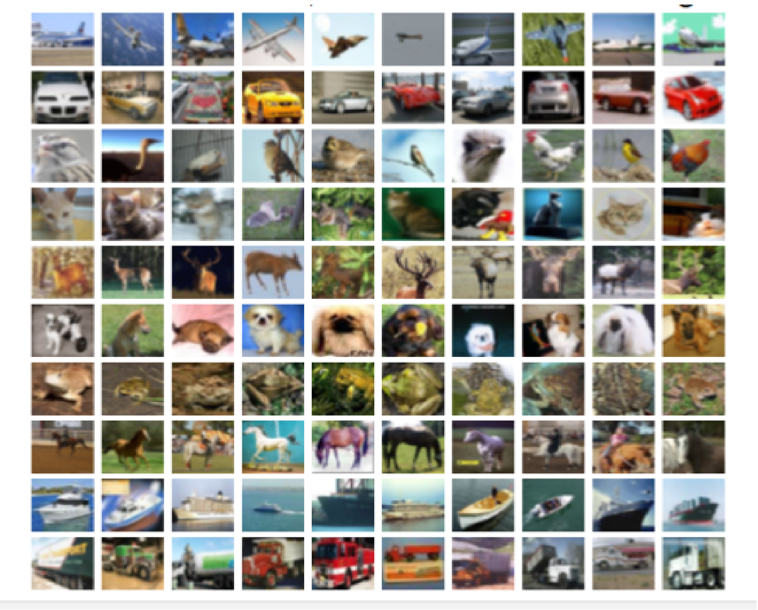


University of Colorado at Boulder

APPM: 5630

Advanced Convex Optimization

Approximate Hessian Based ARC for Deep Learning

Authors:	*Affiliations:*
Cooper Simpson	Applied Mathematics
Jaden Wang	Applied Mathematics

April 26, 2021

Contents

1 Introduction

First-order, gradient-based optimization methods are well-known and widely used. In the field of deep learning they essentially comprise the entirety of practical methods. These methods are desirable for their cheap computational costs, but can suffer from slow convergence and require tuning various hyperparameters. This tuning itself can be a costly endeavour and requires domain experience. In the case of non-convex optimization, first-order methods suffer from an additional drawback: they cannot differentiate saddle points or flat regions from local minima and often terminate prematurely. Stochastic Gradient Descent (SGD) and its variants are first-order methods that have been widely adopted in the field of deep learning. Given that most deep learning problems are non-convex and often riddled with saddle points all these drawbacks are particularly highlighted.

Second-order methods, on the other hand, can exploit curvature information from the Hessian allowing them to move past saddle points and often yield superior solutions in optimization problems. Among these methods Trust Region is fairly well-known, but more recently researchers have been favoring Adaptive Regularization with Cubics (ARC) [CGT11] due to its reasonable assumptions and better performance in many problems. However, for problems from deep learning – which are very high-dimensional – computing and extracting information from the Hessian can be prohibitively expensive. As a result, we have yet to see wide adoption of any second-order methods, particularly ARC, in this field. This begs the question: is there a way to circumvent the costs associated with the Hessian in order to leverage the benefits of second-order optimization schemes in deep learning problems?

In the recent paper [XRM20], the authors theoretically proved that the Hessian can be subsampled considerably to reduce the cost of methods such as Trust Region and ARC. Moreover, they empirically demonstrated that their Trust Region implementation with inexact Hessian achieves comparable if not superior performance compared to SGD on common deep learning problems [XRM18]. If their claims about Trust Region can be generalized, this could have profound implications in advancing deep learning via improved training. Missing from their experiments are ARC's performance on the same problems and whether the Trust Region claims also hold true for ARC. If so, this is desirable because ARC tends to perform better than Trust Region with similar cost. In this paper, we aim to reexamine ARC as a feasible algorithm for deep learning optimization.

2 Theoretical Background

We begin by outlining the mathematical background for Trust Region, ARC, and the approximation methods used to make them computationally tractable for large-scale problems.

2.1 Trust Region

Trust Region uses a model function $m(x)$, often quadratic, to approximate the objective function in a region where the model function is a good fit, hence the name. This region is determined by a ball of radius Δ centered at x, and the radius is adaptively updated at each iteration based on an evaluation score ρ, the ratio of actual descent vs model-predicted descent. If $\rho > 1-\epsilon$, that means we achieve comparable or better descent result than what the model predicted, suggesting that we can keep the same radius or increase it for faster convergence. If $\rho \ll 1$, then the model predicts the function poorly and we need to shrink the radius until we achieve good prediction again (or until we can "trust" the region again).

Trust Region can be intuitively understood as the dual of line search. While line search requires choosing a direction first and then solving for the optimal step-length, Trust Region requires specifying a radius first and then finds the direction that approximately minimizes the model function in that region. Line search is naturally suited for gradient methods since we *a priori* know the search direction must be the negative gradient direction. However, in higher-order methods, this search direction is not given to us, making Trust Region a better iterative scheme for the solver. Although we are not going to use the Trust Region scheme explicitly in this paper, the adaptive algorithm we are about to introduce is heavily inspired by it.

2.2 ARC

Recall that the second-order Taylor expansion of a smooth function $f : \mathbb{R}^n \to \mathbb{R}$ about x_k yields

$$f(x) = f(x_k) + \nabla f(x_k)(x - x_k) + \frac{1}{2}(x - x_k)^T \nabla^2 f(x_k)(x - x_k) + R(x) \tag{1}$$

where the remainder $R(x) = \mathcal{O}((x - x_k)^3)$. If we can reasonably ignore higher-order terms and apply Fermat's rule at each iteration, this leads to Newton's method that super-linearly converges to first-order stationary points:

$$0 = \nabla f_k + \nabla^2 f_k (x_{k+1} - x_k) \tag{2}$$

$$x_{k+1} = x_k - \nabla^2 f_k^{-1} \nabla f_k \tag{3}$$

However, the biggest drawback is that it cannot distinguish different types of first-order stationary points and often converges to unwanted saddle points and local maxima. Another issue that arises in implementation is that it requires Hessian inversion which might not exist or is too expensive to compute for high-dimensional problems.

Recall that by assuming Lipschitz continuous gradient, we derive Gradient Descent by bounding the first-order Taylor expansion with a quadratic *majorizer*:

$$f_{k+1} := f(x_k + s_k) \leq f_k + \langle \nabla f_k, s_k \rangle + \frac{L}{2}\|s_k\|^2 \tag{4}$$

An improvement of Newton's method analogous to the derivation of Gradient Descent can be made by assuming Lipschitz continuous Hessian:

$$\|\nabla^2 f_{k+1} - \nabla^2 f_k\| \leq L\|x_{k+1} - x_k\| \tag{5}$$

We can give an upper-bound for the new function value when a step s_k is taken by using the Lagrange error bound:

$$f_{k+1} \leq f_k + \langle \nabla f_k, s_k \rangle + \frac{1}{2}\langle s_k, \nabla^2 f_k s_k \rangle + \frac{L}{6}\|s_k\|^3 \tag{6}$$

Just as in Gradient Descent, this upper-bound is a local *majorizer* of the objective function. Noting that the Lagrange error bound has introduced a cubic regularization term into the second-order model we refer to methods that exploit this upper-bound as *cubic regularization methods*, first popularized by [NP06]. To achieve descent on the objective f with super-linear convergence, it suffices to find the s_k that minimizes the model function (6). Notably, this cubic regularization method is guaranteed to converge to a local minimum, fixing a fatal flaw of Newton's method. However, in practice it still suffers from expensive computational cost and the need to know the Lipschitz constant. It turns out we can approximate the RHS with much weaker assumptions and thus reduce the cost by introducing an adaptive parameter σ instead of the global Lipschitz constant. That is, at each iteration, we aim to solve the new *subproblem*:

$$\min_{s\in\mathbb{R}^d} g_k^T s + \frac{1}{2}s^T B_k s + \frac{\sigma_k}{3}\|s\|^3 \tag{7}$$

where g_k is the gradient and B_k is an approximation of the Hessian that satisfies certain accuracy conditions. Here σ_k serves to absorb the inaccuracies from both upper-bounding the second-order Taylor expansion of the objective function and the Hessian approximation. A remarkable consequence of replacing L with σ is that we no longer need the global Lipschitz assumption, allowing the algorithm to work on a wider range of "not-so-nice" functions. In addition to weighting the regularization term, σ can also be intuitively understood as the inverse of a "soft" trust region radius. If the model predicts the function poorly and σ becomes very large, then the solution to the subproblem (7) is likely to be a small step, somewhat mimicking searching within a small trust region radius. Therefore, it is natural to use an adaptive scheme similar to Trust Region but with the hard step-length constraint removed. This adaptive version of cubic regularization method is named *Adaptive Regularization with Cubics*, or ARC, in the paper [CGT11].

We do not want to solve (7) exactly since it requires exact eigen-decomposition of the approximated Hessian via Cholesky, which is $\mathcal{O}(n^3)$ and rather expensive for large-scale problems. Instead, we impose conditions on approximated solutions to ensure we achieve sufficient descent at each iteration for convergence. When we descend on the objective function in a neighborhood around a current point, we know that the gradient direction yields the steepest descent near that point. However, if the function surface is curved, the gradient direction may quickly deviate from the optimal descent path due to the curvature. Intuitively we see that the direction where the function curves downward the most is vital for descent in the neighborhood and is roughly given by the eigenvector associated with the smallest eigenvalue, or the *leftmost eigenvector.* If the eigenvalue is negative [1]. Therefore, the best descent step can be approximated by a linear combination of the gradient direction and the leftmost eigenvector direction. That is, we require the step to descend no less than both the minimizer along the negative gradient direction, namely the *Cauchy point* s^C, and the minimizer along the leftmost eigenvector direction, namely the *eigen point* s^E. To satisfy these conditions, it suffices to search for a minimizer within the entire two-dimensional subspace $S := \text{span}\{s^C, s^E\}$. That is we restrict the search space form $\mathcal{R}^d$ to S

$$\min_{s \in S} g_k^T s + \frac{1}{2} s^T B_k s + \frac{\sigma_k}{3} \|s\|^3 \tag{8}$$

This means that once we have both the gradient and the leftmost eigenvector, we can construct a $d \times 2$ orthogonal basis. If we let Q be an orthogonal basis of the two-dimensional subspace S, then via a change of basis on (7) we obtain a two-dimensional, unconstrained, non-convex problem:

$$v_k = \underset{v \in \mathbb{R}^2}{\operatorname{argmin}} \langle Q^T g_k, v \rangle + \frac{1}{2} \langle v, Q^T B_k Q v \rangle + \frac{\sigma_k}{3} \|v\|^3 \tag{9}$$

Then we can find the minimizer v_k of this two-dimensional subproblem cheaply using any accurate solver for non-convex problems, and the approximated solution to the original subproblem is just $s_k = Q v_k$.

2.3 Krylov subspace method

It remains to find the leftmost eigenvector of the Hessian. In neural networks, the dimension of the parameters d is typically too large to compute the eigenvector efficiently via exact methods. Thus, to make ARC practical, we need to substantially reduce the dimension of the Hessian. A common way to achieve this is through the use of a Krylov subspace method. Recall from lectures that Conjugate Gradient is a type of Krylov subspace method suitable for solving large-scale linear

[1] If the smallest eigenvalue is non-negative, then the function is locally convex and we know that a second-order critical point is in the neighborhood, and ARC should converge to it quadratically just like Newton's method.

equations $Ax = b$, where A is positive-semidefinite. In Conjugate Gradient, we start with a vector p_0 and gradually build up conjugate vectors that are A-orthogonal to each other and form a basis. It has the advantage of only requiring the matrix-vector product and converging in at most n steps to the optimal solution. However, for indefinite Hessian it doesn't work very well, since it cannot necessarily pick out the eigen point. The Lanczos method is another Krylov subspace method that generalizes Conjugate Gradient to indefinite systems. Just like Conjugate Gradient, Lanczos only requires computing the Hessian-vector product instead of the full Hessian. In the realm of neural networks this is highly advantageous as computing a hessian-vector is much more practical than computing the full Hessian, and it only requires the use of automatic differentiation.

Our goal is to use Hessian-vector products to construct an orthogonal basis for the p-dimensional Krylov subspace $\mathcal{K} = \text{span}\{q_0, Bq_0, B^2q_0, \ldots, B^{p-1}q_0\}$ where B is the approximated Hessian. This way we obtain a tri-diagonal symmetric matrix T along the way that approximates B under the new basis. Given an initial vector q_0, instead of the Conjugate Gradient scheme, we apply Gram-Schmidt to orthogonalize q_0 and the sequence of $(B^k q_0)$. The coefficients we collect from Gram-Schmidt yields the following tridiagonal matrix:

$$T = \begin{pmatrix} \alpha_1 & \beta_1 & 0 & \cdots & 0 \\ \beta_1 & \alpha_2 & \beta_2 & \cdots & 0 \\ 0 & \beta_2 & \alpha_3 & & \vdots \\ \vdots & \vdots & & \ddots & \beta_{p-1} \\ 0 & 0 & \cdots & \beta_{p-1} & \alpha_p \end{pmatrix}$$

We can stop this process when sufficient accuracy is achieved at iteration p. Notice that $T \approx Q^T BQ$, thus T and B share the rank-p eigenvalues. Diagonalizing T is incredibly cheap and the top-p principal eigenvalues of B lie on the diagonal. Once we obtain the approximated smallest eigenvalue, finding the approximated leftmost eigenvector becomes a problem of solving the linear equation $(B - \lambda I)v = 0$. In practice, a variant of Lanczos method has been highly optimized by the Fortran `ARPACK` routine for large-scale sparse problems [LSY98], so we simply call the routine instead of reinventing the wheel.

Interestingly, the Lanczos implementation described in [XRM20] doesn't explicitly find the leftmost eigenvector. Instead, T is used to solve the subproblem (7) in the p-dimensional Krylov subspace $\mathcal{K}$, serving as the approximated Hessian in that subspace. Then the subproblem becomes

$$v_k = \underset{v \in \mathbb{R}^p}{\text{argmin}} \langle Q^T g_k, v\rangle + \frac{1}{2}\langle v, Tv\rangle + \frac{\sigma_k}{3}\|v\|^3 \tag{10}$$

The rationale is that the solution to (10) should satisfy the sufficient descent conditions as long as $S \subset \mathcal{K}$. That is, the approximated leftmost eigenvector is implicitly contained in the subspace we are searching, so the minimizer of (10) should beat the eigen point. However, in practice this implementation might not work well, as [XRM18] reported that they were not able to achieve expected performance from ARC in deep learning problems. Compared to the two-dimensional search in the explicit method, the implict method searches in a p-dimensional space, making it more expensive and difficult to find an accuracy minimizer. Additional numerical issues might arise from hand-coding the Lanczos method, which notoriously tends to lose orthogonality due to numerical instability unless abundant care is taken in the implementation.

We shall refer to the explicit eigenvector method as ARC-EE and the implicit method as ARC-IE. Now we present the final algorithm ARC-EE that relies on stochastic subsampling and Krylov subspace approximation to scale the original ARC:

Algorithm 1: ARC-EE

```
Input: Starting point $x_0$, initial adaptive parameter $\sigma_0 \gg \sigma_{\min} > 0$, hyper-parameters
       $\varepsilon_g, \varepsilon_H, \varepsilon, \eta_1 < \eta_2 \in (0,1), \gamma > 1$
for $k = 0, 1, \ldots$ do
    Set the approximated Hessian, $B_k$, according to (16)
    if $g_k \leq \varepsilon_g$, $\lambda_{\min}(B_k) \geq -(\varepsilon_H + \varepsilon)$ then
        Return $x_k$
    end if
    Solve the subproblem (7) approximately via Algorithm 2
    if $\rho_k \geq \eta_2$ then
        $x_{k+1} = x_k + s_k$
        $\sigma_{k+1} = \max\{\sigma_k/\gamma, \sigma_{\min}\}$
    else if $\rho_k \geq \eta_1$ then
        $x_{k+1} = x_k + s_k$
    else
        $\sigma_{k+1} = \gamma\sigma_k$
    end if
end for
Output: $x_k$
```

Algorithm 2: ARC subproblem with explicit eigenvector

```
Input: Gradient $g_k$, Hessian-vector product function $\tilde{B}_k(x)$, hyper-parameters: Lanczos
       termination tolerance $\varepsilon_l$
Set $q = \frac{g_k}{\|g_k\|}$, $\varepsilon_l = \max\{\varepsilon_l, \varepsilon_l \|g_k\|\}$
for $i = 1, 2, \ldots$ do
    Set the $i$th column of $Q$, $Q_i = q$
    $z = \tilde{B}_k(q)$
    $\alpha = \langle z, q \rangle$
    Set the diagonal entry of $T$, $T_{i,i} = \alpha$
    Set $r$ to be the remainder vector after performing Gram-Schmidt on $z$ using $Q$
    $\beta = \|r\|$
    if $\beta < \varepsilon_l$ then
        Break
    end if
    Set the off-diagonal entries of $T$, $T_{i,i+1} = T_{i+1,i} = \beta$
    $q = r/\beta$
end for
Find the smallest eigenvalue $\lambda$ after diagonalizing $T$ and the leftmost eigenvector $v$
Perform Gram-Schmidt on $g_k$ and $v$ to obtain a $d \times 2$ orthogonal basis $\tilde{Q}$ and $2 \times 2$
    upper-triangle matrix[a] $R$
$g = \tilde{Q}^T g_k$
Solve the two-dimensional subproblem (9) with $g$ as the gradient, $R$ as the "Hessian", $\varepsilon_l$ as
    the tolerance and obtain solution $v_k$
Output: $s_k = \tilde{Q} v_k$
```

[a] R can be easily converted to a symmetric matrix by averaging the off-diagonal elements, but we don't have to since they yield the same quadratic form.

For the algorithms above we would like to note a few practical considerations that are not covered by the theory. Most notably, in the setting of deep learning we do not consider the convergence criteria as we are training on minibatches and not the whole dataset. Moreover, in a general ARC scheme one would allow for multiple failures ($\rho < \eta_1$) where the parameters were not updated. This means that the adaptive regularization term σ would be increased and the subproblem re-solved. Empirically we note that, for deep learning optimization, this results in diverging ρ values as σ becomes too large due to the fact that we are using minibatches. We can employ a heuristic fix that resets σ to 1 whenever ρ exceeds some unrealistic value. In general more investigation is needed into how minibatches affect adaptive terms such as σ. We do not employ this method here, instead we simply solve the sub-problem once per minibatch whether or not we update the parameters.

2.4 Convergence

Definition 2.1 ($(\varepsilon_g, \varepsilon_H)$-Optimality). *Given $\varepsilon_g, \varepsilon_H \in (0,1)$, $x \in \mathbb{R}^d$ is an $(\varepsilon_g, \varepsilon_H)$-optimal solution to a smooth, possibly non-convex function $f : \mathbb{R}^d \to \mathbb{R}$ if*

$$\|\nabla f(x)\| \leq \varepsilon_g, \; \lambda_{\min}(\nabla^2 f(x)) \geq -\varepsilon_H \tag{11}$$

Remark 2.1. This is a weaker version of the second-derivative test to find local minimum. Note that we can only say that such x is near a local minimum if the Hessian is not degenerate. That is, its eigenvalues are non-zero. Otherwise we can only claim that x is near a second-order stationary point, provided that $\varepsilon_g, \varepsilon_H$ are sufficiently small.

Assumption 1. *(Hessian regularity) Let $f(x)$ be twice differentiable, bounded, and have Lipschitz continuous Hessian on the piece-wise linear path generated by the iterates. That is, for some $0 < K, L < \infty$ and all iterations*

$$\|\nabla^2 f(x) - \nabla^2 f(x_k)\| \leq L\|x - x_k\|, \; \forall \; x \in [x_k, x_k + s_k] \tag{12}$$

$$\|\nabla^2 f(x_k)\| \leq K \tag{13}$$

Remark 2.2. This is also used to prove convergence to second-order critical points by [CGT11] and is a weaker assumption than the global Lipschitz continuous Hessian assumption used in [NP06]. However, it requires further analysis to establish under what circumstances neural networks satisfy this assumption.

Condition 1. *(inexact Hessian regularity) For some $0 < K_H < \infty$, $\varepsilon > 0$, the approximated Hessian B_k satisfies*

$$\|(B_k - \nabla^2 f(x_k))s_k\| \leq \varepsilon\|s_k\| \tag{14}$$

$$\|B_k\| \leq K_H \tag{15}$$

Remark 2.3. In practice we can satisfy this condition using (16), which is an even stronger condition that is easier to implement since it allows us to subsample the Hessian with a fixed, *a priori* error.

$$\|B_k - \nabla^2 f(x_k)\| \leq \varepsilon \tag{16}$$

Condition 2. *(sufficient descent) We require the approximated solution s_k to the subproblem* (7) *satisfy:*

$$-m_k(s_k) \geq \max\{-m_k(s_k^C), -m_k(s_k^E)\}.$$

where s^C is the Cauchy point and s_k^E is along the approximated negative eigenvector direction (if it exists) such that $\langle s_k^E, B_k s_k^E \rangle \leq \nu \lambda_{\min}(B_k)\|s_t^E\|^2 < 0$ for some $\nu \in (0,1]$.

Remark 2.4. The Lanczos method described by [XRM20] starts with the gradient, so the solution immediately satisfies the Cauchy point condition but is not guaranteed to satisfy the eigen point condition. To make sure both are satisfied, we find the leftmost eigenvector explicitly in the ARC-EE algorithm using `scipy.sparse.linalg.eigsh`.

Condition 3. *(sufficient descent for optimal complexity) In addition to Condition 2, the approximated solution s_k to the subproblem* (7) *should also satisfy*

$$\|\nabla m_k(s_k)\| \leq \zeta \max\{\|s_k\|^2, \theta_k \|\nabla f_k\|\}, \;\; \theta_k := \min\{1, \|s_k\|\} \tag{17}$$

where $\zeta \in (0,1)$.

Remark 2.5. This adds an additional hyperparameter ζ and only moderately improves the complexity result. It's more of theoretic interest and thus is omitted in our implementation.

Theorem 2.1. *(Optimal complexity of ARC-IH) Consider any $0 < \varepsilon_g, \varepsilon_H < 1$. Suppose the inexact Hessian, $B(x)$, satisfies Condition 1 with the approximation tolerance $\varepsilon \in \mathcal{O}(\sqrt{\varepsilon_g}, \varepsilon_H)$ (details omitted). Then under Assumption 1, if the approximated solution to the subproblem satisfy Condition 2 and Condition 3, ARC-IH terminates from reaching $(\varepsilon_g, \varepsilon_H + \varepsilon)$-optimality after at most $T \in \mathcal{O}(\max\{\varepsilon_g^{-3/2}, \varepsilon_H^{-3}\})$.*

Remark 2.6. Although here the tolerance in approximating the Hessian appears to be rather strict, in machine learning we typically do not require high accuracy on the solution to prevent overfitting. Therefore, $\varepsilon_g, \varepsilon_H$ do not need to be small, allowing the Hessian approximation to be fairly inexact. This achieves the same optimal complexity as the original ARC paper [CGT11].

2.5 Finite-sum problem

The finite-sum problem is defined as

$$\min_x F(x) := \frac{1}{n} \sum_{i=1}^{n} f_i(x) \tag{18}$$

where $f_i : \mathbb{R}^d \to \mathbb{R}$ are smooth, possibly non-convex functions. This problem often arises in machine learning from the empirical loss objective over finite data points. We will exclusively focus on this type of problem in our experiments.

The Hessian of the objective is just the sum of the Hessians of the $f_i(x)$. To construct B that approximates the Hessian and satisfies (16), we can randomly select according to a uniform distribution a subset of data with indices $\mathcal{J} \subset \{1, \ldots, n\}$:

$$B(x) = \frac{1}{n|\mathcal{J}|} \sum_{j \in \mathcal{J}} \frac{1}{p_i} \nabla^2 f_j(x) \tag{19}$$

For a practical implementation, we instead subsample the gradient and compute the exact Hessian-vector product of the subsampled gradient via automatic differentiation. More sophisticated subsampling schemes are out of the scope of this project and therefore omitted, although they potentially could allow even smaller sample size for the algorithm to converge and thus achieve more speedup.

3 Experiments

We will consider two deep learning model optimization experiments in which we seek to investigate two hypotheses. Namely, that the ARC-IE method cannot achieve its expected performance, and that the ARC-EE variant can satisfy the same claims that [XRM18] states for the Trust Region method they consider. These claims are given below:

C.1: Trust Region performs better in generalizing to test data as compared with SGD.

C.2: Trust region breaks through regions where objective value stagnates (potentially due to saddle points) more easily than SGD.

C.3: The performance of Trust Region is robust to the choice of hyper-parameters, whereas that of SGD is heavily sensitive.

In all of our experiments we take the minibatch size to be 100, and we sub-sample from this for the Hessian approximation with a rate of 0.03 (3 samples). For all ARC variants we use the following hyper-parameters which are similar to those used in [XRM18; Yao+18]: $\eta_1 = 0.1$, $\eta_2 = 0.9$, $\gamma_1 = \gamma_2 = 2$.

We follow [XRM18] in using propagations as a measure of complexity as opposed to an alternative such as epochs, batches, or time. The motivation for doing this comes from the fact that there can be a large amount of variation in performance due to implementation details and hardware specifications. Somewhat informally, this metric gives a better indication of the performance for a given amount of "work". A propagation is defined as a pass through the computational graph of the neural network. For each minibatch of data one forward pass and one backward pass are required to evaluate the objective function and determine the gradients. For a method such as SGD the propagations end there, but in our ARC variants a backward pass is required for each Hessian-vector product and a forward pass for evaluating the proposed parameter update. In addition to this – due to some specific implementation details not discussed here – we need one additional forward and backward pass to get the sub-sampled gradients necessary for the Hessian-vector product. Really, we shouldn't have to do this, and the data being propagated is smaller, but we are limited by software frameworks and we treat the cost as the same. In the subsequent plots below the x-axis is this propagation count.

3.1 Image Classification: MNIST

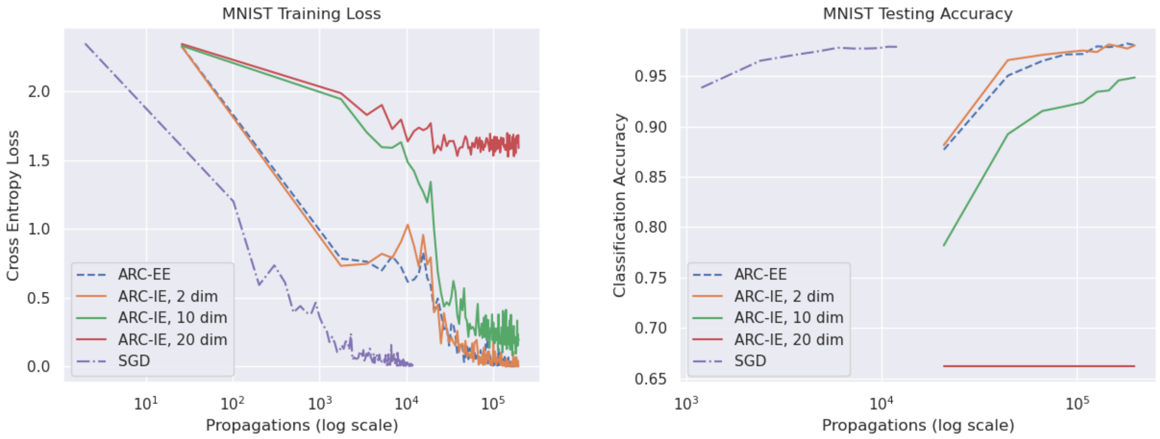
We first consider the MNIST dataset, which is a collection of images containing hand-written digits from 0-9. As such, this presents an image classification task with ten mutually exclusive labels. The dataset contains 70,000 samples with 60,000 for training and the remaining 10,000 for testing. The resolution of all sample images is 28x28 pixels, and they have only a single channel (i.e. grey-scale). This dataset is chosen not for its difficulty, but because it allows for a quick evaluation of the feasibility of optimization methods, of which we consider SGD, ARC-IE, and ARC-EE.

3.1.1 Experimental Setup

We use a simple three-layer fully connected neural network as our image classifier. The images are first flattened into vectors before being passed through three dense layers (affine transformations), with ReLU activation functions being applied to the first two layers. We also employ dropout after the first and second layers. For prediction (i.e. during testing) we use a softmax activation to obtain class probabilities. In total this network comprises 118,282 parameters.

With all methods we train for 10 iterations (epochs) through the whole training set. With SGD we take the learning rate to be 0.01 and the momentum to be 0.9. For ARC-EE we take the termination tolerance ε_l described in Algorithm 2 to be 0.01, and in ARC-IE we take the cubic sub-problem solution tolerance to be 10^-6 and then vary the maximum number of Lanczos iterations. Throughout training we validate the model on the entirety of the testing data at the end of every epoch.

3.1.2 Experimental Results



(a) Objective loss during training.

(b) Testing accuracy evaluated after each epoch.

Figure 1: The loss and accuracy are only computed after the first minibatch and epoch respectively. Thus the number of total propagations at this point can vary among the methods which results in the different starting points. As well, given we train for a fixed number of epochs (and not propagations) the ending points also differ.

In Figure 1a and Figure 1b above we vary the maximum number of iterations in the Lanczos method used by ARC-IE to solve the sub-problem. This effectively translates to increasing the dimension of the Krylov subspace which, in theory, should make it more likely to contain the eigen point. In turn, this should make for a better solution to the sub-problem, but we observe the opposite behaviour. Specifically, we note that as we increase the maximum dimension from two to ten to twenty, the accuracy on the testing data decreases. Although it does have reasonable performance with two maximum dimensions, it is unreasonable to expect that the leftmost eigenvector can be included in the subspace after so few iterations, and the success seems to mostly come from satisfying the Cauchy point condition. Essentially, this makes 2-dim ARC-IE a more expensive variant of SGD without any performance benefits, missing the point of second-order methods.

In fact, [XRM18] also reported poor performance of ARC-IE on similar deep-learning problems using a maximum dimension of 250. Subsequently, they left out these results and did not expand on exactly the issues encountered or any potential reasoning as to why, but it appears that we have replicated their findings. Why did the result diverge from theoretical expectation? An simple explanation is that Lanczos method is highly prone to numerical instability in the orthogonalization process, and a naive hand-coded implementation is likely to experience problems. However, [XRM18] reported that they also tried standard library implementations that still maintained similar poor performance. Another possible explanation is that when we search for the minimizer to the non-convex sub-problem in the higher-dimensional subspace instead of a two-dimensional subspace, the solution becomes harder to find; a solver like L-BFGS might be more likely to return a bad solution.

It is not quite within the scope of this work to investigate exactly why ARC-IE performs as such, but it is clear that it does not behave as expected. Therefore, we exclude ARC-IE from further experiments and focus instead on ARC-EE which we expect to perform better. This is validated in Figure 1a and Figure 1b as well, and it is part of why we include these methods in this experiment. Specifically, we can see that ARC-EE performs on par with SGD in terms of performance, albeit at a higher cost of propagations.

3.2 Image Classification: CIFAR-10

We now move on to the much harder image classification task using the CIFAR-10 dataset, which is in many ways similar to MNIST but with an increased difficulty coming from the structure of the images. In this case there are 50,000 training samples and 10,000 testing samples with 10 mutually exclusive classes distributed evenly across the dataset (6000 images per class). The images in this case have resolution 32x32 pixels with three channels, so they are larger and full colour. This is part of what makes the classification task here harder, with the other part coming from the variety of image subjects which include airplane, cat, frog, and ship. As discussed earlier, we will consider only SGD and ARC-EE as optimization methods.

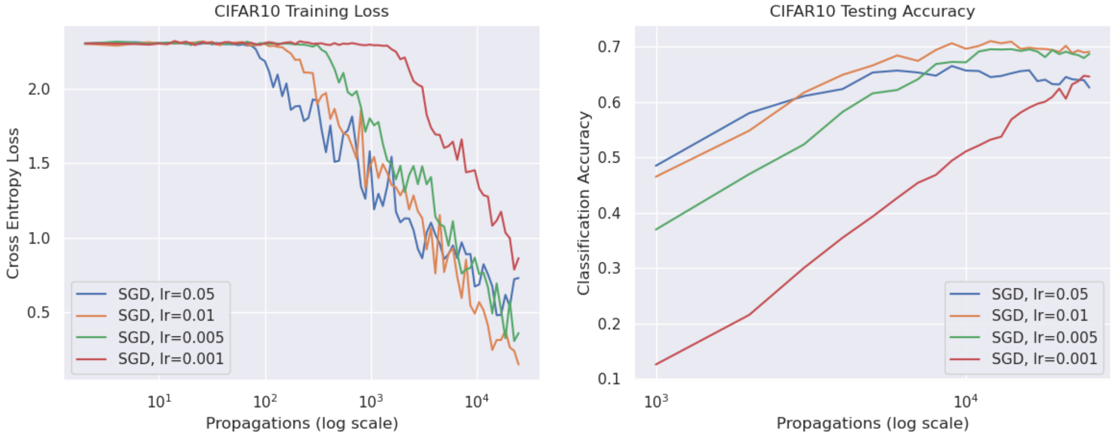
3.2.1 Experimental Setup

For this experiment we employ a Convolutional Neural Network (CNN) which aides in performance on the harder task, and presents a more complicated optimization problem. Specifically, we use two convolutional layers with max-pooling followed by three dense layers. ReLU is used as an activation function, and for prediction we again use softmax to obtain class probabilities. The total number of parameters in this network comes out to 121,182 which we note is similar to the network used in the MNIST task. In this case, however, the architecture is more specialized for image data, so the parameters are being used in a more informed way.

Before training we normalize the images by subtracting the mean and dividing by the standard deviation. This is common practice in deep learning, and often necessary for a model to perform well. We conduct training for 25 epochs and we validate on the entire test set to get a thorough picture of the performance of the network as it is optimized. In the experiments shown below we vary the learning rate and the momentum hyper-parameters for SGD, and the termination tolerance ε_l in Algorithm 2 for ARC-EE.

3.2.2 Experimental Results

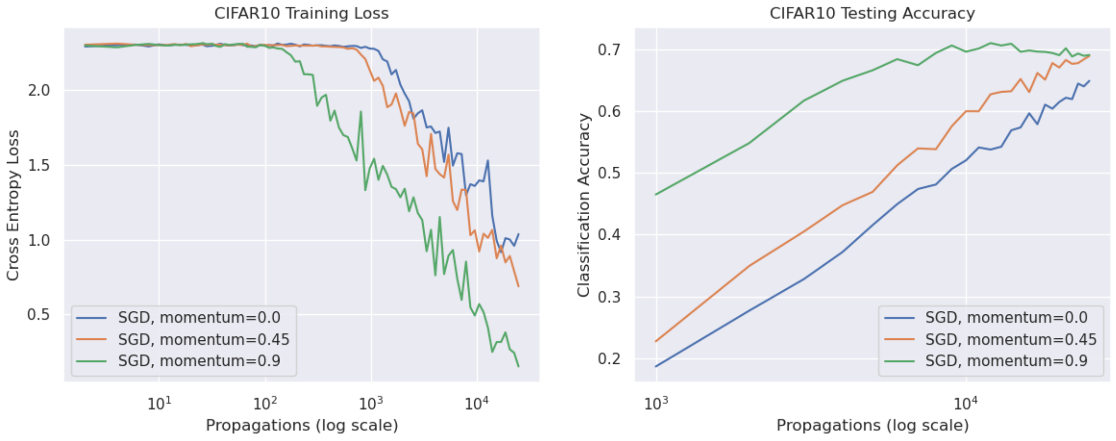
As noted earlier in the MNIST experiment, the loss and the test accuracy are computed after the first minibatch and epoch respectively. Therefore there is a non-zero number of propagations required before we have these data, resulting in the x-axis starting at some non-zero number. In the case of ARC-EE, the number of propagations per update step is variable, resulting in differing start/end points.



(a) (b)

Figure 2: Training loss and testing accuracy for SGD as we vary the learning rate parameter.

In Figure 2a and Figure 2b we see that a larger learning rate leads to faster training and vice versa. However, if the learning rate is too small (0.001) or too big (0.05) then we see a noticeable drop in testing accuracy. We can also see that an ideal choice of learning rate (in this case 0.01) rewards one with quick training and good generalization. Going beyond the range of learning rates that we consider would likely result in severe drops in performance or complete divergence. This result also casts doubt on part of **C.1** where [XRM18] observed that SGD trials most successful at reducing training loss has worse testing accuracy than less successful trials.

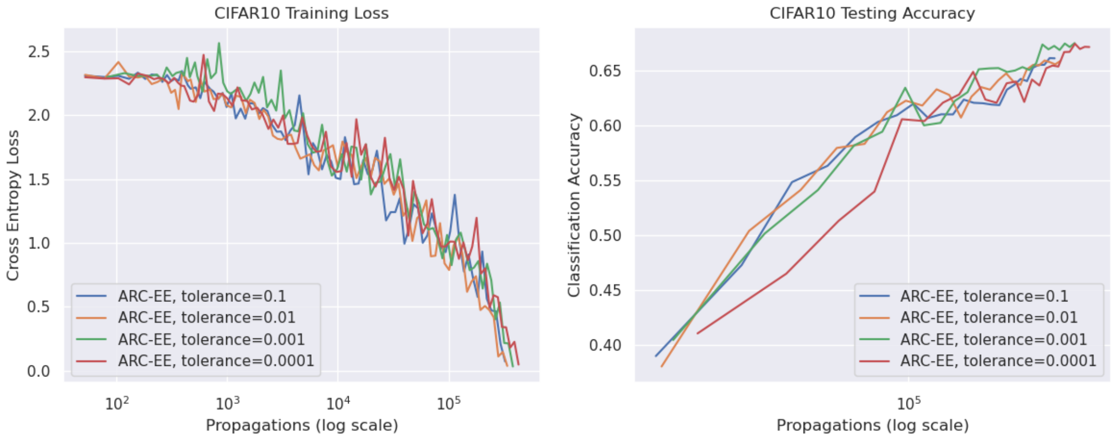


(a)

(b)

Figure 3: Training loss and testing accuracy for SGD as we vary the momentum parameter.

From Figure 3a and Figure 3b we see that a high momentum factor can significantly speed up the training process. The purpose of momentum is to break through regions in the loss landscape that have a small gradient but are not minima. From this we can hypothesize that without momentum certain levels of test accuracy would not necessarily be achievable. Our results do not directly show this to be true, but we can conclude from the preceding two figures that SGD is somewhat sensitive to the choice of hyperparameters. In [XRM20] they claim a to show a more intense sensitivity, but much of the evidence for this comes down to them choosing unrealistic values such as using a learning rate of 5. It is also import to note that there are many other variants of SGD used in practice which can have many more hyperparameters potentially resulting in compounding variation.



(a)

(b)

Figure 4: Training loss and testing accuracy for ARC-EE as we vary the required tolerance in finding the approximated leftmost eigenvector.

Looking at Figure 4a and Figure 4b, we see that the training loss of ARC-EE is almost zero yet the testing acurracy is only at around 67%. This implies a large degree of overfitting especially compared to SGD. While it is impressive that ARC-EE can consistently make progress on reducing loss, this typically desirable characteristic is counterproductive in deep learning problems. Clearly, just using minibatches and multiple epochs are not sufficient to prevent ARC-EE from overfitting, and we need additional measures to keep it in check.

In addition, we can observe that ARC-EE exhibits robustness to varying the termination tolerance ε_l, with all four tolerances converging to a similar accuracy in the end. However, if we take a more nuanced look, we see that a low tolerance resulted in a slow accuracy increase per propagation at first, but eventually caught up. There are two potential explanations to this. First, it is possible that our ARC-EE implementation cannot progress much further than about 67%, so trials with various tolerances inevitably converge in the end regardless of their efficiency in the beginning. Another explanation might be that as training evolves, we require higher accuracy in the Lanczos process to make decent progress. Although we didn't have the time to run the training longer to test our hypotheses, if the latter explanation is true, it could justify using an adaptive tolerance where we request an increasingly more accurate leftmost eigenvector as we train.

One may ask why we chose to vary the termination tolerance in investigating sensitivity to hyperparameters for this method. It is certainly true that our implementation of ARC-EE for the deep learning problems has other hyperparameters involved, including subsampling rate and batch size, but many of these are less immediately important to the generic algorithm described in Algorithms 1 and 2. In [XRM20] they choose to vary the initial trust region radius, which would be analogous to our initial regularization parameter σ_0. However, because σ is adaptive, we naturally expect the algorithm to be robust with varying σ_0. Thus, we do not believe it is informative to test sensitivity to varying σ_0 in our experiments.

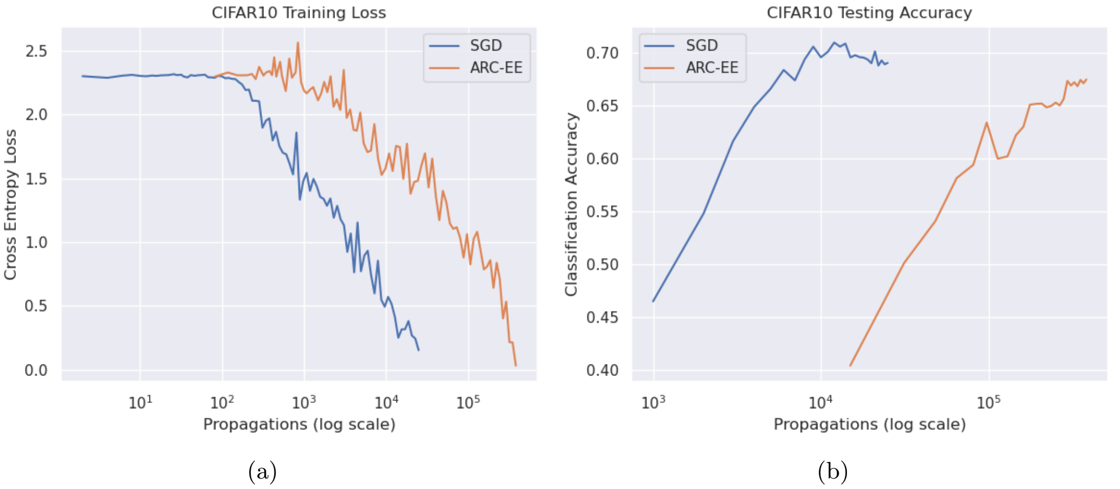


Figure 5: Training loss and testing accuracy for SGD and ARC-EE using the hyperparameters that result in the best performance.

In subsection 3.2.2 we see the results from SGD using a learning rate of 0.001 and momentum 0.9 alongside the results of ARC-EE using a tolerance of $\varepsilon_l = 0.001$. Clearly SGD achieves greater testing accuracy in much fewer propagations than ARC-EE. It is interesting, however, to observe that SGD seems to plateau at a testing accuracy of around 0.7, and it is unclear whether it can break through with more training. It is possible that ARC-EE might break through where SGD could not, and if that is the case, we might consider a hybrid algorithm where we predominantly use SGD but switch to ARC-EE when SGD fails to make progress.

Having seen all of our experimental results we are now ready to consider the claims stated at the beginning of this section. **C.1** would imply that even though SGD and ARC-EE might achieve similar training loss, ARC-EE would be able to achieve better testing accuracy. This is clearly not supported by our experiments (see subsection 3.2.2). In fact, the opposite appears to be true. A few results for a single dataset are hardly conclusive, but at the moment this claim does not hold. **C.2** would imply SGD could stagnate at certain levels of performance where ARC-EE would continue to gain ground. Again we do not see this supported in the data. It is quite possible that the network we consider does not have enough stagnation areas in its loss landscape for us to observe this phenomenon. For example, [XRM18] showed this occurring for a deep autoencoder containing many more parameters. Thus the results here are inconclusive and a more in-depth study with a variety of models is required to answer this question fully. Lastly, we can say that **C.3** partially holds. In particular, the impact of varying the learning rate on the performance of SGD seems to be greater than that of varying tolerance on the performance of ARC-EE in our experiment. However, since we didn't use a learning rate that is usually considered too large (such as 0.5 or 5), we failed to replicate the dramatic failure of SGD reported in [XRM18]. Therefore, we would only characterize SGD as moderately sensitive to the learning rate parameter for the CIFAR-10 dataset. SGD seems to have more sensitivity to the momentum parameter, but 0.9 appears to be a common default value and indeed provides better results. Hence, it wouldn't be very fair to count this against SGD when the parameter is rarely tuned. On the other hand, although Algorithm 1 has five hyperparameters, we can somewhat ignore $\varepsilon_g, \varepsilon_H, \varepsilon$ for deep learning problems and set η_1, η_2, γ to values commonly used in the literature as they work well across problems. The only hyperparameter left is ε_l in Algorithm 2 and our experiment indeed showed that ARC-EE is quite robust to variations in ε_l within a reasonable range.

4 Conclusion

We have presented an investigation and discussion of ARC – a second-order method for non-convex optimization. Specifically, we have considered an approximate variant which uses sub-sample Hessian information. Second-order methods have many benefits given they account for curvature information, and sub-sampling seeks to make the cost reasonable enough for use in large-scale problems. Furthermore, we implemented a new variant to ARC (which we call ARC-EE) that solves the associated sub-problem by using explicit leftmost eigenvector information. This fixes the flawed implementation from previous work which attempts to capture the information implicitly. Next, we proceeded to investigate these methods through the lens of neural network model optimization. We find that the previous implicit method for solving the ARC subproblem does not perform as expected, but that our new variant succeeds. Unfortunately, we did not find that ARC-EE was better in generalizing to test data as compared with the standard first-order method SGD. Given the experiments we considered it is inconclusive whether ARC-EE is able to break through regions of stagnation where SGD is not. Finally, we did find that ARC-EE was more robust to hyperparameter tuning than SGD, although some of this comes down to the fact that ARC doesn't have many hyperparameters that require tuning.

We have identified many interesting future directions for the work presented here. First, we see that ARC-EE might be more prone to overfitting than SGD and require additional safeguards to generalize its impressive training accuracy. It would be interesting to investigate to what extent the common techniques for overfitting prevention used on SGD can help ARC-EE. Second, due to the limitations of the PyTorch framework, our current subsampling scheme requires two more propagations per minibatch than should be necessary. In addition to fixing this, there are many code based improvements that would allow ARC-EE to be more competitive with SGD. Third, our results from Figure 4b hint at the potential benefits of using an adaptive tolerance, which requires additional experiments to confirm. In particular, as we make further progress on training, we might require an increasingly accurate eigenvector to improve testing accuracy. This might especially be true if the gradient is small, so we have to rely on the eigen point to achieve sufficient descent. Therefore, a reasonable adaptiveness of the tolerance can come from the norm of the gradient. Fourth, assuming we can successfully mitigate ARC-EE's overfitting problem, then to investigate the benefits of a second-order method we should compare SGD and ARC-EE on a much harder problem where even SGD with momentum struggles. The autoencoder problem used in [XRM18] might be a good start, but there are a plethora of such tasks. Fifth, as we discussed earlier, it is unclear whether it is theoretically sound to update σ across minibatches as if we are training on the same batch. That is, theoretical results showing that ARC-EE works on an optimization problem does not necessarily generalize to deep learning problems with stochastic training. Finally, even if it turns out that ARC-EE is not suitable as a primary optimization algorithm for training neural networks, we could perhaps design a hybrid algorithm of SGD and ARC-EE so that we can take advantage of the second-order magic of ARC-EE when SGD struggles to make progress.

Altogether the road ahead is promising, but much more work, experimentation, and theory development are needed.

5 References

[LSY98] Richard B Lehoucq, Danny C Sorensen, and Chao Yang. *ARPACK users' guide: solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods*. SIAM, 1998.

[NP06] Yurii Nesterov and Boris T Polyak. "Cubic regularization of Newton method and its global performance". In: *Mathematical Programming* 108.1 (2006), pp. 177–205.

[CGT11] Coralia Cartis, Nicholas IM Gould, and Philippe L Toint. "Adaptive cubic regularisation methods for unconstrained optimization. Part I: motivation, convergence and numerical results". In: *Mathematical Programming* 127.2 (2011), pp. 245–295.

[KL17] Jonas Moritz Kohler and Aurélien Lucchi. "Sub-sampled Cubic Regularization for Non-convex Optimization". In: *CoRR* abs/1705.05933 (2017). arXiv: `1705.05933`. URL: `http://arxiv.org/abs/1705.05933`.

[XRM18] Peng Xu, Farbod Roosta-Khorasani, and Michael W. Mahoney. *Second-Order Optimization for Non-Convex Machine Learning: An Empirical Study*. 2018. arXiv: `1708.07827 [math.OC]`.

[Yao+18] Zhewei Yao et al. *Inexact Non-Convex Newton-Type Methods*. 2018. arXiv: `1802.06925 [math.OC]`.

[XRM20] Peng Xu, Fred Roosta, and Michael W Mahoney. "Newton-type methods for non-convex optimization under inexact hessian information". In: *Mathematical Programming* 184.1 (2020), pp. 35–70.

6 Appendix

6.1 Connections to Class

Here we link parts of the paper that made connections to what we learned in class:

1. How Trust Region can be intuitively understood as the dual to line search which we used in homework: subsection 2.1.
2. Derivation of Newton's method and its drawbacks: (1).
3. How the derivation of Cubic Regularization is analogous to that of Gradient Descent: (4).
4. How Lanczos method is different from Conjugate Gradient which is another Krylov subspace method: subsection 2.3.

6.2 Code Repository

GitHub Repository: https://github.com/RS-Coop/convex-optimization-project