Scalable ADMM for Convex Neural Networks

Miria Feng, Zachary Shah, Daniel Abraham

Abstract

The non-convex training landscape of a ReLU-activated neural network can be reformulated as a convex optimization problem. This is a significant step towards achieving globally optimal interpretable results. We examine three practical ADMM based methods for solving this reformulated problem, and examine their performance with GPU acceleration on PyTorch and JAX. In order to meliorate the expensive primal step bottleneck of ADMM, we incorporate a randomized block-coordinate descent (RBCD) variant. We also experiment with NysADMM, which treats the primal update step as a linear solve with a randomized low-rank Nystrom approximation. This project examines the scalability and acceleration of these methods, in order to encourage applications across a wide range of statistical learning settings. Results show promising directions for scaling ADMM with accelerated GPU techniques to optimize two-layer neural networks.

1 Introduction

Artificial Neural Networks (ANN) are one of the most popular machine learning tools, however optimizing these networks with non-linear activation functions requires solving extensive non-convex optimization problems. Currently the tremendous empirical success of ANN based applications are trained with variants of back-propagation and stochastic gradient descent (SGD). Despite this, very little is understood about the non-convex optimization landscape of neural networks, which lack global convergence guarantees and often suffer from vanishing or exploding gradients. Additionally, back-propagation is extremely sensitive to both initialization and hyperparameter tuning, with very few heuristics present to suggest a dependable paradigm for hyperparameter selection.

Recent work ([1], [2], [3]) has successfully leveraged a convex framework for training two-layer neural network models with ReLU activation functions. This is achieved via a convex reformulation of the training objective, and was derived by [1] using duality theory to show that two-layer neural networks with ReLU activations and weight decay regularization may be re-expressed as a linear model with a group one penalty and polyhedral cone constraints. This is significant since within a convex framework, there exists the possibility of arriving at globally optimal interpretable results. To practically solve this convex optimization problem, this project examines varying approaches based on the Alternating Direction Method of Multipliers (ADMM) [4]. ADMM offers several attractive advantages, such as its robustness against hyperparameter selection, linear decomposability for distributed optimization, and immunity to vanishing/exploding gradients. Though the performance of ADMM is promising, there are still practical challenges to overcome: namely the scalability of data size and acceleration in solving machine learning tasks. This project seeks to make progress in overcoming these disadvantages by examining several modified ADMM-based strategies to solve the binary classification problem on CIFAR-10. Our main contributions are as follows:

- Deployment of a Nystrom Preconditioner with conjugate gradient to solve the linearized bottleneck primal update step of ADMM. Comparison in performance with Randomized Block Coordinate Descent (RBCD [5]).
- Evaluation on the effect of condition number of these methods, as well as stability, scalability, and solve time.
- Acceleration of these methods on a GPU, with both PyTorch and JAX [6] experiments.

2 Problem Background

2.1 Convex Reformulation of Two-Layer ReLU Networks

In the following sections we will refer to the multi-layer perceptrons activated by rectified linear units as ReLU-MLPs. The proposed non-convex formulation of a two-layer ReLU-MLP with weights $w_j^{(1)}$, $w_j^{(2)}$ is then given by:

$$f_{\text{NCVX-MLP}}(x) = \sum_{j=1}^{m} \left(x^T w_j^{(1)} \right)_+ w_j^{(2)} \tag{1}$$

Here, $(\cdot)_+$ denotes the ReLU activation function, which selects for the positive part of the argument. To train such a network, we usually try to optimize the following regularized non-convex training objective:

$$\min_{w_j^{(1)}, w_j^{(2)}} l(\sum_{j=1}^{m} (Xw_j^{(1)})_+ w_j^{(2)}, y) + \frac{\beta}{2} \sum_{j=1}^{m} ||w_j^{(1)}||_2^2 + (w_j^{(2)})^2 \tag{2}$$

where β is the ℓ-2 regularization hyper-parameter. As shown in [1], the convex re-formulation enumerates all P ReLU activation patterns on the training data matrix $X \in \mathbb{R}^{n \times d}$, where P is bounded exponentially by the rank of X. These P activation patterns act as separating hyperplanes which essentially multiply the rows of X by 0 or 1, and hence can be denoted by diagonal matrices $\{D_i\}_{i=1}^P \in \{0,1\}^{n \times n}$. Then, the convex reformulation takes the form

$$\begin{aligned} \min_{(v_i, w_i)_{i=1}^P} & \ \ell(\sum_{i=1}^{P} D_i X(v_i - w_i), y) + \beta \sum_{i=1}^{P} ||v_i||_2 + ||w_i||_2 \\ \text{s.t.} & \quad (2D_i - I)Xv_i \geq 0, (2D_i - I)Xw_i \geq 0, i \in [P] \end{aligned} \tag{3}$$

Here, ℓ is some convex loss function. As shown in [2], the optimal weights $w^{(1)\star}, w^{(2)\star}$ of (1) can be recovered as a simple rescaling on the sparse selected $v^\star, w^\star$ of (3). In practice, a subset $P_s < P$ of the ReLU activation patterns are randomly sampled to be learned during training. Specifically, we set $D_i = \textbf{diag}(Xh_i \geq 0)$ for $h_i \in \mathcal{N}(0, I)$, $i \in \{1, \ldots, P_s\}$.

Our project is an extension of the ADMM approach proposed by [7]. Following the notation in [7], we first denote each sampled training feature set as $F_i = D_i X$, with the corresponding constraint map $G_i = (2D_i - I)X$. Then, we group the primal variables u, v, and introduce slack variables s:

$$\begin{aligned} u &:= \begin{bmatrix} u_1^\top & \cdots & u_{P_s}^\top & z_1^\top & \cdots & z_{P_s}^\top \end{bmatrix}^\top \\ v &:= \begin{bmatrix} v_1^\top & \cdots & v_{P_s}^\top & w_1^\top & \cdots & w_{P_s}^\top \end{bmatrix}^\top \\ s &:= \begin{bmatrix} s_1^\top & \cdots & s_{P_s}^\top & t_1^\top & \cdots & t_{P_s}^\top \end{bmatrix}^\top \end{aligned}$$

We look specifically at MSE loss: $\ell(\hat{y}, y) = \frac{1}{2}||\hat{y} - y||_2^2$, which results in the problem:

$$\min_{v,s,u} ||Fu - y||_2^2 + \beta||v||_{2,1} + \mathbb{I}_{\geq 0}(s) \quad \text{s.t.} \begin{bmatrix} I_{2dP_s} \\ G \end{bmatrix} u - \begin{bmatrix} v \\ s \end{bmatrix} = 0, \tag{4}$$

where $||.||_{2,1}$ is a mix of ℓ-2 and ℓ-1 norm to promote group sparsity in the optimal activation patterns; y is the data labels; $\mathbb{I}_{\geq 0}(s)$ forces each slack to be positive and imposes an infinite penalty otherwise; $F = [F_1 \ \cdots \ F_P \ -F_1 \ \cdots \ -F_P]$, and $G = \text{blkdiag}(G_1, \cdots, G_P, G_1, \cdots, G_P)$. One can now derive the Lagrangian

$$\mathcal{L}(u, v, s, \nu, \lambda) = \ell(Fu, y) + \beta||v||_{2,1} + \mathbb{I}_{\geq 0}(s) + \frac{\rho}{2}(||u - v + \lambda||_2^2 - ||\lambda||_2^2) + \frac{\rho}{2}(||Gu - s + \nu||_2^2 - ||\nu||_2^2) \tag{5}$$

with the dual variables

$$\lambda := [\lambda_{11} \ \cdots \ \lambda_{1P} \ \lambda_{21} \ \cdots \ \lambda_{2P}]^\top, \nu := [\nu_{11} \ \cdots \ \nu_{1P} \ \nu_{21} \ \cdots \ \nu_{2P}]^\top$$

Using (5), ADMM updates are computed to solve optimality to tolerance ϵ as shown in Algorithm 1.

A natural issue with Algorithm 1 is the large linear solve of the primal u update. To naiively compute this update requires solving a $(2dP_s \times 2dP_s)$ linear system, which is computationally infeasible at large problem dimensions. Accordingly, we will consider three iterative methods to solve the large linear system in Algorithm 1 in a scalable and accelerated manner, as listed below.

2.2 Randomized Block Coordinate Descent (RBCD)

To approximate the u update at each ADMM iteration, Randomized Block Coordinate Descent (RBCD), as outlined by [8], only computes gradients for blocks of u, namely blocks of size $2d$. While allowing for scalable data dimensions, RBCD suffers from a convergence rate that is similar to SGD. The RBCD updates are shown in Algorithm 2. Since the RBCD gradient calculations are much cheaper, we implement a backtracking line search to compute the step size γ_r at each iteration.

2.3 Conjugate Gradient (CG) with basic Preconditioner (PCG)

The Conjugate Gradient (CG) method is a strong alternative to gradient descent for solving linear systems, which can tackle large scale problems. CG is used to iteratively solve the linear system $Au^{k+1} = b$ in an efficient manner. We also examine the Preconditioned Conjugate Gradient (PCG) with a diagonal preconditioner (diagPCG), where $M = \textbf{diag}(1/A_{11}, \ldots, 1/A_{2dP_s, 2dP_s})$. This can be formed without ever needing to compute A by considering the repeat-structure of A as blockwise samplings of the rows of $X^T X$.

Algorithm 1 ADMM MSE Loss Updates

```
repeat
    $b^k \leftarrow \frac{1}{\rho}F^\top y + v^k - \lambda^k + G^\top s^k - G^\top \nu^k$
    Solve $Au^{k+1} = b^k$   for $A = I + \frac{1}{\rho}F^T F + G^T G$          ▷ Primal u update
    $v^{k+1} \leftarrow \mathbf{prox}_{\frac{\beta}{\rho}\|\cdot\|_2}(u^{k+1} + \lambda^k)$          ▷ Primal v update
    $s^{k+1} \leftarrow (Gu^{k+1} + \nu)_+$          ▷ Slack s update
    $\lambda^{k+1} \leftarrow \lambda^k + \frac{\gamma_\alpha}{\rho}(u^{k+1} - v^{k+1})$          ▷ Dual $\lambda$ update
    $\nu^{k+1} \leftarrow \nu^k + \frac{\gamma_\alpha}{\rho}(Gu^{k+1} - s^{k+1})$          ▷ Dual $\nu$ update
until
     $u^k - v^k \leq \epsilon$
     $Gu^k - s^k \leq \epsilon$          ▷ Primal Optimality Conditions
     $F^T(Fu^k - y) + \rho(\lambda^k + G^T \nu^k) \leq \epsilon$
     $\frac{v_i^k}{\|v_i^k\|_2} + \rho\lambda_{1i}^k \leq \epsilon, \quad i = 1, \ldots, P_s$
     $\frac{w_i^k}{\|w_i^k\|_2} + \rho\lambda_{2i}^k \leq \epsilon, \quad i = 1, \ldots, P_s$          ▷ Dual Optimality Conditions
```

Algorithm 2 Randomized Block Coordinate Descent

```
$\hat{y} \leftarrow Fu$
$\tilde{s} \leftarrow G^\top(s - \nu_1)$
$\tilde{t} \leftarrow G^\top(t - \nu_2)$
repeat
    $\tilde{y} \leftarrow \nabla_{\hat{y}}\ell(\hat{y}, y)$
    Uniformly select $i$ from $[P]$ at random;
    $u_i^+ \leftarrow u_i - \gamma_r F_i^\top \tilde{y} - \gamma_r \rho(u_i - v_i + \lambda_{1i} + G_i^\top G_i u_i - \tilde{s}_i)$
    $z_i^+ \leftarrow z_i + \gamma_r F_i^\top \tilde{y} - \gamma_r \rho(z_i - w_i + \lambda_{2i} + G_i^\top G_i z_i - \tilde{t}_i)$
    $\hat{y}^+ \leftarrow \hat{y} + F_i((u_i^+ - z_i^+) - (u_i + z_i))$
```

2.4 Nystrom-PCG (nysPCG)

In order to speed up PCG, we investigate the use of the Nystrom Preconditioner [9]. This method leverages a low-rank formulation to derive a general preconditioner that works well across different problem domains. Since MLP classification robust across many problem domains, it is important that the Preconditioner is robust in the same way, hence the choice of the Nystrom Preconditioner for Conjugate Gradient (nysPCG). We form the Nystrom Preconditioner M_{nys} with a rank of 100 (which lowers the condition number by a factor of 100 as shown by Figrure 2) and use Nystrom PCG to solve our primal update according to Algorithms 4 and 5 of [9], respectively. As in CG, we can compute M_{nys} and run Nystrom PCG without ever needing to explicitly compute A.

2.5 JAX Acceleration

To further support the scalability of ADMM for large problem dimensions, we aim to leverage GPU acceleration for our optimizers due to their success in solving large linear systems. We completed our base optimizer implementations in Pytorch, which allows for ease of access to GPUs. In order to further accelerate our optimization procedure we implement JAX as a proof of concept for the RBCD method. Google JAX is a high performance numerical computing library with incorporated composable function transforms. It features automatic differentiation which allows us to compute function gradients without explicitly writing the derivative code. It is also designed to run efficiently across GPU, CPU, and TPU with just-in-time compilation approach to optimize computations for the target hardware. This results in extremely attractive speedups for numerical computation, as shown in [10], where JAX performed 86 times faster than NumPy on the task of calculating sum of matrix powers. As a result, both DeepMind and OpenAI have adapted JAX to accelerate research, making it a promising tool in optimization.

3 Experiments

3.1 Baseline Comparison of ADMM solvers

In order to characterize the quality of ADMM based training algorithms, we first implement our ADMM solvers on a small-scale binary classifcation problem solved on CPU. We take a subset of $n = 1000$ images for the first two classes of CIFAR-10, with features downsampled to $d = 100$, and observe the performance of our methods for $P_s = 10$ in Figure 1.

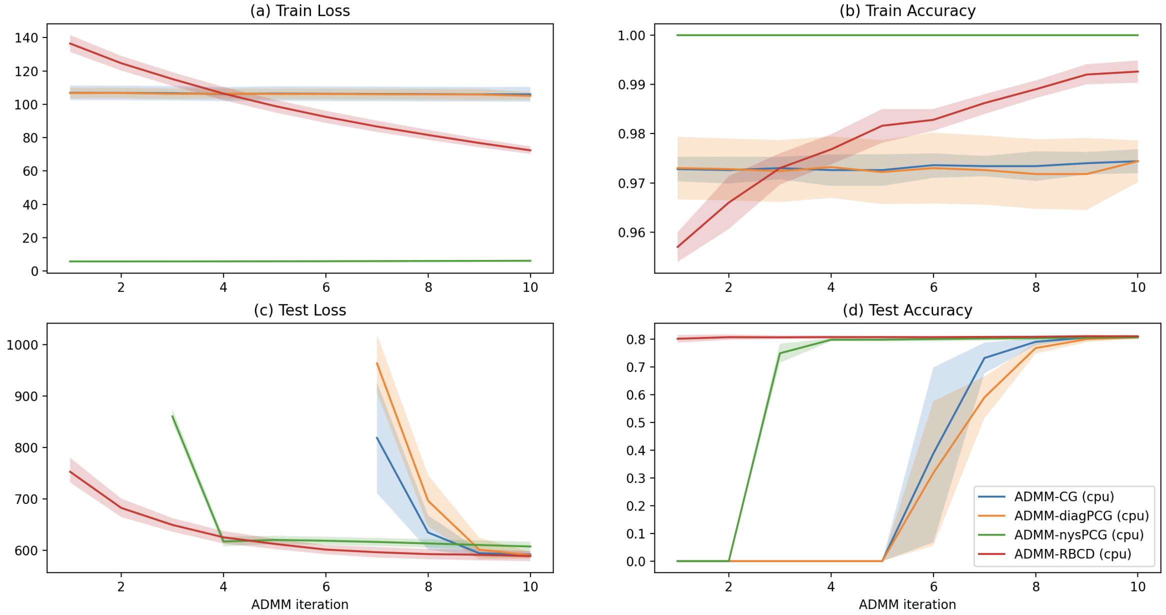

Figure 1: Approximate ADMM Methods on CIFAR-10 2-Digit Classification Problem

From Figure 1, we observe that nysPCG reaches better loss and accuracies compared to the alternative conjugate gradient methods. This is because nysPCG lowers the condition number of A significantly (in fact, the diagonal preconditioner increases the condition number of A, as shown in Figure 2). We also observe that RBCD takes more iterations to reach similar levels of test loss and accuracy as nysPCG, but the cost of each iteration is much lower. Moving forward, we chose to only implement these two methods at larger scales.

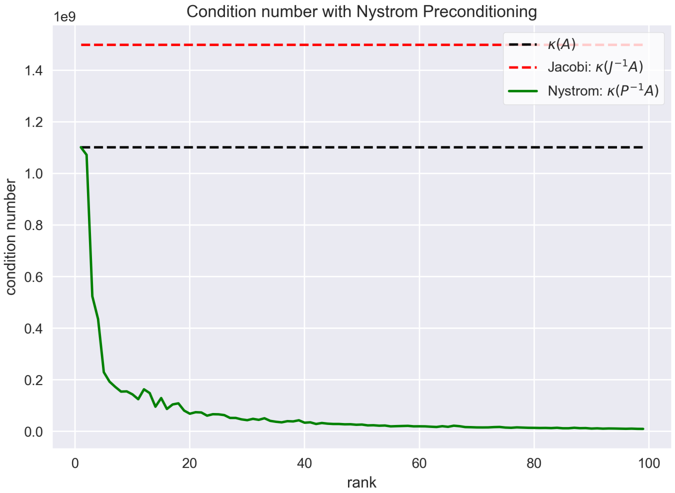

Figure 2: Condition number of A as a function of the rank of the Nystrom approximation

3.2 Training at Scale for NysPCG and RBCD vs Adam-MLP

We now compare the convex ADMM-RBCD and ADMM-nysPCG methods at a larger scale for classifcation on CIFAR-10 ($n = 10000$, $d = 3072$, $P_s = 50$). To do a Cholesky solve of our primal update at this scale would require computing and storing all of A, which would be over 90 GiB, exceeding the memory size for most modern GPUs. As a baseline, we compare to a non-convex MLP trained via backpropagation with an Adam optimizer. We now show implementations of RBCD, PCG, and MLP-Adam wth acceleration via PyTorch on the GPU in Figure 3.

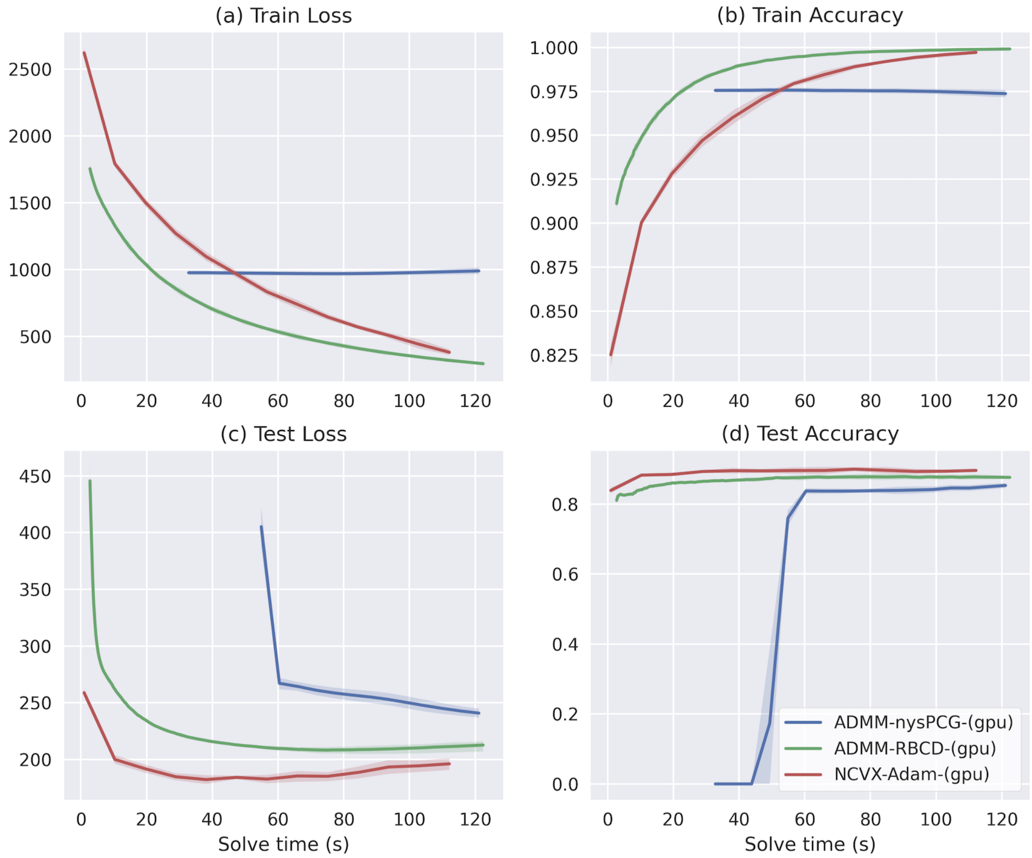

Figure 3: Approximate ADMM Methods on CIFAR-10 Classifcation

3.3 JAX Acceleration for RBCD Method

Since our RBCD solver scaled the best at higher dimensions, we also demonstrate further acceleration of RBCD with a JAX framework in Figure 4.

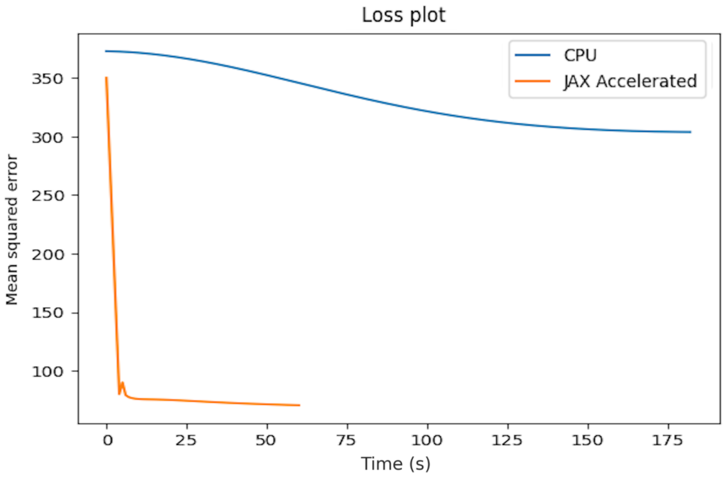

Figure 4: Training Loss for RBCD with JAX Acclereration on CIFAR-10 Classification

Notably, the JAX implementation shows the most significant improvement in speed, which is consistent with our expectations.

Below, we also summarize the runtimes for our methods on GPU reach at least 97% train accuracy for the problem in section 4.2. We also compare this to CPU and JAX for RBCD.

Table 1: Solve time to 97% Train Accuracy

Method	Total (s)	Precomp.(s)	u updates (s)	v updates (s)	dual updates (s)
nysPCG (GPU)	39.484	16.321	23.119	0.043	0.001
RBCD (GPU)	23.502	0.000	23.136	0.361	0.005
RBCD (JAX-GPU)	**16.243**	0.000	16.118	0.125	0.000
RBCD (CPU)	185.896	0.523	181.689	3.663	0.020
Adam (GPU)	55.129	n/a	n/a	n/a	n/a

4 Discussion

Both Figure 3 and Table 1 show that the RBCD method has the most desirable tradeoffs for approximating the full ADMM step. The speed is much closer to the non-convex Adam-MLPs, while the test accuracy is far more stable. The PCG method struggles with convergence, and suffers from very long CPU run-time due to heavy precomputations required to form the Nyquist preconditioner in our implementation.

Additionally, in Figure 3, we observe that the NysPCG method does not achieve a decreasing training loss or accuracy with further iterations at higher scale. This is likely because we did not scale the number of CG iterations with the increase in the problem dimension, preventing CG from converging in further iterations.

We also observe that the RBCD method achieves the best train loss and accuracy out of all methods. However, it fails to generalize well in comparison to the Adam optimizer. This is likely due the minimum number of hyperplane samples in our training scheme, which were not tuned to best recover the equivalent optimal weights for the non-convex formulation (used to evaluate unseen test data). Training with higher P_s would likely combat this issue.

The significant improvement in speedup observed with the JAX RBCD method is both extremely promising, and consistent with existing research in scientific computing. In our implementation the acceleration is largely due to just-in-time (JIT) compilations using Accelerated Linear Algebra (XLA). Since JAX is fundamentally built on XLA, which is specifically designed for linear algebra, this raises the computational speed ceiling by several orders of magnitude in comparison to NumPy. The key challenge being adding JIT computational function transforms with consideration of low level code. This is since JAX transforms only work with numerical functions, therefore untracked side-effects can easily throw off accuracy of intended computations. Another disadvantage to JAX is its higher barrier to entry (versus PyTorch), resulting in significant overhead when attempting to implement equivalent methods from Python.

5 Conclusion

The successful application of ADMM based approaches to solve the convex reformulation of a shallow neural network problem takes a small step towards achieving more efficient, elegant, and globally optimal solutions. We examine the problem of binary classification on the CIFAR-10 dataset, and experiment with several ADMM modifications to improve both scalability and acceleration. Most notably, the RBCD method gave the best performance results with respect to training loss, and the JAX RBCD implementation demonstrated a stunning speed up in terms of performance. Promising directions for future work includes: scaling the problem up to ImageNet classification, incorporating the Nystrom Preconditioner into the RBCD method, and adding JAX acceleration to more methods while taking advantage of JAX's ability to automatically parallelize code across multiple CPUs, GPUs and TPUs.

6 Appendix

All code used to generate figures can be found at the public repo: https://github.com/zachary-shah/admmNN/

References

[1] Mert Pilanci and Tolga Ergen. Neural networks are convex regularizers: Exact polynomial-time convex optimization formulations for two-layer networks. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 7695–7705. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/pilanci20a.html. 1, 2

[2] Tolga Ergen and Mert Pilanci. Revealing the structure of deep neural networks via convex duality. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 3004–3014. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/ergen21b.html. 1, 2

[3] Aaron Mishkin, Arda Sahiner, and Mert Pilanci. Fast convex optimization for two-layer relu networks: Equivalent model classes and cone decompositions, 2022. 1

[4] Stephen Boyd. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Found. Trends® Mach. Learn.*, 3(1):1–122, 2010. 1

[5] Zhaosong Lu and Lin Xiao. On the complexity analysis of randomized block-coordinate descent methods. *Mathematical Programming*, 152:615–642, 2015. 1

[6] Umberto Ravaioli, Kyle Dunlap, and Kerianne Hobbs. A universal framework for generalized run time assurance with jax automatic differentiation. *arXiv preprint arXiv:2209.01120*, 2022. 1

[7] Yatong Bai, Tanmay Gautam, and Somayeh Sojoudi. Efficient global optimization of two-layer ReLU networks: Quadratic-time algorithms and adversarial training. January 2022. 2

[8] Yatong Bai, Tanmay Gautam, and Somayeh Sojoudi. Efficient global optimization of two-layer relu networks: Quadratic-time algorithms and adversarial training. *arXiv preprint arXiv:2201.01965*, 2022. 2

[9] Shipu Zhao, Zachary Frangella, and Madeleine Udell. Nysadmm: faster composite convex optimization via low-rank approximation. In *International Conference on Machine Learning*, pages 26824–26840. PMLR, 2022. 3

[10] Alexey Kurakin, Shuang Song, Steve Chien, Roxana Geambasu, Andreas Terzis, and Abhradeep Thakurta. Toward training at imagenet scale with differential privacy. *arXiv preprint arXiv:2201.12328*, 2022. 3