ECE 273 SP22 Project: Convexifying Neural Network

Saqib Azim (A59010162) sazim@ucsd.edu
Mehmet Hucumenoglu (TA)
Prof. Piya Pal (Instructor) pipal@eng.ucsd.edu

1.1 Describe the problem setting and the objective

- This paper introduces a finite dimensional convex program that provides global optimal solution for the problem of training two-layer neural networks with ReLU activation function.
- The number of variables in the convex program can be represented as a polynomial function of the number of training samples n and number of hidden neurons m.
- This work shows that ReLU networks trained with standard weight decay are equivalent to block l_1 penalized convex models.
- In this problem, we have a two-layer neural network $f : \mathbb{R}^d \to \mathbb{R}$ with input size d, m neurons in hidden layer, single-valued output. The input layer neurons depends on input feature dimension d. The hidden layer consists of m neurons each of which outputs $x^T u_j, \quad j = 1, 2, ..., m$. The output of each hidden neuron j is passed to ReLU activation function resulting in $\phi(x^T u_j)$. Finally, the output layer contains a single neuron outputting -

$$f(x) = \phi(x^T u_1)\alpha_1 + \phi(x^T u_2)\alpha_2 + ... + \phi(x^T u_m)\alpha_m = \sum_{j=1}^{m} \phi(x^T u_j)\alpha_j \tag{1}$$

$x \in \mathbb{R}^d$ - input to the neural network
$m \in \mathbb{N}$ - number of neurons in hidden layer
$\phi(x)$ = ReLU activation function = $\max(0, x)$
$u_j \in \mathbb{R}^d$ - weights corresponding to jth hidden neuron in layer-1
$\alpha_j \in \mathbb{R}$ - weights corresponding to the output neuron in layer-2
$f(x)$ - output of the neural network

1.2 Specify the problem parameters and the neural network architecture used

- The neural network architecture consists of fully connected 2-layers with the input layer size (d) depending on the size of input $x \in \mathbb{R}^d$. The hidden layer consists of m neurons followed by a ReLU activation function. The output layer contains a single neuron.

1.3 What is the non-convex optimization problem used to train the network?

$$p_1^* = \min_{\{\alpha_j, u_j\}_{j=1}^m} \frac{1}{2} \left\| \sum_{j=1}^{m} \phi(Xu_j)\alpha_j - y \right\|_2^2 + \frac{\beta}{2} \sum_{j=1}^{m} (\|u_j\|_2^2 + \alpha_j^2) \tag{2}$$

$X \in \mathbb{R}^{n \times d}$ - Data matrix (n data samples)
$y \in R^n$ - ground-truth labels corresponding to n data samples
$\beta \in \mathbb{R}^+$ denotes the regularization parameter

Preprint. Under review.

p_1^* represents the optimal value of this non-convex optimization problem.

1.4 Why is this problem non-convex?

- The objective function in Eq. 2 is highly non-convex wrt parameters u_j and α_j due to the non-linear ReLU activation function and the product between hidden layer weights u_j and outer layer weights α_j.
- ReLU is a convex function (easy to see from its plot) but the output of the neural network $f(x) = \sum_{j=1}^{m} \phi(Xu_j)\alpha_j$ can be seen as a weighted sum of ReLU convex functions with α_j as the weights. These weights can be negative and therefore we cannot guarantee that the overall sum would also be convex (in general). Any norm is a convex function and hence the L_2-norm of the weight parameters in regularization term is convex and their sum is also convex. But due to the L_2-norm of the error between $f(x)$ and y, where $f(x)$ may not be always convex, the overall objective function is non-convex.

1.5 What kind of loss is used?

- The authors have used a squared L_2-norm error between the output of neural network $f(x) \in \mathbb{R}^n$ and the ground-truth label $y \in R^n$ in addition to the squared L_2-norm of all the network weight parameters weighted by the regularization parameter $\beta > 0$.

1.6 What kind of regularization is used and what is the rationale for this choice?

- The non-convex objective function in Eq. 2 uses squared L_2-norm regularization of all the network weight parameters weighted by the regularization parameter $\beta > 0$.
- The reason for using L2-norm weight regularization is to prevent weight magnitude from taking very large values. If we take the gradient of the regularized objective function $\nabla_u L_R(u)$, it is equal to the sum of the gradient of non-regularized loss $\nabla_u L_{NR}(u)$ + $2\beta u$. When updating weights in SGD (Eq. 4), the weight u is subtracted by its correction term $2\epsilon\beta u$. Hence, the larger the weight parameter u, larger would be the correction term, therefore keeping the weight magnitude from exploding.

$$\nabla_u L_R(u) = \nabla_u L_{NR}(u) + 2\beta u \tag{3}$$

$$u \leftarrow u - \epsilon\nabla_u L_R(u) = u - \epsilon\nabla_u L_{NR}(u) - 2\epsilon\beta u \tag{4}$$

- The authors have not used L1 norm because the gradient of the L1-regularized loss subtracts a constant $2\epsilon\beta$ during the update step and this leads to sparsity in the optimal solution.

$$\nabla_u L_R(u) = \nabla_u L_{NR}(u) + \beta \tag{5}$$

$$u \leftarrow u - \epsilon\nabla_u L_R(u) = u - \epsilon\nabla_u L_{NR}(u) - 2\epsilon\beta \tag{6}$$

- In neural networks, the weight magnitudes are kept smaller, so that no individual weight parameter dominates the output which helps prevent overfitting.

2.1 What is the convex problem authors propose to solve in [1]?

The authors propose to solve the below finite-dimensional constrained convex problem Eq. 7 as a replacement of non-convex problem in Eq. 2. This is because, under suitable conditions, the paper shows that the optimal values of both problems are identical and the optimal solutions of non-convex problem can be derived from optimal solutions of convex problem.

$$p_0^* = \min_{\{v_i, w_i\}_{i=1}^P} \quad \frac{1}{2}\Big\| \sum_{i=1}^{P} D_i X(v_i - w_i) - y \Big\|_2^2 + \beta \sum_{i=1}^{P} (\|v_i\|_2 + \|w_i\|_2) \tag{7}$$

subject to following constraints -

$$(2D_i - I_n)Xv_i \geq 0, \quad (2D_i - I_n)Xw_i \geq 0 \quad \forall i = 1, 2, ..., P \tag{8}$$

2.2 How was the non-convex problem converted into a convex problem?

Here we show the derivation for obtaining convex representations of the non-convex objective function in Eq. 2.

Firstly, we write the L_1-penalized representation of the optimization problem in Eq. 2. That is $p_1^* = p_2^*$. This equivalence can be shown by appropriately scaling the weight parameters u_j, α_j and noting that the NN output remains unchanged by this scaling.

$$p_1^* = \min_{\|u_j\|_2 \leq 1 \forall j} \min_{\{\alpha_j\}_{j=1}^m} \frac{1}{2} \left\| \sum_{j=1}^m \phi(Xu_j)\alpha_j - y \right\|_2^2 + \beta \sum_{j=1}^m |\alpha_j| \tag{9}$$

Now, we replace the inner minimization with its convex dual problem given by -

$$p_1^* = \min_{\|u_j\|_2 \leq 1 \forall j} \max_{v \in R^n \, st \, |v^T \phi(Xu_j)| \leq \beta \forall j} -\frac{1}{2} \left\| y - v \right\|_2^2 + \frac{1}{2} \|y\|_2^2 \tag{10}$$

Interchanging the order of min and max, we obtain the lower bound d^* (Eq. 11) via weak duality which is a semi-infinite convex problem

$$p_1^* \geq d^* = \max_{v \in R^n \, st \, |v^T \phi(Xu)| \leq \beta \forall u \in B_2} -\frac{1}{2} \left\| y - v \right\|_2^2 + \frac{1}{2} \|y\|_2^2 \tag{11}$$

The main idea behind converting the non-convex objective function to a constrained convex objective is to represent the ReLU function into linearized matrix multiplications. The paper breaks down the R^d - parameter search space into certain number of partitions P using hyperplanes. They convert the dual constraint in Eq. 11 over all possible hyperplane arrangement patterns given below -

$$\max_{\|u\|_2 \leq 1} v^T \phi(Xu) = \max_{\|u\|_2 \leq 1} v^T \text{diag}(Xu \geq 0) Xu \tag{12}$$

The optimal values of the convex and non-convex problem are identical only when the 2-layer ReLU neural network has sufficient number of hidden neurons m so that the NN has enough learning capacity to be able to learn over the space of $u \in R^d$. Essentially, strong duality holds only when $m \geq m^*$ = number of non-zero optimal solutions of the convex problem.

2.3 What kind of convex optimization problem is it?

- The convex optimization problem in Eq. 7 represents a constrained finite dimensional problem (in space $\mathbb{R}^{2dP}$) with $2d \times P$ variables and $2n \times P$ linear inequality constraints.
- The dual of the non-convex problem (Eq. 2), given by Eq. 11, is a convex semi-infinite optimization problem with n variables and infinitely many constraints.

3.1 How do the authors propose to train the neural network?

- The paper uses a 1-dim randomly generated dataset with small number of samples. These data points are fit using a 2-layer ReLU network trained with Stochastic gradient descent.
- The authors propose to first convert the non-convex problem in Eq. 2 to a constrained convex problem in Eq. 7 and then solve the convex problem using standard optimization libraries (such as CVXPY).

- Using the obtained optimal solutions of convex problem (v^*, w^*), the paper uses theorem 1 to estimate the optimal solutions to the original non-convex problem (u^*, α^*).

3.2 What are the theoretical guarantees given in [1]? Under what conditions do these guarantees hold?

- The convex problem defined in Eq. 7 and the non-convex problem defined in Eq. 2 where $m \geq m^*$ have the same optimal values $\implies p_0^* = p_1^*$. This guarantees that the 2-layer ReLU network with m hidden neurons can be globally optimized using the constrained convex problem in Eq. 7 with $2d \times P$ variables and $2n \times P$ linear inequality constraints.
- Let the optimal solutions to the convex problem in Eq. 7 be denoted by these P sets of values $\{v_i^*, w_i^*\}_{i=1}^P$. Then the optimal solution of non-convex problem in Eq. 2 denoted by $\{u_i^*, \alpha_i^*\}_{i=1}^m$ can be written in terms of optimal solution of the convex problem in Eq. 7.

$$(u_{j_{1i}}^*, \alpha_{j_{1i}}^*) = \left(\frac{v_i^*}{\sqrt{\|v_i^*\|_2}}, \sqrt{\|v_i^*\|_2} \right) \quad \text{if} \quad v_i^* \neq 0$$
$$(u_{j_{2i}}^*, \alpha_{j_{2i}}^*) = \left(\frac{w_i^*}{\sqrt{\|w_i^*\|_2}}, -\sqrt{\|w_i^*\|_2} \right) \quad \text{if} \quad w_i^* \neq 0\,,$$

- The paper proves theorem 1 results (above) given that the number of hidden neurons $m \geq m^* = \sum_{i:v_i^* \neq 0}^P 1 + \sum_{i:w_i^* \neq 0}^P 1$ = number of non-zero optimal solutions of Eq. 7. In order to solve the convex problem, we need the number of partitions P and therefore, m^* can be used to select partitions.
- Thus, the convex problem solution is unlike local search heuristics such as backpropagation which may end up converging to sub-optimal local solutions.
- The above equations implies that the ReLU network are equivalent to sparse mixtures of linear models. The non-convex neural network approach projects the data matrix X to higher dimensional feature matrix $[D_1 X, ..., D_P X]$ and then tries to estimate a group sparse model.

3.3 What is the computational complexity? Is the complexity polynomial or exponential in problem size?

- The computational complexity for solving the constrained convex problem in Eq. 7 is at most $O(d^3 r^3 (\frac{n}{r})^{3r})$. Here $r = \text{rank}(X) \leq \min(n, d)$ represents the rank of the data matrix. For fixed rank r (or fixed input dimension d), the complexity is polynomial in n and m (network size). For fixed n and $r = rank(X) = d$ (full rank), the complexity becomes $O(d^6 (\frac{n}{d})^{3d})$ which is exponential in problem size d.
- The paper claims to be the first polynomial time algorithm to train non-trivial neural networks with global optimality guarantees.

4.1 Note that in the convex formulation in Eq. 7, the inner sum consists of P terms (instead of m terms as in the non-convex problem in Eq. 2). What does P denote? What does each matrix $D_i, i = 1, ..., P$ correspond to in this setting? Why do you think such matrices D_i show up in the convex problem? Explain.

- In Eq. 7, the P denotes the number of regions in a partition of $\mathbb{R}^d$ space, created by hyperplanes passing through the origin. Consider a random vector $u \in \mathbb{R}^d$. Consider many hyperplanes each

passing through the origin with equations $u^T x = 0, \quad u \in R^d$ represents vector orthogonal to the hyperplane. For a fixed u, each data sample $x_i \in R^d$ from the data matrix X lies either on the positive halfspace or negative halfspace of the hyperplane. Thus, each hyperplane divides the n data samples in R^d into two partitions (positive and negative sides). For each random vector u, we have a unique hyperplane through the origin but the partition of data samples won't be always unique. This is explained very elaborately in [3]. If we create a diagonal matrix with the diagonal entries $[1(x_1^T u \geq 0), ..., 1(x_n^T u \geq 0)]$ we will see that there can only be at most 2^n such distinct arrangements.

- For all possible $u \in R^d$, we find such $n \times n$ diagonal matrices and denote them as $D_1, ..., D_P$ where $P \leq 2^n$. If $r = \text{rank}(X) \leq \min(n, d)$, then $P \leq 2\sum_{k=0}^{r-1} \binom{n-1}{k} \leq 2r\left(\frac{e(n-1)}{r}\right)^r$. These diagonal matrices D_i reflect all possible values of $\text{sign}(Xu) \quad \forall u \in R^d$. This helps to transform the non-linear and non-convex weighted ReLU functions to linear convex operations.

5.1 How does the optimization problem change if the underlying network architecture is a Convolutional Neural Network (CNN)?

- For a 2-layer CNN, consider input to be an image (or tensors). Each image (or tensor) can be broken down into K patch matrices each of dimension d. Thus, for n data samples (or n images), with each data sample consisting of K matrix patches, the data can be represented using K matrices $X_1, X_2, ..., X_K$ where each $X_i \in \mathbb{R}^{n \times d}$.
- In the 2-layer CNN, the hidden layer contains m hidden neurons (each neuron represents a CNN filter). $u_j \in \mathbb{R}^d \quad j = 1, ..., m$ represents weights corresponding to jth hidden neuron (or filter). For each filter, the hidden layer outputs K vectors each of length n given by $X_k u_j \quad k = 1, ..., K$.
- The hidden layer output, for each filter index $j = 1, ..., m$, passes through a ReLU activation layer resulting in $\phi(X_k u_j) \in \mathbb{R}^n \quad k = 1, ..., K$. The final output layer is a fully connected layer with weights represented by $\{\alpha_{jk} \in \mathbb{R} \quad j = 1, .., m \quad k = 1, .., K\}$. Following is the network output -

$$f(X_1, ..., X_K) = \sum_{j=1}^{m} \sum_{k=1}^{K} \phi(X_k u_j) \alpha_{jk} \tag{13}$$

- For CNNs, each patch $k = 1, ..., K$ has its own label vector $y_k \in \mathbb{R}^n$. The optimization problem in Eq. 2 modifies to the following problem for CNNs which is separable over the patch indices k -

$$p_3^* = \min_{\{\alpha_j, u_j\}_{j=1}^m} \frac{1}{2} \sum_{k=1}^{K} \left\| \sum_{j=1}^{m} \phi(X_k u_j) \alpha_{jk} - y_k \right\|_2^2 + \frac{\beta}{2} \sum_{j=1}^{m} (\|u_j\|_2^2 + \|\alpha_j\|_2^2) \tag{14}$$

- The above problem can be converted into fully connected non-convex problem (Eq. 2) by suitably transforming the data matrix X_k and labels y_k.
- *Linear CNN trained as semi-definite program (SDP)*: In this case, the authors have considered linear activation $\phi(x) = x$ in above equation 14. Strong duality holds based on arguments similar to Theorem 1. The dual of above problem Eq. 14 can be transformed into an SDP and the dual of this SDP is a convex optimization problem with nuclear-norm penalty.
- *Linear circular CNNs*: In this case, the authors assume that if the patch matrices are sufficiently zero-padded and extracted with stride=1, then the linear circular version of Eq. 14 can be transformed into an SDP whose dual is a convex problem in the complex field.

6.1 Study the paper [2] and the theoretical guarantees concerning the convex problem in Theorem 1. What does it say? How are those guarantees different from the ones given in [1] ?

- In paper [1], the authors have proved a theorem stating that under the condition that the number of hidden neurons $m \geq m^*$, the optimal values of convex problem 7 and non-convex problem 2 are

identical. In addition, the paper guarantees that all optimal solutions of the non-convex loss can be found via the optimal solutions of the convex program upto permutation and splitting/merging of the neurons.

- In paper [2], there is an extension that can be used to estimate all possible optimal solutions of the non-convex problem using the global optimal solution of the convex problem. That is, the authors prove that that it is possible to find all globally optimal 2-layer ReLU NNs by solving a convex problem with cone constraints.
- In paper [1], theorem 1 only guarantees that given strong duality holds, the optimal solution of convex program can provide one particular set of optimal weight parameters of 2-layer ReLU NN. Whereas this paper guarantees all possible globally optimal solutions can be estimated from the convex optimal solution. That is the difference.

7. Implement the first experiment from Fig 2 in [1]. Generate the data randomly. In particular for a small n (the sample size) and m = 5 (small number of hidden neurons), use the Stochastic Gradient Descent method to solve the non-convex problem, with different initializations. Plot the training objective value vs the number of iterations. Next, solve the convex program with CVX/CVXPY and plot its optimal value. What do you observe? Does the SGD always converge to the optimal value of the convex program? Perform the same experiment with larger values of m. Do your observations agree with the theoretical guarantees?

Response in the code section on Page 16

8. Replicate this experiment for a larger value of n and an appropriate m. Plot the objective value against the number of iterations. How does this experiment differ from the case for small n? In addition to the original convex program, solve the approximate convex program described in Remark 3.3. Compare the objective value of the approximate version with both SGD and convex program. What is the advantage of solving the approximate convex program?

Response in the code section on Page 20

9. Perform an experiment for the binary classification task on CIFAR-10 dataset. Choose any two classes you prefer. Choose the value of m appropriately. Train the network using SGD on the non-convex problem for different initializations. Solve the approximate convex program with CVX/CVPY. Plot objective value vs iterations. Evaluate the performance of trained networks on the test set, and plot test accuracy vs iterations. Comment on your observations.

Response in the code section on Page 26

10.1 Summarize your overall conclusions about what you learned from this project.

- Firstly, This paper shows the importance of convex optimization in globally solving a non-convex neural network objective function. Not only that, they also proved that if we know a global solution of convex problem, we can estimate a global solution of neural network weights as well.
- In order to prepare for this project, I read and learnt a lot about duality and KKT conditions, linear and quadratic programming by myself in order to understand many of the duality conversion and theorem 1 in paper 1.
- The paper gives an elegant theoretical guarantee about the relation between convex and non-convex objective functions. In the future, it will be interesting to see whether such guarantees and conversion can be achieved for larger NNs with more complicated non-convex loss and not just a toy model.
- To be honest, I was not able to understand many of these nuanced mathematical steps, especially in the paper [2]. Everytime I tried to google and learn about something, it made me realize the vast field related to optimization, various types of programming (linear, quadratic, semi-definite, semi-infinite, etc.) and most importantly the duality theory. I spent too much time learning about duality and KKT conditions (thinking that it might help in understanding these papers) but still

found it difficult to follow along the main proof in theorem 1 with such big equations. Made me wonder and appreciate the hard work and efforrts put in by the authors. This project was helpful in understanding some deeper nuances of convex and non-convex optimization and also practically seeing NN training from much deeper mathematical view (which most of us are not used to).

References

[1] Pilanci, Mert, and Tolga Ergen. "Neural networks are convex regularizers: Exact polynomial-time convex optimization formulations for two-layer networks." International Conference on Machine Learning. PMLR, 2020.

[2] Wang, Yifei, Jonathan Lacotte, and Mert Pilanci. "The Hidden Convex Optimization Landscape of Regularized Two-Layer ReLU Networks: an Exact Characterization of Optimal Solutions." International Conference on Learning Representations. 2021.

[3] Pilanci, Mert "Presentation slides - https://stanford.edu/ pilanci/papers/TALK_Convex_NN.pdf".

ECE273_project_code

June 10, 2022

[22]:
```
import os
import numpy as np
import matplotlib.pyplot as plt

import cvxpy as cp

import torch
import torch.nn as nn
from torch.autograd import Variable
import torchvision.datasets as datasets
import torchvision.transforms as transforms

from sklearn.model_selection import train_test_split
from sklearn.datasets import make_blobs

import progressbar
```

[23]:
```
plt.gcf().set_facecolor("white")
plt.rcParams.update({'font.size': 14})
```

```
<Figure size 432x288 with 0 Axes>
```

0.1 Convex Program Solver using CVXPY library

[24]:
```
def generate_Dmat(X, num_samples, input_dim, Dmat_max_itr):
    Dmat = np.empty((num_samples, 0))

    ## Finite approximation of all possible sign patterns
    for i in range(Dmat_max_itr):
        u = np.random.randn(input_dim, 1)
        Dmat = np.append(Dmat, (np.dot(X, u) >= 0) * 1.0, axis=1)
        Dmat = np.unique(Dmat, axis=1)

    print("Dmat Unique Shape:", Dmat.shape)

    return Dmat
```

[25]:
```
def optimize_using_cvxpy(X, y_gt, num_samples, input_dim, Dmat_max_itr,␣
↪beta=1e-4):
    Dmat = generate_Dmat(X, num_samples, input_dim, Dmat_max_itr)
    P = Dmat.shape[1]

    v = cp.Variable((input_dim, P))
    w = cp.Variable((input_dim, P))

    ## Below we use squared loss as a performance metric for binary␣
↪classification
    y_pred = cp.Parameter((num_samples, 1))
    y_pred = cp.sum(cp.multiply(Dmat, X @ (v - w)), axis=1)

    loss_val = 0.5 * cp.sum_squares(y_pred - y_gt) + beta * (cp.mixed_norm(v.T,␣
↪p=2, q=1) + cp.mixed_norm(w.T, p=2, q=1))

    constraints = []
    constraints += [cp.multiply((2*Dmat - np.ones((num_samples, P))), (X @ v))␣
↪>=0]
    constraints += [cp.multiply((2*Dmat - np.ones((num_samples, P))), (X @ w))␣
↪>=0]

    optimizer = cp.Problem(cp.Minimize(loss_val), constraints)
    optimizer.solve()
    opt_val = optimizer.value

    print("Convex program objective value (eq (8)): ", opt_val)

    return opt_val
```

0.2 Using SGD to train a 2-layer ReLU neural network.

[26]:
```
class FCNetwork(nn.Module):
    def __init__(self, input_dim, num_hidden_units, num_classes):
        self.num_classes = num_classes
        super(FCNetwork, self).__init__()

        self.layer1 = nn.Sequential(nn.Linear(input_dim, num_hidden_units,␣
↪bias=True), nn.ReLU())
        self.layer2 = nn.Linear(num_hidden_units, num_classes-1, bias=True)

    def forward(self, x):
        num_samples = x.shape[0]
        x = x.reshape(num_samples, -1)
```

```
        output = self.layer2(self.layer1(x))
        return output
```

[27]:
```
def compute_loss(h_theta_x, y_gt, model, beta=1e-4):
    loss = 0.5 * torch.linalg.norm(h_theta_x - y_gt, 2)**2

    for layer, theta in enumerate(model.parameters()):
        if layer == 0:
            loss += 0.5 * beta * torch.linalg.norm(theta, 'fro')**2
        elif layer == 1:
            if len(theta.shape) == 1:
                loss += 0.5 * beta * torch.linalg.norm(theta, 2)**2
            elif len(theta.shape) == 2:
                loss += 0.5 * beta * sum([torch.linalg.norm(theta[:, j], 1)**2␣
↪for j in range(theta.shape[1])])

    return loss
```

[28]:
```
def generate_data_binary_classification(num_samples, input_dim, num_classes=2):
    assert(num_classes == 2)

    # generate data randomly from a uniform/standard normal N(0,1) dist
    X = np.random.randn(num_samples, input_dim)

    # Rule 1: generate binary labels. If x lies outside unit norm ball, then␣
↪assign +1, else assign -1
    Y = ((np.linalg.norm(X[:, 0:input_dim], axis=1) > 1) - 0.5) * 2

    # Rule 2: generate binary labels for input dim=1. If x >= 0, assign +1,␣
↪else assign -1
    # Y = ((X[:, 0:input_dim] >= 0) - 0.5) * 2

    # Rule 3: Y = np.random.choice([-1,1], (num_samples,))

    # Y = np.reshape(Y, (num_samples, 1))

    return X, Y
```

[29]:
```
def generate_data_sklearn(num_samples, input_dim, num_classes):
    (X, Y) = make_blobs(n_samples=num_samples, n_features=input_dim,␣
↪centers=num_classes,
                        cluster_std=2.5, random_state=95)

    return X, Y
```

[30]:
```
def get_next_batch(X, Y, batch_size):
    num_samples = X.shape[0]

    # loop over the dataset
    for i in range(0, num_samples, batch_size):
        # yield a tuple of the current batched data and labels
        yield(X[i:i + batch_size], Y[i:i + batch_size])
```

[31]:
```
def sgd_solver(X_train, Y_train, X_test, Y_test, input_dim, num_hidden_units,␣
 ↪num_classes, num_epochs, batch_size,
               step_size=1e-3, beta=1e-4, schedule=0):

    num_train_samples = X_train.shape[0]
    num_test_samples = X_test.shape[0]

#     print("[INFO] Input dim: {}, Num hidden units: {}, Num classes: {}".
 ↪format(input_dim, num_hidden_units, num_classes))

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
#     print("[INFO] Training using {}...".format(device))

    layer2_relu_model = FCNetwork(input_dim, num_hidden_units, num_classes).
 ↪to(device)
#     print("[INFO] 2-layer ReLU Model:", layer2_relu_model)

#     solver_type = 'sgd'
    solver_type = 'adam'
    if solver_type == 'sgd':
        optimizer = torch.optim.SGD(layer2_relu_model.parameters(),␣
 ↪lr=step_size, momentum=0.9)
    elif solver_type == 'adam':
        optimizer = torch.optim.Adam(layer2_relu_model.parameters(),␣
 ↪lr=step_size)

    # lists for storing loss and accuracy
    train_loss_lst = []
    train_acc_lst = []
    test_loss_lst = []
    test_acc_lst = []

    if num_test_samples > 0:
        x_test_batch = Variable(X_test).to(device)
        y_test_batch = Variable(Y_test).to(device)

        h_theta_x = torch.reshape(layer2_relu_model(x_test_batch),␣
 ↪(num_test_samples,))
        y_pred = torch.add((h_theta_x >= 0), -0.5) * 2
```

```
        loss_val = compute_loss(h_theta_x, y_test_batch, layer2_relu_model,␣
↪beta)
        frac_corr_pred = torch.eq(y_pred, y_test_batch).float().sum() /␣
↪num_test_samples

        test_loss_lst.append(loss_val.item())
        test_acc_lst.append(frac_corr_pred)

    if schedule == 1:
        scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer,␣
↪verbose=True, factor=0.5, eps=1e-12)
    elif schedule == 2:
        scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, 0.99)

    itr = 0
    for epoch in range(num_epochs):
        for (x_batch, y_batch) in get_next_batch(X_train, Y_train, batch_size):
            x_batch = Variable(x_batch).to(device)
            y_batch = Variable(y_batch).to(device)

            h_theta_x = torch.reshape(layer2_relu_model(x_batch), (batch_size,))
            y_pred = torch.add((h_theta_x >= 0), -0.5) * 2

            loss_val = compute_loss(h_theta_x, y_batch, layer2_relu_model, beta)
            frac_corr_pred = torch.eq(y_pred, y_batch).float().sum() /␣
↪batch_size

            optimizer.zero_grad()   # make the gradients=0 before each backprop
            loss_val.backward()
            optimizer.step()

            train_loss_lst.append(loss_val.item())
            train_acc_lst.append(frac_corr_pred)

            itr += 1

        # compute test loss and accuracy every epoch
        if num_test_samples > 0:
            h_theta_x = torch.reshape(layer2_relu_model(x_test_batch),␣
↪(num_test_samples,))
            y_pred = torch.add((h_theta_x >= 0), -0.5) * 2

            loss_val = compute_loss(h_theta_x, y_test_batch, layer2_relu_model,␣
↪beta)
```

```
            frac_corr_pred = torch.eq(y_pred, y_test_batch).float().sum() /␣
 ↪num_test_samples

            test_loss_lst.append(loss_val.item())
            test_acc_lst.append(frac_corr_pred)

#         if epoch % 1000 == 0:
#             print("[INFO] Epoch {}/{} - Train loss: {:0.3f}, Train acc: {:0.
 ↪3f}, Test loss: {:0.3f}, Test acc: {:0.3f}".format(
#                 epoch, num_epochs, train_loss_lst[itr-1],␣
 ↪train_acc_lst[itr-1], test_loss_lst[epoch+1], test_acc_lst[epoch+1] ))

#         if schedule > 0:
#             scheduler.step(train_loss_lst[itr-1])

    return train_loss_lst, train_acc_lst, test_loss_lst, test_acc_lst
```

[32]:
```
def solve_nonconvex_SGD(input_dim, num_hidden_units, num_classes, num_samples,
                        num_epochs, batch_size, split_data, num_trials,␣
 ↪is_q8=False):

    # generate a 2-class classification problem with some data points
    # where each data point is a n-dim feature vector

#     print("[INFO] Generating data...")
    # X, Y = generate_data_sklearn(num_samples, input_dim, num_classes)
    X, Y = generate_data_binary_classification(num_samples, input_dim,␣
 ↪num_classes)

    if split_data == True:
        # create training and testing splits
        (X_train, X_test, Y_train, Y_test) = train_test_split(X, Y, test_size=0.
 ↪5, random_state=95)
    else:
        X_train = np.copy(X)
        Y_train = np.copy(Y)
        X_test = np.empty(0)
        Y_test = np.empty(0)

    num_train_samples = X_train.shape[0]
    num_test_samples = X_test.shape[0]

    # convert numpy arrays to PyTorch tensors
    X_train_tensor = torch.from_numpy(X_train).float()
    Y_train_tensor = torch.from_numpy(Y_train).float()
    X_test_tensor = torch.from_numpy(X_test).float()
```

```
    Y_test_tensor = torch.from_numpy(Y_test).float()
#     print("Generated tensor size:", X_train_tensor.shape, Y_train_tensor.
 ↪shape, X_test_tensor.shape,
#           Y_test_tensor.shape)

    fig = plt.figure(figsize=(16,14))
    ax1 = fig.add_subplot(2,2,1)
    ax1.set_yscale('log')
    ax1.title.set_text("Train Obj vs Iteration")
    plt.xlabel("Iteration")
    plt.ylabel("Objective Value")
    plt.grid(linestyle='--')

    ax2 = fig.add_subplot(2,2,2)
    ax2.title.set_text("Train Acc vs Iteration")
    plt.xlabel("Iteration")
    plt.ylabel("Train Accuracy")
    plt.grid(linestyle='--')

    ax3 = fig.add_subplot(2,2,3)
    ax3.set_yscale('log')
    ax3.title.set_text("Test Obj vs Iteration")
    plt.xlabel("Iteration")
    plt.ylabel("Objective Value")
    plt.grid(linestyle='--')

    ax4 = fig.add_subplot(2,2,4)
    ax4.title.set_text("Test Acc vs Iteration")
    plt.xlabel("Iteration")
    plt.ylabel("Test Accuracy")
    plt.grid(linestyle='--')

    plt.suptitle("Plot for m = "+str(num_hidden_units)+", n=␣
 ↪"+str(num_train_samples)+", d = "+str(input_dim))

    lw = 2
#     color_lst = ['red', 'blue', 'brown', 'yellow', 'green', 'lightblue',␣
 ↪'darkred']

    for trial in range(num_trials):
        train_loss_lst, train_acc_lst, test_loss_lst, test_acc_lst =␣
 ↪sgd_solver(X_train_tensor, Y_train_tensor,
                                                                        ␣
 ↪X_test_tensor, Y_test_tensor,
                                                                        ␣
 ↪input_dim, num_hidden_units,
```

```
                                                                        ␣
↪num_classes, num_epochs, batch_size)

        line, = ax1.plot(train_loss_lst, linewidth=lw, label='Trial␣
↪#'+str(trial+1))
        line, = ax2.plot(train_acc_lst, linewidth=lw, label='Trial␣
↪#'+str(trial+1))
        line, = ax3.plot(test_loss_lst, linewidth=lw, label='Trial␣
↪#'+str(trial+1))
        line, = ax4.plot(test_acc_lst, linewidth=lw, label='Trial␣
↪#'+str(trial+1))

    # generate results using convex programming CVXPY
    X_train_new = np.append(X_train, np.ones((num_train_samples, 1)), axis=1)
    cvx_opt_val = optimize_using_cvxpy(X_train_new, Y_train, num_train_samples,␣
↪input_dim+1, Dmat_max_itr=10000,
                                       beta=1e-4)

    ax1.axhline(cvx_opt_val, color='black', linewidth=lw, linestyle='-',␣
↪label='CVX-opt')

    if is_q8 == True:
        cvx_opt_val_approx = optimize_using_cvxpy(X_train_new, Y_train,␣
↪num_train_samples, input_dim+1,
                                                  Dmat_max_itr=100, beta=1e-4)

        ax1.axhline(cvx_opt_val_approx, color='darkred', linewidth=lw,␣
↪linestyle='-', label='CVX-opt-approx')

    ax1.legend(prop={'size': 10})
    ax2.legend(prop={'size': 10})
    ax3.legend(prop={'size': 10})
    ax4.legend(prop={'size': 10})

    plt.show()
```

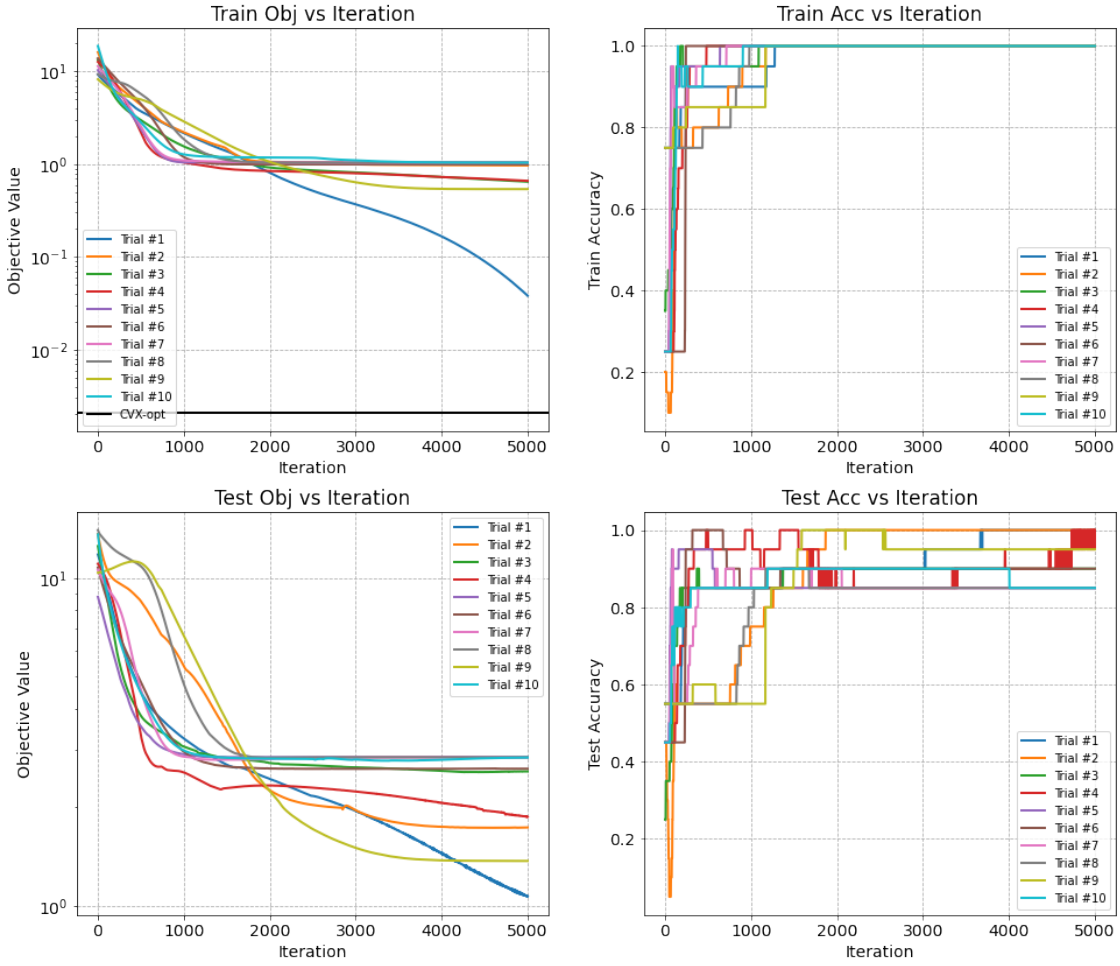
0.2.1 Q7. Implement the first experiment from Fig 2 in [1]. Generate the data randomly. In particular for a small n (the sample size) and m = 5 (small number of hidden neurons), use the Stochastic Gradient Descent method to solve the non-convex problem m with different initializations. Plot the training objective value vs the number of iterations. Next, solve the convex program with CVX/CVXPY and plot its optimal value. What do you observe? Does the SGD always converge to the optimal value of the convex program? Perform the same experiment with larger values of m. Do your observations agree with the theoretical guarantees?

- We generated random IID data of dimension 1 sampled from standard normal distribution $\mathcal{N}(0, 1)$. We used training sample size $n = 20$ and test sample size $n = 20$ (small n) - both data subsets sampled from the same distribution, and number of hidden neurons $m = 5, 15, 50, 100$.
- SGD does not always converge to the optimal value obtained by the convex optimizer (CVXPY library) as the training loss curve of SGD is always above the optimal value. This might be due to SGD being stuck at local optimal solutions as the SGD is applied over a non-convex loss function. From these experiments, for small m, SGD is unable to minimize training loss beyond a certain point. In some cases, it might seem that SGD loss curve is going below the convex optimal value. This is due to the fact that in the convex optimization, we are using an approximate value of P due to high computational cost. Increasing the number of random samples of u vectors to $1e^8$ might end up in getting exact P value.
- Yes, our observations agree with the theoretical guarantees in paper [1]. As we increase the number of hidden neurons m from 5, 15, 50 upto 100, we observe that more and more SGD trials with random initializations are converging approximately very close to the optimal value obtained by the convex optimizer (CVXPY solver). This shows that as we increase m, the learning capacity of the 2-layer ReLU network improves and therefore, the training loss reaches closer to the global optimal value. For small m (let's say m=5, 15), and n = 20, $1 \leq m^* \leq n$, therefore, the condition for strong duality in Theorem 1 $m \geq m^*$ might not hold. But with large m (say m=50, 100), the strong duality condition is definitely fulfilled, and hence we see most of the random trials converging very close to the convex optimal loss value.

[33]:
```
input_dim = 1
num_hidden_units = 5
num_classes = 2
num_samples = 40
num_epochs = 5000
batch_size = 20
split_data = True
num_trials = 10

solve_nonconvex_SGD(input_dim, num_hidden_units, num_classes, num_samples,
                    num_epochs, batch_size, split_data, num_trials)
```

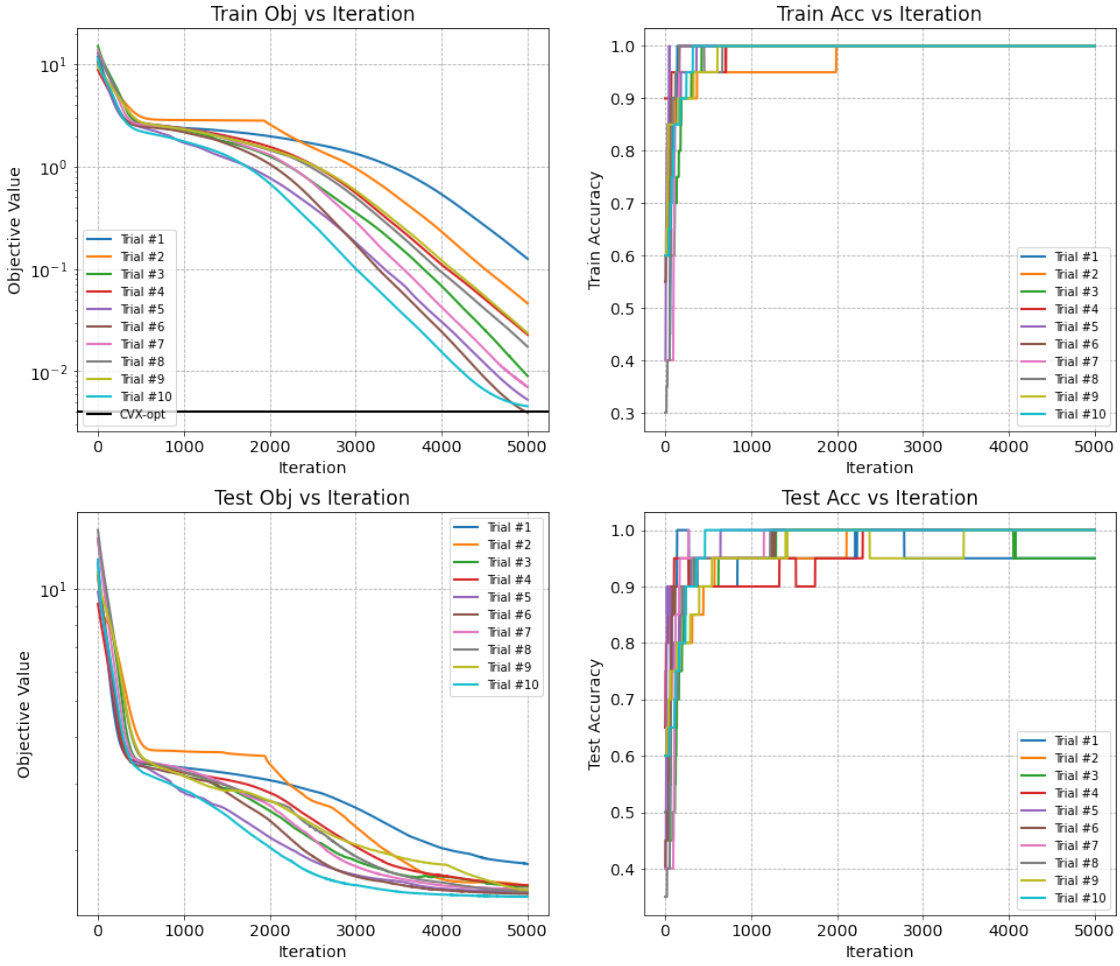
```
Dmat Unique Shape: (20, 40)
Convex program objective value (eq (8)):  0.0020884593772920092
```



[34]:
```
input_dim = 1
num_hidden_units = 15
num_classes = 2
num_samples = 40
num_epochs = 5000
batch_size = 20
split_data = True
num_trials = 10

solve_nonconvex_SGD(input_dim, num_hidden_units, num_classes, num_samples,
                    num_epochs, batch_size, split_data, num_trials)
```

```
Dmat Unique Shape: (20, 40)
Convex program objective value (eq (8)):  0.004005579677215383
```

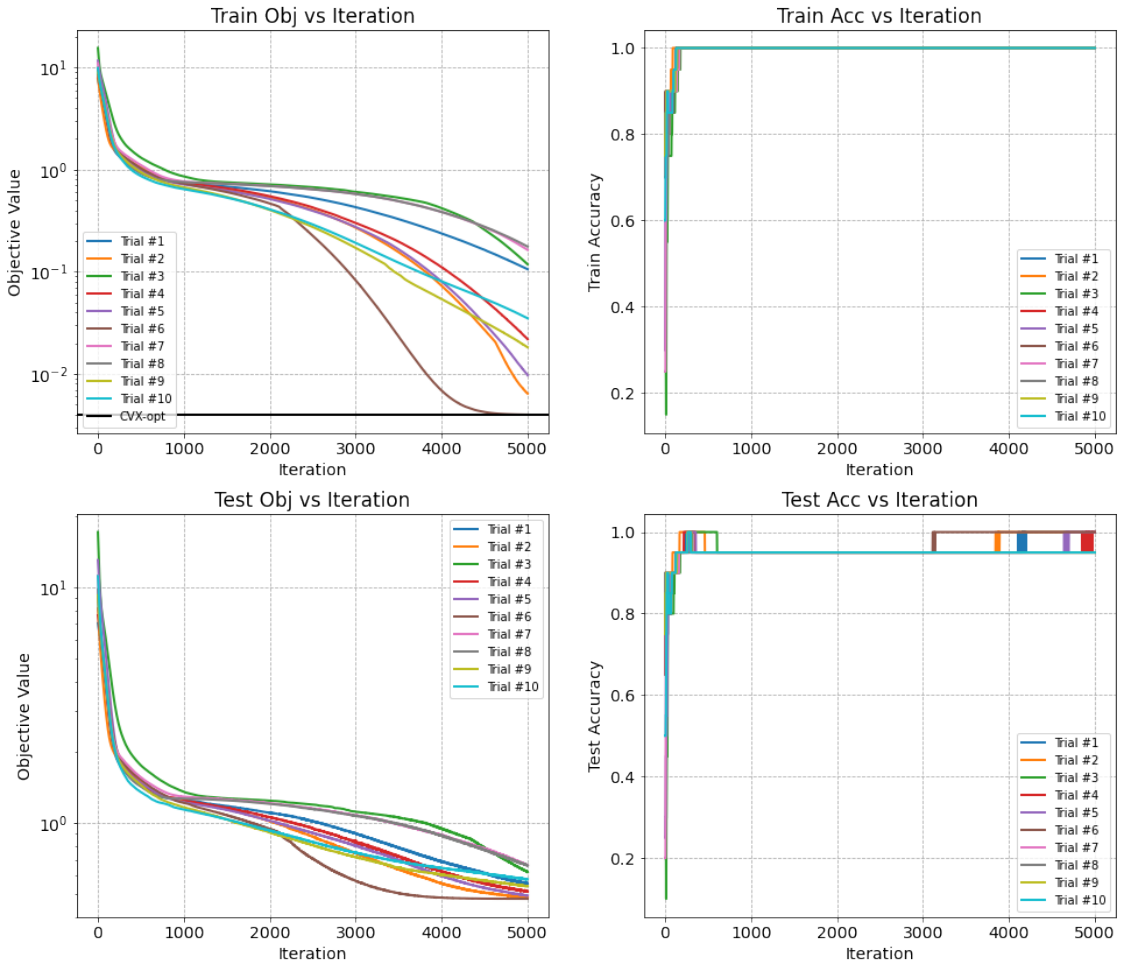


[35]:
```
input_dim = 1
num_hidden_units = 50
num_classes = 2
num_samples = 40
num_epochs = 5000
batch_size = 20
split_data = True
num_trials = 10

solve_nonconvex_SGD(input_dim, num_hidden_units, num_classes, num_samples,
                    num_epochs, batch_size, split_data, num_trials)
```

```
Dmat Unique Shape: (20, 40)
Convex program objective value (eq (8)):  0.003980895820738493
```

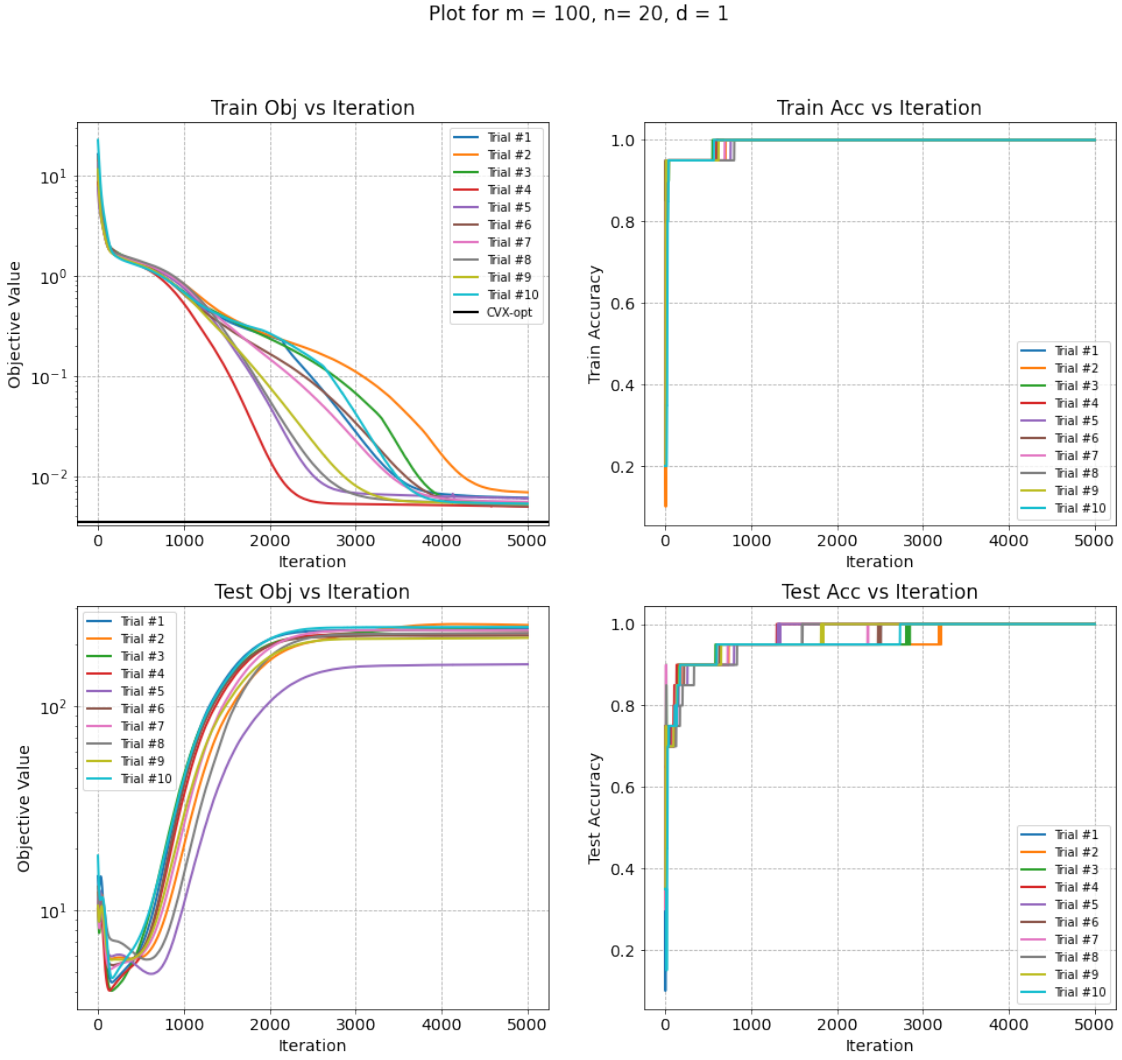
Plot for m = 50, n= 20, d = 1



[36]:
```
input_dim = 1
num_hidden_units = 100
num_classes = 2
num_samples = 40
num_epochs = 5000
batch_size = 20
split_data = True
num_trials = 10

solve_nonconvex_SGD(input_dim, num_hidden_units, num_classes, num_samples,
                    num_epochs, batch_size, split_data, num_trials)
```

```
Dmat Unique Shape: (20, 40)
Convex program objective value (eq (8)):  0.003515337432210059
```



0.2.2 Q8. Replicate this experiment for a larger value of n and an appropriate m. Plot the objective value against the number of iterations. How does this experiment differ from the case for small n? In addition to the original convex program, solve the approximate convex program described in Remark 3.3. Compare the objective value of the approximate version with both SGD and convex program. What is the advantage of solving the approximate convex program?

- For this experiment, we increased our training data sample size (n) from 20 to 500 and varied m to be 5, 10, 50, 100.
- The convex problem can be approximated by sampling a set of diagonal matrices from P unique diagonal matrices. According to the authors, this approximation to solve convex problem works extremely well and often better than backpropagation. This method has advantage over the original problem as it can be solved faster and sometimes it is the only

way to solve convex problem when data or network size is large.

- We observe that for small number of hidden neurons n, the SGD loss curve is above both the exact convex and approximate convex optimal values. But for larger m, the SGD loss curve crosses and goes below the approximate convex optimal value (in some cases) but still stays above the exact convex optimal value. This is reasonable as increasing m increases the learning and representation capacity of the NN. In any case, approximate convex optimal value is still better than SGD with small m values.
- **Advantage of solving the approximate convex program:** The convex problem requires computing P diagonal matrices D_i corresponding to the total number of partitions P. For each diagonal matrix D_i, we sample $u \in R^d$ and compute $1[Xu \geq 0]$ which corresponds to the diagonal entries of D_i. Many of these diagonal matrices are repeated while we sample random u vector. In order to estimate diagonal matrices corresponding to all P partitions, we need to sample u large number of times ($\sim 10^6$). In addition, large number of these diagonal matrices also requires large amount of time for the CVXPY library to reach optimal solution. Therefore, solving approximate convex program requires us to estimate only a small subset of these partitions making the convex optimization solver much faster compared to solving the convex exact version.

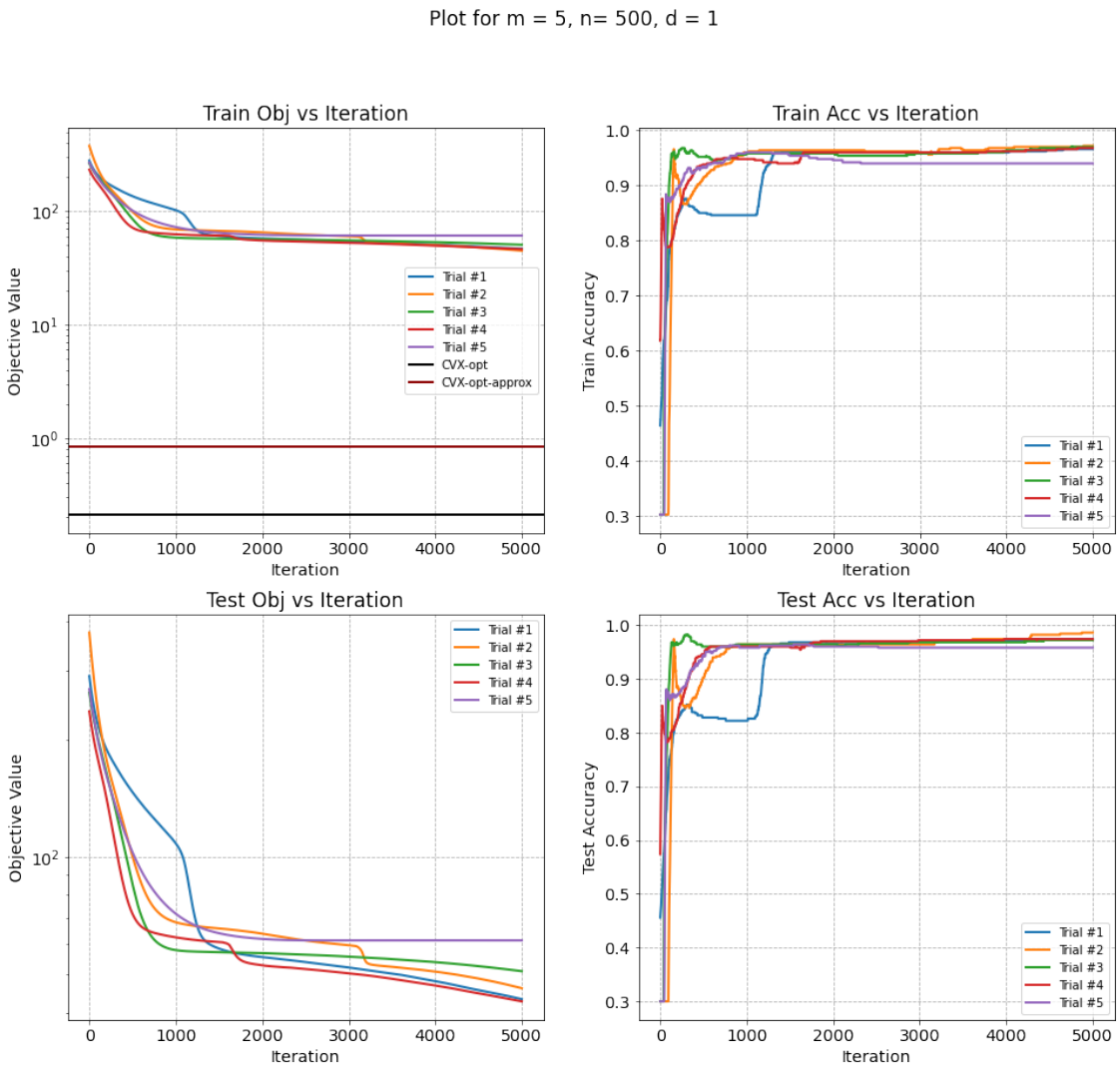
[37]:
```
input_dim = 1
num_hidden_units = 5
num_classes = 2
num_samples = 1000
num_epochs = 5000
batch_size = 500
split_data = True
num_trials = 5

solve_nonconvex_SGD(input_dim, num_hidden_units, num_classes, num_samples,
                    num_epochs, batch_size, split_data, num_trials, is_q8=True)
```

```
Dmat Unique Shape: (500, 869)

/Users/siriusA/anaconda3/lib/python3.8/site-
packages/cvxpy/problems/problem.py:1337: UserWarning: Solution may be
inaccurate. Try another solver, adjusting the solver settings, or solve with
verbose=True for more information.
  warnings.warn(

Convex program objective value (eq (8)):  0.2123580629922117
Dmat Unique Shape: (500, 73)
Convex program objective value (eq (8)):  0.8429447811590626
```

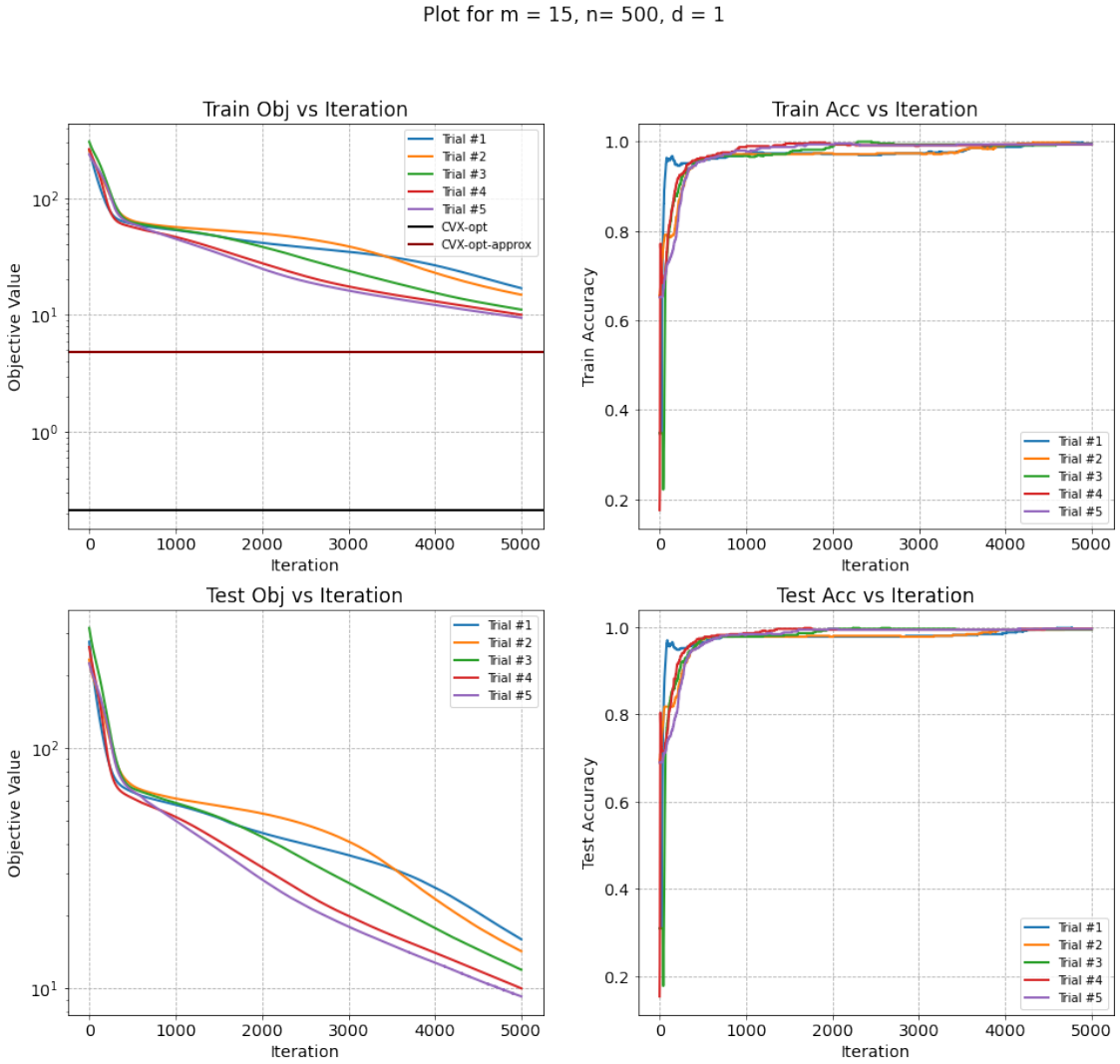


[38]:
```
input_dim = 1
num_hidden_units = 15
num_classes = 2
num_samples = 1000
num_epochs = 5000
batch_size = 500
split_data = True
num_trials = 5

solve_nonconvex_SGD(input_dim, num_hidden_units, num_classes, num_samples,
                    num_epochs, batch_size, split_data, num_trials, is_q8=True)
```

```
Dmat Unique Shape: (500, 877)
Convex program objective value (eq (8)):  0.21376574802201603
Dmat Unique Shape: (500, 75)
```

```
Convex program objective value (eq (8)):  4.851033055198961
```

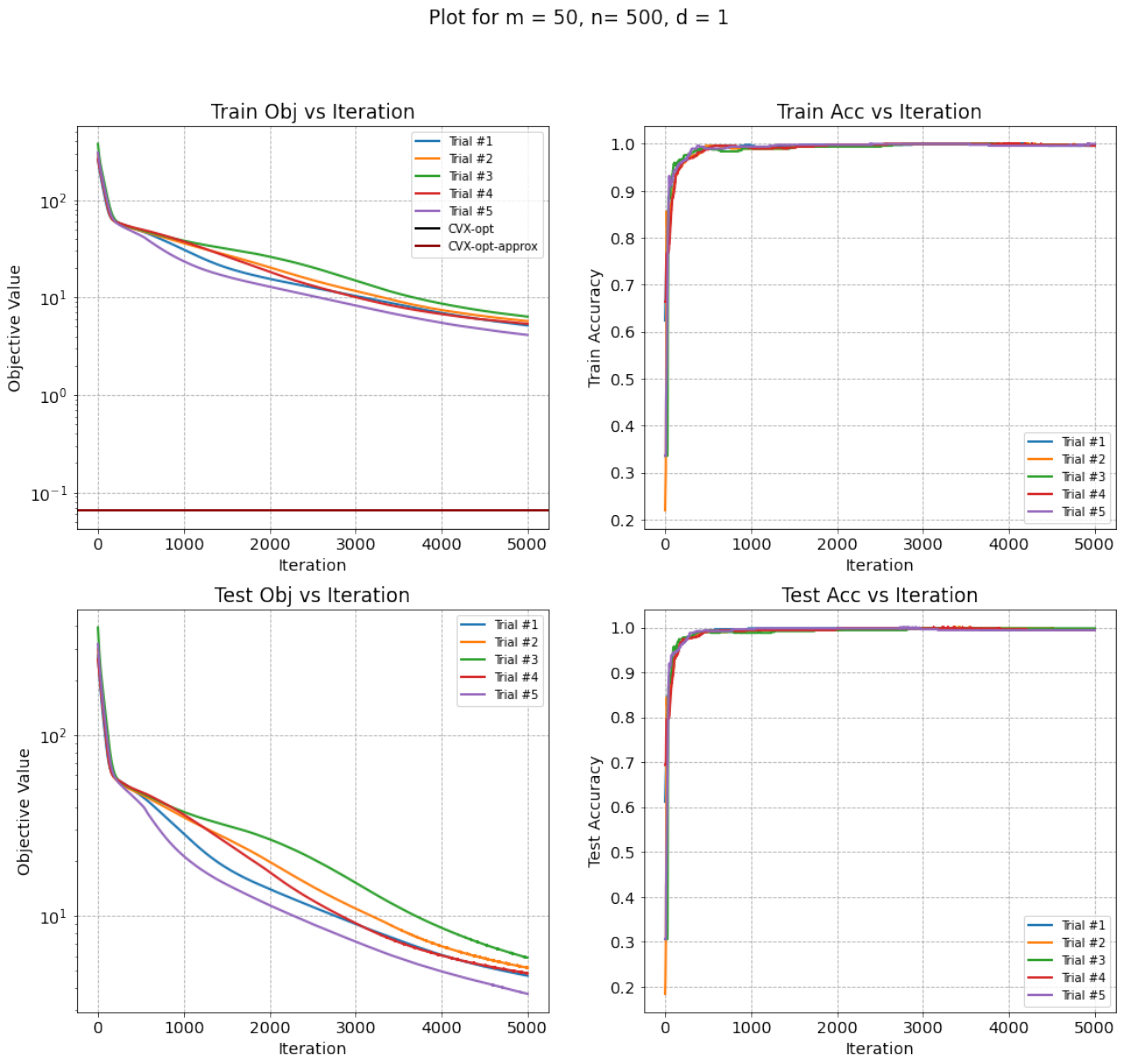


[39]:
```
input_dim = 1
num_hidden_units = 50
num_classes = 2
num_samples = 1000
num_epochs = 5000
batch_size = 500
split_data = True
num_trials = 5

solve_nonconvex_SGD(input_dim, num_hidden_units, num_classes, num_samples,
                    num_epochs, batch_size, split_data, num_trials, is_q8=True)
```

```
Dmat Unique Shape: (500, 899)
```

```
Convex program objective value (eq (8)):  0.06565978260525755
Dmat Unique Shape: (500, 72)
Convex program objective value (eq (8)):  0.06611374904851539
```

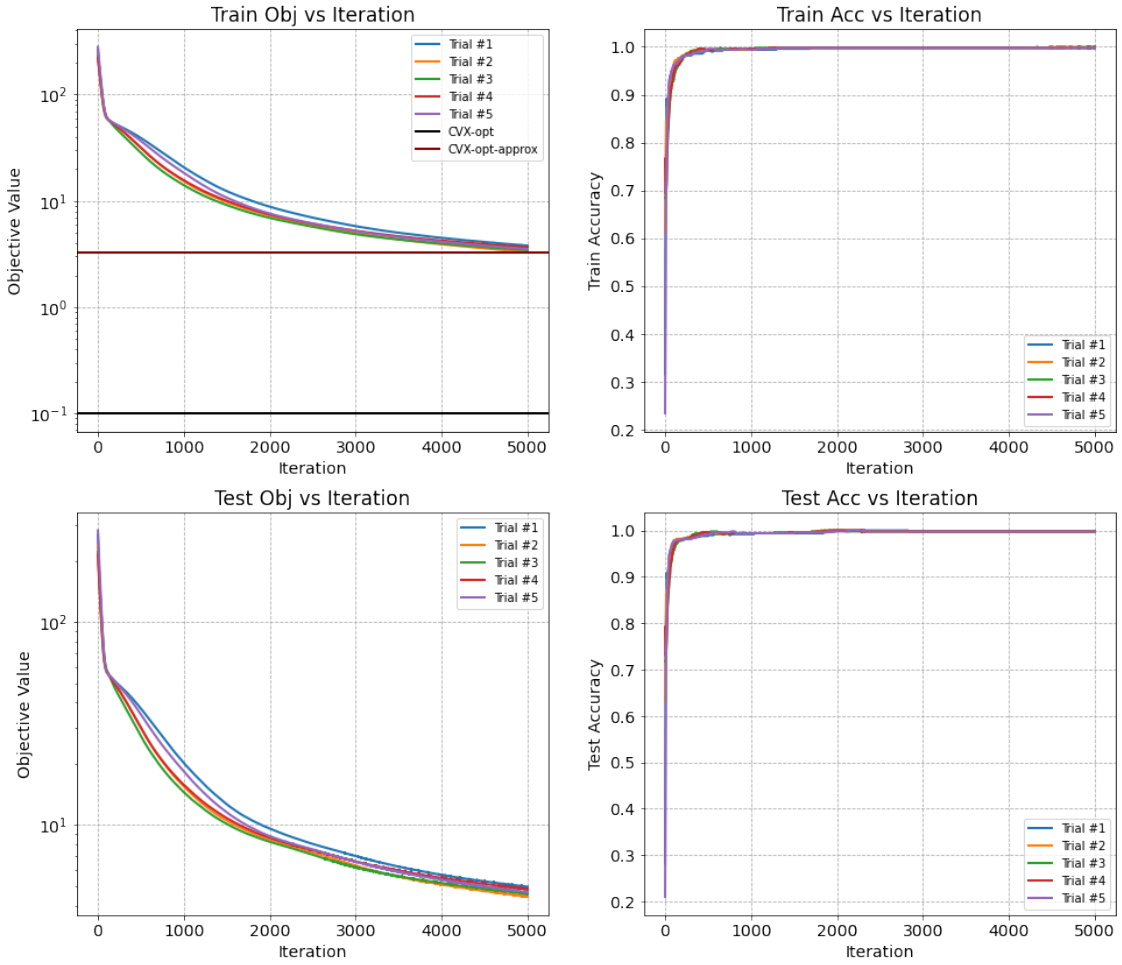


[40]:
```
input_dim = 1
num_hidden_units = 100
num_classes = 2
num_samples = 1000
num_epochs = 5000
batch_size = 500
split_data = True
num_trials = 5

solve_nonconvex_SGD(input_dim, num_hidden_units, num_classes, num_samples,
                    num_epochs, batch_size, split_data, num_trials, is_q8=True)
```

```
Dmat Unique Shape: (500, 892)
Convex program objective value (eq (8)):  0.10085867445692462
Dmat Unique Shape: (500, 67)
Convex program objective value (eq (8)):  3.26831115966998
```

Plot for m = 100, n= 500, d = 1



0.3 Experiment - Binary Classification Task on CIFAR - 10

0.3.1 Q9. Perform an experiment for the binary classification task on CIFAR-10 dataset. Choose any two classes you prefer. Choose the value of m appropriately. Train the network using SGD on the non-convex problem for different initializations. Solve the approximate convex program with CVX/CVPY. Plot objective value vs iterations. Evaluate the performance of trained networks on the test set, and plot test accuracy vs iterations. Comment on your observations.

- We choose the class labels - 0 and 1 - and extract 5000x2 training and 1000x2 test samples corresponding to these 2 classes. Each data sample is linearized into a 3072-dim vector. We use m=100 hidden neurons.
- Running the convex solver CVXPY on 10000 training samples requires high computation power and it lead to my PC running out of memory. I tried running it on UCSD DSMLP cluster but there also the CVXPY solver did not converge and seems to be taking infinite amount of time. Therefore, I sampled small amounts of data for each of the 2 classes and performed SGD training and convex optimization with this small batch of CIFAR-10 data.
- For our CIFAR-10 experiment, we use all the 3072 features. For each class - 0,1 - we used 100 training samples and 50 test samples per class. Also, m = 100 and n = 200 which might be the reason for the big gap between SGD loss and convex loss.
- From the plot, we can see that the training objective curve is well above the convex optimal value. The test loss is also decreasing and saturates after some iterations. In addition, the training and test accuracy plots further assure that there is no overfitting of data.

[41]:
```
directory = os.path.dirname(os.path.realpath("./"))
normalize = transforms.Normalize(mean=[0.507, 0.487, 0.441], std=[0.267, 0.256,␣
↪0.276])

train_dataset = datasets.CIFAR10(directory, train=True, download=True,
                                 transform=transforms.Compose([transforms.
↪ToTensor(), normalize,]))

test_dataset = datasets.CIFAR10(directory, train=False, download=True,
                                transform=transforms.Compose([transforms.
↪ToTensor(), normalize,]))
```

```
Files already downloaded and verified
Files already downloaded and verified
```

[42]:
```
# data extraction
print('Extracting CIFAR-10 dataset')

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=50000,␣
↪shuffle=True,
                                           pin_memory=True, sampler=None)
```

```
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=10000,␣
↪shuffle=False,
                                          pin_memory=True, sampler=None)

for train_x, train_y in train_loader:
    pass

for test_x, test_y in test_loader:
    pass

train_x = train_x.view(train_x.shape[0], -1)
test_x = test_x.view(test_x.shape[0], -1)

index_01_class = torch.where(train_y <= 1)[0]
X_train = train_x[index_01_class]
Y_train = train_y[index_01_class]

index_01_class = torch.where(test_y <= 1)[0]
X_test = test_x[index_01_class]
Y_test = test_y[index_01_class]

print("Train data size: (", X_train.shape, Y_train.shape, ") \nTest data size:␣
↪(", X_test.shape, Y_test.shape, ")")

# print(np.unique(Y_train), np.unique(Y_test))
Y_train = (Y_train - 0.5) * 2
Y_test = (Y_test - 0.5) * 2
# print(np.unique(Y_train), np.unique(Y_test))
```

```
Extracting CIFAR-10 dataset
Train data size: ( torch.Size([10000, 3072]) torch.Size([10000]) )
Test data size: ( torch.Size([2000, 3072]) torch.Size([2000]) )
```

[18]:
```
index_1_class = torch.where(Y_train == 1)[0]
index_0_class = torch.where(Y_train == -1)[0]

index_1_class = index_1_class[0:100]
index_0_class = index_0_class[0:100]
index_01_class = torch.cat((index_1_class, index_0_class), dim=0)
print(index_1_class.shape, index_0_class.shape, index_01_class.shape)

X_train_short = X_train[index_01_class]
Y_train_short = Y_train[index_01_class]

index_1_class = torch.where(Y_test == 1)[0]
index_0_class = torch.where(Y_test == -1)[0]
```

```
index_1_class = index_1_class[0:50]
index_0_class = index_0_class[0:50]
index_01_class = torch.cat((index_1_class, index_0_class), dim=0)
print(index_1_class.shape, index_0_class.shape, index_01_class.shape)

X_test_short = X_test[index_01_class]
Y_test_short = Y_test[index_01_class]

print("Train data size: (", X_train_short.shape, Y_train_short.shape, ") \nTest␣
 ↪data size: (", X_test_short.shape, Y_test_short.shape, ")")
```

```
torch.Size([100]) torch.Size([100]) torch.Size([200])
torch.Size([50]) torch.Size([50]) torch.Size([100])
Train data size: ( torch.Size([200, 3072]) torch.Size([200]) )
Test data size: ( torch.Size([100, 3072]) torch.Size([100]) )
```

[46]:
```
input_dim = X_train_short.shape[1]
num_hidden_units = 100
num_classes = 2
num_train_samples = X_train_short.shape[0]
num_epochs = 1000
batch_size = num_train_samples // 1
num_trials = 5
step_size = 1e-3
beta = 1e-4

fig = plt.figure(figsize=(16,14))
ax1 = fig.add_subplot(2,2,1)
ax1.set_yscale('log')
ax1.title.set_text("Train Obj vs Iteration")
plt.xlabel("Iteration")
plt.ylabel("Objective alue")
plt.grid(linestyle='--')

ax2 = fig.add_subplot(2,2,2)
ax2.title.set_text("Train Acc vs Iteration")
plt.xlabel("Iteration")
plt.ylabel("Train Accuracy")
plt.grid(linestyle='--')

ax3 = fig.add_subplot(2,2,3)
ax3.set_yscale('log')
ax3.title.set_text("Test Obj vs Iteration")
plt.xlabel("Iteration")
plt.ylabel("Objective Value")
plt.grid(linestyle='--')
```

```
ax4 = fig.add_subplot(2,2,4)
ax4.title.set_text("Test Acc vs Iteration")
plt.xlabel("Iteration")
plt.ylabel("Test Accuracy")
plt.grid(linestyle='--')

lw = 2
# color_lst = ['red', 'blue', 'lightcoral', 'lime', 'darkred', 'lightblue',␣
↪'green']

for trial in range(num_trials):
    train_loss_lst, train_acc_lst, test_loss_lst, test_acc_lst =␣
↪sgd_solver(X_train_short, Y_train_short,
                                                                    ␣
↪X_test_short, Y_test_short,
                                                                    ␣
↪input_dim, num_hidden_units,
                                                                    ␣
↪num_classes, num_epochs, batch_size,
                                                                    ␣
↪step_size, beta)

    line, = ax1.plot(train_loss_lst, linewidth=lw, label='Trial #'+str(trial+1))
    line, = ax2.plot(train_acc_lst, linewidth=lw, label='Trial #'+str(trial+1))
    line, = ax3.plot(test_loss_lst, linewidth=lw, label='Trial #'+str(trial+1))
    line, = ax4.plot(test_acc_lst, linewidth=lw, label='Trial #'+str(trial+1))

print("SGD Training Complete !!! --> Starting CVXPY Optim")

X_train_new = np.append(X_train_short, np.ones((num_train_samples, 1)), axis=1)
cvx_opt_val = optimize_using_cvxpy(X_train_new, Y_train_short,␣
↪num_train_samples, input_dim+1,
                                   Dmat_max_itr=10, beta=beta)

ax1.axhline(cvx_opt_val, color='black', linewidth=2, linestyle='-',␣
↪label='CVX-optimal')

ax1.legend(prop={'size': 10})
ax2.legend(prop={'size': 10})
ax3.legend(prop={'size': 10})
ax4.legend(prop={'size': 10})

plt.show()
```

```
SGD Training Complete !!! --> Starting CVXPY Optim
Dmat Unique Shape: (200, 10)
```

```
Convex program objective value (eq (8)):  8.13348744996559e-05
```