88 ± 0.17
		CP	22.6 ± 1.3	22.4 ± 1.2	9.0 ± 4.8	847.5 ± 448.7	94.44 ± 0.14
	4 x 16 + Skip	SGD	54.1 ± 1.8	57.0 ± 2.1	106.8 ± 22.4	1004.6 ± 223.9	9.38 ± 0.16
		Adam	59.4 ± 0.6	67.6 ± 0.8	13.0 ± 3.2	123.9 ± 32.1	9.49 ± 0.08
		DR	31.4 ± 1.2	31.8 ± 1.6	3.2 ± 0.4	365.3 ± 45.5	114.13 ± 0.18
		AP	33.4 ± 1.0	33.5 ± 1.1	24.6 ± 4.6	2352.5 ± 439.3	95.68 ± 0.06
		CP	33.7 ± 1.2	33.8 ± 1.2	24.8 ± 4.6	1433.4 ± 266.9	57.82 ± 0.02

Table 4: RNN Performance Comparison.

Dataset	Architecture	Method	Test Acc. (%)	Train Acc. (%)	Steps (k)	Time (s)	Time/Step (ms)
Shakespeare	1 x 256	SGD	35.7 ± 5.1	38.8 ± 6.4	179.0 ± 81.4	480.9 ± 218.7	2.69 ± 0.00
		Adam	49.9 ± 0.0	59.8 ± 0.4	7.4 ± 1.4	20.0 ± 3.7	2.70 ± 0.00
		DR	32.4 ± 0.5	34.0 ± 0.7	16.6 ± 5.5	849.0 ± 283.4	51.14 ± 0.02
		AP	29.6 ± 0.8	30.4 ± 1.0	98.8 ± 56.2	1646.0 ± 935.6	16.66 ± 0.00
	4 x 128	SGD	35.1 ± 4.1	37.1 ± 5.3	63.0 ± 34.7	255.3 ± 140.8	4.05 ± 0.00
		Adam	49.8 ± 0.1	57.9 ± 0.3	28.6 ± 8.3	118.4 ± 34.3	4.14 ± 0.00
		DR	26.7 ± 1.1	27.4 ± 1.2	10.4 ± 6.7	234.1 ± 149.6	22.51 ± 0.00
		AP	26.5 ± 1.1	27.1 ± 1.2	182.4 ± 84.0	1688.6 ± 777.3	9.26 ± 0.00
	4 x 128 + Skip	SGD	38.9 ± 4.5	42.6 ± 6.1	129.6 ± 80.2	619.7 ± 383.6	4.78 ± 0.00
		Adam	50.2 ± 0.3	59.0 ± 0.7	15.8 ± 5.2	75.1 ± 24.7	4.75 ± 0.00
		DR	32.2 ± 0.8	33.9 ± 0.9	32.8 ± 8.4	1210.8 ± 308.3	36.91 ± 0.00
		AP	30.2 ± 0.5	31.1 ± 0.7	205.2 ± 78.3	2964.4 ± 1131.2	14.44 ± 0.01