Convex Optimization: Homework 4

Akshay Sharma (akshaysh)

Answer 1

1.1

Over-parameterization helps in solving non-convex optimization problem because it gives more degrees of freedom to the algorithm to tweak. This allows easier escape from bad second order local minimas.

1.2

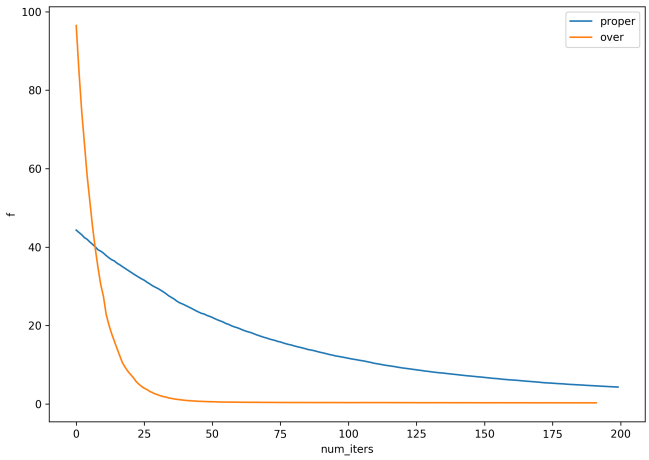

Figure 1: Loss value vs number of iterations for proper and over parameterization

The network shows the loss value after each update of the network, i.e. after each mini-batch. The mini-batch was of size 32, and the loss function was evaluated after the update on a static test set of 1000 samples randomly drawn from the standard normal distribution. The test set was kept the same for all the values of m.
The training was stopped as soon as either 200 updates have been made, or the absolute change in the loss value before and after an update fell below $1e-3$.
Figure 1, clearly shows that $m = 200$, achieves a lower minima than $m = 20$, and also converges faster.

Bonus

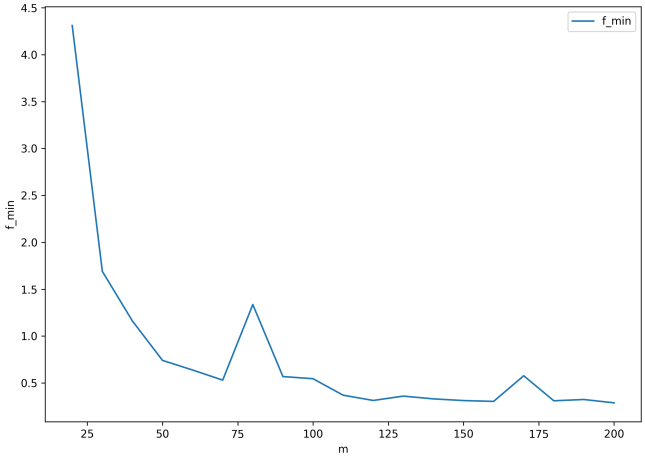

Figure 2: Minimum loss value achieved vs m

To analyse the effect of m, I plotted the minima achieved by all every 10^{th} value of m, starting from $m = 20$, and ending at $m = 200$. The above figure clearly shows that over parameterization helps in reaching better minimas. Though the improvement decreases after m has been increased enough. Based on this plot, one should surely use overparameterization, but only upto a certain limit, as their is not additional benefit on increasing the parameters and will only increase the evaluation time.

Answer 2

Neural networks for image classification are trained using SGD which has the following update rule:

$$x_{t+1} = x_t - \eta \widetilde{\nabla} f(x_t)$$

This is a noisy update, and changing the learning rate eventually leads to changing the effect of this noise on the update.
Given we have much more neurons to learn than the number of training examples used, the neural network can just memorise low variance or less noisy features of the data, and start classifying images according to that. It will practically memorize those low variance features in the initial few updates. This means that all the images in the training data which can indeed be classified using that low variance feature will be correctly classified and hence the network will not get any new information from them. If the learning rate is kept low initially, the neural network will get stuck in that local minima where the training error will be low because of memorising those low variance features but the generalization error will be large. A higher initial learning rate can help in such scenarios as it will scale up the noise in the parameter updates, which will prevent the neurons to memorise any low variance features, as their parameters will keep changing. This will help in reaching a network state which might have the same training error as that achieved in the low initial learning rate case, but will have lower generalization error.

Answer 3

Sparse coding case, $M \in \mathrm{R}^{dXd}$, $MM^T = I$, $x \in R\mathrm{R}^d$.

$$x = Mz$$

where z comes from a sparse distribution.

Let the true labeling function be,

$$y = sign(< w^*, z >)$$

If the data used is clean, i.e. no adversarial samples, we can learn to classify the data using a linear classifier,

$$L(x) =< Mw^*, x >$$

We can show that the data can be classified using $L(x)$ as,

$$L(x) =< Mw^*, x >=< Mw^*, Mz >= w^{*T}M^TMz$$

$$L(x) = w^{*T}z =< w^*, z >$$

To test the robustness of $L(x)$ to adversarial samples, assume $w_i^* = 1$, $\forall i \in [d]$ This means,

$$||Mw^*||_2 = ||Iw^*||_2 = ||w^*||_2 = \sqrt{\sum_{i=0}^{d} 1} = \sqrt{d}$$

Now working under the assumption that each coordinate is sampled independently of the others, we can write

$$\mathrm{E}[(< w^*, z >)^2] = \sum_{i=0}^{d} \mathrm{E}[([w^*]_i z_i)^2] = \sum_{i=0}^{d} ([w^*]_i)\mathrm{E}[z_i^2]$$

Now using $w_i^* = 1$, $\forall i \in [d]$,

$$\sum_{i=0}^{d} |[w^*]_i|^2 = d$$

and

$$\mathrm{E}[z_i^2] = \Theta(\frac{1}{d})$$

$$\sum_{i=0}^{d} ([w^*]_i)\mathrm{E}[z_i^2] = d\Theta(\frac{1}{d}) = \Theta(1)$$

So overall,

$$\mathrm{E}[(< w^*, z >)^2] = \Theta(1)$$

so

$$(< w^*, z >)^2 = \Theta(1)$$

$$| < w^*, z > | = \Theta(1)$$

which further imples,

$$||w^*|| = \sqrt{d}$$

This means that if we perturb the clean x by some $= \theta(\frac{1}{\sqrt{d}})$ in the direction $-yMw^*$ the predicted label can change. If y was the original label of the clean data, $y = sign(< Mw^*, x >)$, the new label after perturbation,

$$y' = sign(< Mw^*, x + dx >) = sign(< Mw^*, x > + < Mw^*, dx >) = y - \Theta(1)$$

$$y' = sign(| < w^*, z > | + < Mw^*, dx >) = \Theta(1) - y$$

So,

$$y' = \Theta(1) - y$$

Given that $y \in -1, 1$, this can in lead to change in the predicted class. Hence the linear classifier $L(x)$ is not robuts to perturbations with l_2 norm bounded by $\tau = \frac{1}{d}$

Answer 4

4.1

$$f(w) = \left\| w^* - \frac{w}{\|w\|_2} \right\|_2^2 + \lambda \|w\|_2^2.$$

Now if $w = \epsilon w^*$,

$$f(w) = \left\| w^* - \frac{\epsilon w^*}{\epsilon \|w^*\|_2} \right\|_2^2 + \lambda \epsilon^2 \|w^*\|_2^2$$

Given $\|w^*\|_2 = 1$

$$f(w) = \lambda \epsilon^2$$

Now as $\epsilon \to O^+$, $f(w) \to 0^+$

Now for the infimum: Both the terms of $f(w)$ are bounded as:

$$\left\| w^* - \frac{w}{\|w\|_2} \right\|_2^2 \geq 0$$

$$\lambda \|w\|_2^2 \geq 0$$

This means,

$$f(w) = \left\| w^* - \frac{w}{\|w\|_2} \right\|_2^2 + \lambda \|w\|_2^2 \geq 0$$

Now if $w = \epsilon w^*$, and as $\epsilon \to 0+$, $f(w) \to 0+$, i.e. the lower bound of 0 is achievable in limit, making it the maximum lower bound and hence the infimum of $f(w)$.
This means $f^* = 0$, and hence $f(w) \to f^*$, as $\epsilon \to 0+$

4.2

Let $w_1 = -\epsilon w^*$, and $w_2 = \epsilon w^*$.
This gives:

$$f(w_1) = \left\| w^* + \frac{\epsilon w^*}{\epsilon \|w^*\|_2} \right\|_2^2 + \lambda \epsilon^2 \|w^*\|_2^2 = 4 + \lambda \epsilon^2$$

$$f(w_2) = \left\| w^* - \frac{\epsilon w^*}{\epsilon \|w^*\|_2} \right\|_2^2 + \lambda \epsilon^2 \|w^*\|_2^2 = \lambda \epsilon^2$$

For Lipschitz: For some constant $C \geq 0$

$$|f(w_1) - f(w_2)| \leq C \|w_1 - w_2\|_2$$

$$|4 + \lambda \epsilon^2 - \lambda \epsilon^2| \leq 2 \epsilon C$$

This gives us,

$$\epsilon \geq \frac{2}{C}$$

This means if I select $0 \leq \epsilon < \frac{2}{C}$, the Lipschitz condition will not hold.

L-smoothness Assuming L-smoothness, and using upper quadratic bound we can write:

$$f(w_1) \leq f(w_2) + < \nabla f(w_2), (w_1 - w_2) > + \frac{L||w_1 - w_2||_2^2}{2}$$

Using information given in the FAQ section for this HW on Piazza, for $w = \epsilon w^*$, and $\epsilon > 0$ I can write $\nabla f(w_2) = 2\lambda\epsilon$

$$4 + \lambda\epsilon^2 \leq \lambda\epsilon^2 + 2\lambda\epsilon * (-2\epsilon) + 2L\epsilon^2$$

$$2 \leq -2\lambda\epsilon^2 + L\epsilon^2$$

This gives us,

$$\epsilon^2 \geq \frac{2}{L - 2\lambda}$$

Now similar to the Lipschitz condition argument, I can choose $0 \leq \epsilon < \sqrt{\frac{2}{L-2\lambda}}$, which will violate the L-smoothness condition. Combining the two, I can choose, $0 \leq \epsilon < \min(\frac{2}{C}, \sqrt{\frac{2}{L-2\lambda}})$ which will violate both Lipschitz and L-smoothness properties of the function.

Answer 5

5.1

For finding the min-max optimal solution, I ran a grid search in the domain of x, and y. I first divided the x-domain, $[-1, 1]$ and the y-domain, $[-2\pi, 2\pi]$ into 1000 points each. Then for each x I found the y which maximized $f(x, y)$ and collected all of them. This gave one y corresponding every x. Now I chose the pair with the lowest $f(x, y)$ value. This gave me $(x^*, y^*) = (0, \pi)$.
Proof that this is the global min-max optimal solution:
According to lecture 26, slide 12, at the global min-max optimal solution,

$$f(x^*, y) \leq \max_{y'} f(x, y')$$

For the given $(x^*, y^*) = (0, \pi)$, $f(x^*, y^*) = 1$.
Now for any $x \in [-1, 1]$, choose $y' = sign(x) * *\pi$. This means

$$f(x, y') = 2 * \pi * |x| + 1 \geq 1$$

Now as,

$$\max_{y'} f(x, y') \geq f(x, y') \geq 1$$

This means,

$$f(x^*, y^*) = 1 \leq \max_{y'} f(x, y')$$

This proves that this point is indeed the global min-max optimal solution.

This point is not the second order local min-max optimal solution because,

$$\nabla_x f(0, \pi) = 0.2\pi \neq 0$$

which is one of the requirements for the point to be local min-max optimal solution.

5.2

1. Original Setup:

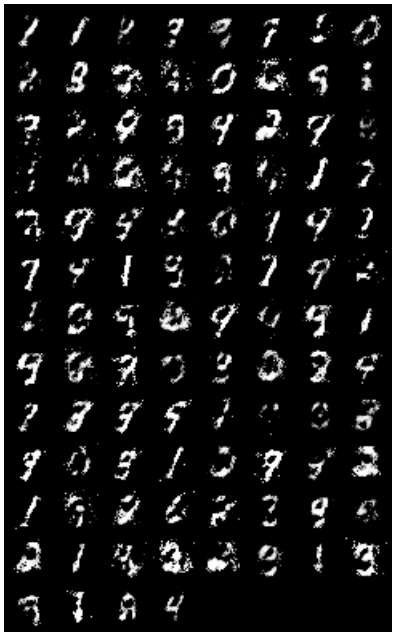

Figure 3: Generated samples after $T = 20$

2. Discriminator too powerful:

Figure 4: Generated samples after $T = 20$

3. Low noise:

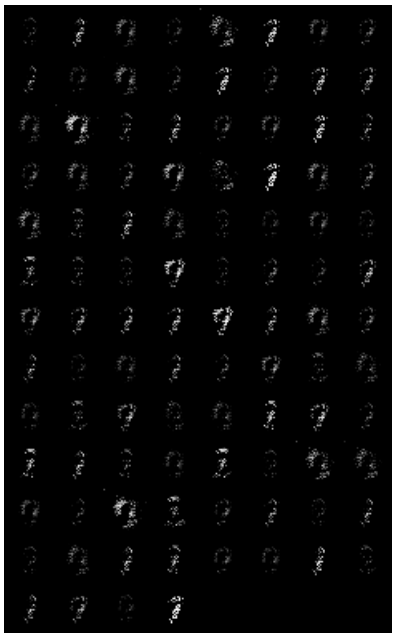

Figure 5: Generated samples after $T = 45$

As the learning rate of the discriminator is increased and that of the generator is decreased, the discriminator learns much more quickly, this means that after a few updates the generator will stop receiving any gradient signals as the discriminator will start to identify fake data easily. This seems to be the case Fig 4, the generator has seemingly learned to generate only some mode of data, in this case something which looks like 1, and then its training effectively stop. For the last part, when a lot of data is used, the noise in the updates become very low, and the discriminator is able to easily learn feature which can help in identifying real images, and hence the generator is not able to learn anything at all. This can be seen in Fig 5, it is generating some random noise for every sample.

```
import numpy as np
import os
import matplotlib.pyplot as plt
import pdb
import matplotlib as mpl

mpl.rcParams['figure.dpi'] = 300

def Relu(x, require_grad=True):
    mask = 1.0*(x>=0)
    val = mask*x
    if(require_grad):
         return val, mask
    else:
        return val

def gen_gt(x):
    '''
    x: input of shape (b,d)
    '''
    return np.sum(Relu(x, require_grad=False),axis=1)

def h(W,x, require_grad=True):
    '''
    W: weight matrix of shape (d,m)
    x: input of shape (b,d)
    '''
    d, m = W.shape
    b = x.shape[0]
    W_T = np.repeat(np.expand_dims(W.T,axis=0), b, axis=0) # b,m,d
    X = np.repeat(np.expand_dims(x,axis=1), m, axis=1) # b,m,d

    if require_grad:
        val, mask = Relu(np.sum(W_T*X, axis=2), require_grad=True)
        MASK = np.repeat(np.expand_dims(mask,2), d, axis=2)
        grad_h_W = MASK*X
        return np.sum(val, axis=1), grad_h_W
    else:
        val= Relu(np.sum(W_T*X, axis=2), require_grad=False)
        return np.sum(val, axis=1)


def SGD_step(W, X, Y, lr):
    '''
    W: (d,m)
    X: (b,d)
    Y: (b,)
    '''
    pred, grad_h_W = h(W, X, require_grad=True)
    diff = Y - pred
    loss_f = np.mean(diff**2)

    grad_W = np.mean(-2*np.expand_dims(diff, axis=(1,2))*grad_h_W, axis=0)
    W = W - lr*grad_W.T

    return W, loss_f

def plot_prop(x_label, datas, labels, prop_name, figname, X=None):
    fig = plt.figure(figsize=(10,7))
    if(X==None):
        for data, label in zip(datas, labels):
            plt.plot(data, label=label)
    else:
        for data, label in zip(datas, labels):
            plt.plot(X, data, label=label)

    plt.xlabel(x_label)
    plt.ylabel(prop_name)
    plt.legend()
```

```
    plt.savefig("{}.png".format(figname))
    plt.close()

def train(d, m, b, lr, X_test=None, Y_test=None, eps=1e-3):
    print("d: {}, m: {}, b: {}".format(d,m,b))

    mean_w = np.zeros(shape=(d,))
    std_w = (1.0/d)*np.ones(shape=(d,))

    W = np.array([np.random.normal(mean_w, std_w) for i in range(m)]).T

    num_iters = 200
    f_arr = []
    f_old = 99999999
    count = 0
    for iter in range(num_iters):
        X = np.array([np.random.normal(np.zeros(shape=(d,)), np.ones(shape=(d,))) for
j in range(b)])
        Y = gen_gt(X)
        W, f = SGD_step(W, X, Y, lr)
        print("iter: [{}/{}], f: {}".format(iter, num_iters, f))
        pred_test = h(W, X_test, require_grad=False)
        err_test = Y_test - pred_test
        f_test = np.mean(err_test**2)
        f_arr.append(f_test*1.0)
        if(abs(f-f_old)<eps):
            break
        else:
            f_old = f*1.0
            count += 1

    return np.array(f_arr), count

def train_epochs(X, Y, d, m, b, lr, eps=1e-3):
    print("d: {}, m: {}, b: {}".format(d,m,b))

    mean_w = np.zeros(shape=(d,))
    std_w = (1.0/d)*np.ones(shape=(d,))

    W = np.array([np.random.normal(mean_w, std_w) for i in range(m)]).T

    num_iters = 200
    f_arr = []
    f_old = 99999999
    count = 0

    ids = np.arange(X.shape[0])
    for epoch in range(num_epochs):
        np.random.shuffle(ids)
        f_epoch = 0.0
        for i in range(0, X.shape[0], b):
            batch_ids = ids[i:np.amin(i+b, X.shape[0])]
            X_batch = X[batch_ids]
            Y_batch = Y[batch_ids]
            W, f = SGD_step(W, X_batch, Y_batch, lr)
            print("iter: [{}/{}], f: {}".format(iter, num_iters, f))
            f_epoch += f*1.0
            if(abs(f-f_old)<eps):
                break
            else:
                f_old = f*1.0
                count += 1
        f_arr.append(f_epoch*1.0)

    return np.array(f_arr), count

def main():
    d = 20
    m_start = 20
```

```
    m_end = 201
    m_step = 10
    b = 32
    lr = 1e-3
    datas = []
    labels = []
    counts = []
    f_min_arr = []
    Ms = []
    save_path = os.path.join(os.getcwd(), "d_{}_b_{}_lr_{}".format(d,b,lr))
    if not os.path.exists(save_path):
        os.mkdir(save_path)
    mean_x = np.zeros((d,))
    std_x = np.ones((d,))

    num_test_samples = 1000
    X_test = np.array([np.random.normal(mean_x, std_x) for i in range(num_test_samples
)])
    Y_test = gen_gt(X_test)

    for m in range(m_start,m_end, m_step):
        f_arr, count = train(d, m, b, lr, X_test, Y_test)
        f_min_arr.append(np.amin(f_arr))
        datas.append(f_arr)
        labels.append("m = {}".format(m))
        Ms.append(m)
        counts.append(count)
    plot_prop("num_iters", [datas[0], datas[-1]], ["proper", "over"], "f", os.path.joi
n(save_path,"part_1"))
    plot_prop("num_iters", datas, labels, "f", os.path.join(save_path, "bonus_f"))
    plot_prop("m", [counts], ["num_iters"], "num_iters", os.path.join(save_path, "bonu
s_iters"), Ms)
    plot_prop("m", [f_min_arr], ["f_min"], "f_min", os.path.join(save_path, "bonus_f_m
in"), Ms)
    # plot_prop(f_arr, "f", figname="q1_d_{}_m_{}_b_{}".format(d,m,b))
main()
```

```
import numpy as np
import matplotlib as mpl
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def f(x,y):
    return 0.2*x*y - np.cos(y)

def grad_f(x,y):
    grad_x = 0.2*y
    grad_y = 0.2*x + np.sin(y)

    return grad_x, grad_y

num_points = 1000
X = np.arange(-1,1,2/num_points)
Y = np.arange(-2*np.pi, 2*np.pi, 4.*np.pi/num_points)

Y_maxs = []
Z_Y_maxs = []

for x in X:
    Zs = f(x*np.ones_like(Y), Y)
    z_max_id = np.argmax(Zs)
    Y_maxs.append(Y[z_max_id]*1.0)
    Z_Y_maxs.append(Zs[z_max_id]*1.0)

X_star = X[np.argmin(Z_Y_maxs)]
Y_star = Y_maxs[np.argmin(Z_Y_maxs)]

Z_star = f(X_star, Y_star)



print("(x*,y*,Z*) ",(X_star, Y_star, Z_star))



grad_x, grad_y = grad_f(X_star, Y_star)
print("(grad_x*,grad_y*) ",(grad_x, grad_y))
```