A Proofs

Proof of Theorem 2.3 We first prove the existence of a solution $x \in \mathbb{R}^n$ to the equation $x = \phi(Ax+b)$ if $\lambda_{\text{pf}} < 1$. Since ϕ satisfies Assumption (2.1), we have that for $t \geq 1$, the picard iteration

$$x_{t+1} = \phi(Ax_t + b),\ x_0 = 0,\ t = 0, 1, \cdots$$

satisfies

$$\begin{aligned}|x_{t+1} - x_t| &= |\phi(Ax_t + b) - \phi(Ax_{t-1} + b)| \\ &\leq |A(x_t - x_{t-1})| \leq |A||x_t - x_{t-1}| \\ &\leq |A|^t |x_1 - x_0|.\end{aligned}$$

Hence, for every $t, \tau \geq 1$, we have

$$\begin{aligned}|x_{t+\tau} - x_t| &= \left|\sum_{i=t+1}^{t+\tau} (x_i - x_{i-1})\right| \leq \sum_{k=t}^{t+\tau} |A|^k |x_1 - x_0| \\ &\leq |A|^t \sum_{k=0}^{\tau} |A|^k |x_1 - x_0| \leq |A|^t \sum_{k=0}^{\infty} |A|^k |x_1 - x_0| \\ &= |A|^t (I - |A|)^{-1} |x_1 - x_0|.\end{aligned}$$

The inverse of $I - |A|$ exists as $\lambda_{\text{pf}}(|A|) < 1$. Since $\lim_{t\to\infty} |A|^t = 0$, we have

$$0 \leq \lim_{t\to\infty} |x_{t+\tau} - x_t| \leq \lim_{t\to\infty} |A|^t (I - |A|)^{-1} |x_1 - x_0| = 0.$$

We obtain that x_t is a Cauchy sequence, and thus the sequence converges to some limit point, x_∞, which by continuity of ϕ can be obtained by $x_\infty = \phi(Ax_\infty + b)$, thus establishes the existence of a solution to $x = \phi(Ax + b)$.

For uniqueness, consider two solutions $x_a, x_b \in \mathbb{R}^n_+$ to the equation, the following inequality holds,

$$0 \leq |x_a - x_b| \leq |A||x_a - x_b| \leq |A|^k |x_a - x_b|.$$

As $k \to \infty$, we have that $|A|^k \to 0$, and it follows that $x_a = x_b$, which establishes the unicity of the solution.

Proof of Theorem 2.4 Consider a neural network $\mathcal{N}$ in its equivalent implicit form $(A_\mathcal{N}, B_\mathcal{N}, C_\mathcal{N}, D_\mathcal{N}, \phi)$, since the matrix $|A_\mathcal{N}|$ is strictly upper triangular, all of its eigenvalues are zeros, automatically satisfying the PF sufficient condition for well-posedness. From the Collatz-Wielandt formula (Meyer, 2000), the PF eigenvalue of a well-posed implicit model can be represented as

$$\lambda_{\text{pf}}(|A|) = \inf_{s>0} \left\|\mathbf{diag}(s)|A|\mathbf{diag}(s)^{-1}\right\|_\infty.$$

The scaling factor s such that $\left\|\mathbf{diag}(s)|A|\mathbf{diag}(s)^{-1}\right\|_\infty < 1$ can be obtained by solving

$$s_i = 1 + \sum_{j=i+1}^{n} |A_{i,j}| s_j,\ i \in [n],$$

which can then be solved by backward substitution. The new model matrices (A', B', C', D', ϕ), are obtained by

$$\left(\begin{array}{c|c} A' & B' \\ \hline C' & D' \end{array}\right) = \left(\begin{array}{c|c} SAS^{-1} & SB \\ \hline CS^{-1} & D \end{array}\right)$$

where $S = \mathbf{diag}(s)$, with $s > 0$ a PF eigenvalue of $|A|$. More generally, provided that $\lambda_{\text{pf}}(|A|) < 1$, we simply set $s = (I - |A|)^{-1}\mathbf{1}$, which can be obtained as the limit point of fixed-point iterations.

B More on Parallel Training

Data structure. Fitting all the weight matrices into memory requires a substantial amount of storage space. However, we can leverage the high-sparsity property of the problem to reduce the memory consumption when storing the weight matrices. In the high-sparsity regime, schemes known from high-performance computing such as compressed sparse row (CSR) and compressed sparse column (CSC) can store indices of matrices, respectively. Since in this problem, we operate in a row-wise fashion, we choose to store the weight matrices in CSR format. CSR represents the indices in an $n = n_r \times n_c$ matrix using row and column index arrays. The row array is of length n_r and store the offsets of each row in the value array in $\lceil \log_2 m \rceil$ bits, where m is the number of non-zero elements. The column array is of length m and stores the column indices of each value in $\lceil \log_2 n_c \rceil$ bits. The total storage space required is therefore $n_r \times \lceil \log_2 m \rceil + m \times \lceil \log_2 n_c \rceil$.

Multiprocessing. Given state matrices from a neural network, the basis pursuit problem of (8) and (9) can be paralleled, each involving a single or a block of rows. Each block is trained independently by a child processor with an auxiliary objective, and returns the solutions back to the main processor. We implement our parallel training algorithm with the MULTIPROCESSING package using Python. The MULTIPROCESSING package[5] supports spawning processes and offers both local and remote concurrency. In Python, its Global Interpreter Lock (GIL) only allows one thread to be run at a time under the interpreter, which means we are unable to leverage the benefit of multi-threading. However, with multiprocessing, each process has its own interpreter and the instructions are executed by its own interpreter, which allows multiple processes to be run in parallel, side-stepping the GIL by using sub-processes instead of threads. In MULTIPROCESSING, a process is a program loaded into memory to run and does not share its memory with other processes. The decomposability of the training problem can be viewed as *data parallelism* where the execution of a function, i.e. solving the convex optimization problem, is parallelized, and the input values are distributed across processes. We use the `Pool` object to offer a means of defining a function in a module so that child processes can each import the module and execute it independently.

Memory sharing. In MULTIPROCESSING, data in the arguments are pickled and passed to the child processors by default. In the basis pursuit problem, the state matrix X and the input data matrix U remain unchanged during task execution across all the processors, and thus only need read-only access to X and U. Passing X and U to each processor whenever a new task is scheduled consumes a significant amount of memory space and increases the communication time. As a result, instead of treating them as data input to the function, we put X and U into a shared memory, providing direct access of the shared resources across processes.

Ray. We also implement our parallel algorithm using RAY[6], an open-source and general-purpose distributed compute framework for machine learning and deep learning applications. By transforming the execution of the convex training problems into ray `actors`, we are able to distribute the input values to multiple ray actors to run on multiple ray nodes. Similar to the memory sharing in the multiprocessing approach, we use `ray.put()` to save objects into the ray object store, saving memory bandwidth by only passing the object ids around. We run our experiments on the Cori clusters[7] hosted by National Energy Research Scientific Computing (NERSC) Center and use the SLURM-RAY-CLUSTER scripts[8] for running multi-nodes.

Performance benchmark. Figure 5 show the run-time for our serial and parallel implementation using both MULTIPROCESSING and RAY. We observe that MULTIPROCESSING

[5] https://docs.python.org/3/library/multiprocessing.html
[6] https://www.ray.io/
[7] https://docs.nersc.gov/systems/cori/
[8] https://github.com/NERSC/slurm-ray-cluster

provides the best speedup as compared to Ray. We hypothesize that since Ray is a general-purpose distributed compute framework, it contains more overhead than solving the training problem directly using multiprocessing.

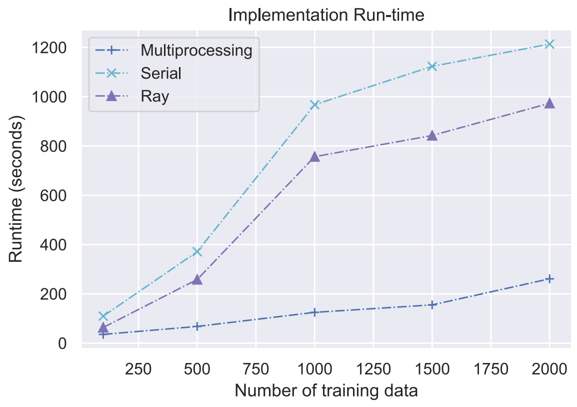

Figure 5: Performance benchmark for serial, multiprocessing (parallel), and Ray (parallel) implementation on FashionMNIST dataset using 8 processors.

C More on Numerical Experiments

Table 2 and Table 3 shows the number of training samples, hyper-parameters, and adversarial test accuracy for perspective relaxation and ℓ_1-norm objective functions with state and outputs matching penalties as in problem (7). For perceptive relaxation, we solve the following problem:

$$\min_{M,t,s} \ \alpha \sum_{ij} s_{ij} + \lambda_1 \left\| Z - (AX + BU) \right\|_F^2 \tag{10a}$$

$$+ \lambda_2 \left\| \hat{Y} - (CX + DU) \right\|_F^2 \tag{10b}$$

$$\text{s.t.} \quad (2d), \ t_{ij} \in [0,1], M_{ij}^2 \leq s_{ij} \cdot t_{ij}, \ s_{ij} \geq 0. \tag{10c}$$

For the ℓ_1-norm problem, we solve the following problem:

$$\min_{M} \quad \beta \sum_{ij} |M_{ij}| + \lambda_1 \left\| Z - (AX + BU) \right\|_F^2 \tag{11a}$$

$$+ \lambda_2 \left\| \hat{Y} - (CX + DU) \right\|_F^2 \ : \ (2d), \tag{11b}$$

where β controls the degree of regularizing for robustness.

Table 2: Experimental settings for perspective relaxation on Fashion-MNIST.

					Test Acc. (%)	
# Train Samples	**Sparsity (%)**	λ_1	λ_2	α	$\epsilon = 0.004$	$\epsilon = 0.008$
700	15	0.1	0.1	0.01	78.7	75.4
500	23	0.1	0.1	0.01	77.3	73.8
400	28	0.1	0.1	0.01	76.6	72.8
300	36	0.1	0.1	0.01	74.9	72.4
200	54	0.1	0.1	0.01	73.7	70.1
100	77	0.1	0.1	0.01	57.2	49.5

Table 3: Experimental settings for ℓ_1-norm on Fashion-MNIST.

					Test Acc. (%)	
# Train Samples	**Sparsity (%)**	λ_1	λ_2	β	$\epsilon = 0.004$	$\epsilon = 0.008$
600	20	0.1	0.1	0.001	79.6	76.2
1000	47	0.01	0.01	0.001	79.3	76.2
500	26	0.01	0.01	0.01	78.3	75.0
2000	6	0.01	0.01	0.01	77.6	74.6
900	65	0.01	0.01	0.001	74.9	69.9
400	76	0.01	0.01	0.001	72.3	68.4