Homework #2 - B

Spring 2020, CSE 446/546: Machine Learning
Richy Yun
Due: 5/12/20 11:59 PM

Convexity and Norms

B.1 *[6 points]* For this problem we simply need to show $||x||_n \leq ||x||_{n-1}$:

$$||x||_n \leq ||x||_{n-1}$$
$$\left(\sum_{i=1}^{n} |x_i|^n\right)^{1/n} \leq \left(\sum_{i=1}^{n} |x_i|^{n-1}\right)^{1/n-1}$$
$$\sum_{i=1}^{n} |x_i|^n \leq \left(\sum_{i=1}^{n} |x_i|^{n-1}\right)^{n/n-1}$$
$$\sum_{i=1}^{n} y_i^m \leq \left(\sum_{i=1}^{n} y_i\right)^m$$

where $y_i = |x_i|^{n-1}$ and $m = \frac{n}{n-1}$. Thus the problem is showing that the sum of an mth power is less than or equal to the mth power of the sum:

$$\sum_{i=1}^{n} y_i^m \leq \left(\sum_{i=1}^{n} y_i\right)^m$$

If $\sum_{i=1}^{n} y_i = Y$ then by definition $\frac{y_i}{Y} \in [0,1]$ and $\left(\frac{y_i}{Y}\right)^m \leq \frac{y_i}{Y}$

$$\sum_{i=1}^{n} y_i^m = Y^m \sum_{i=1}^{n} \left(\frac{y_i}{Y}\right)^m \leq Y^m \sum_{i=1}^{n} \frac{y_i}{Y} = Y^m = \left(\sum_{i=1}^{n} y_i\right)^m$$

Therefore, we know $||x||_n \leq ||x||_{n-1}$ and it follows that $||x||_\infty \leq ||x||_2 \leq ||x||_1$.

B.2

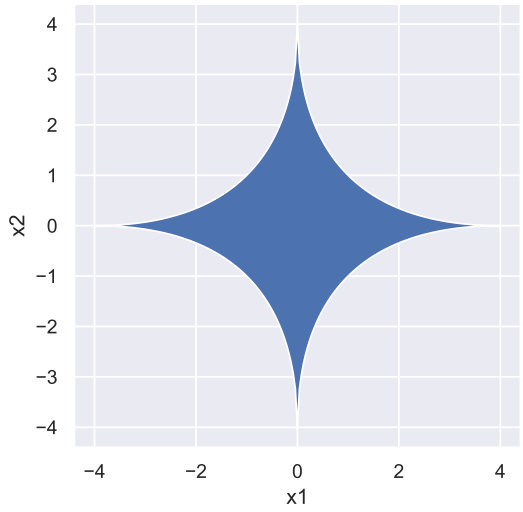
a. *[3 points]* As mentioned in lecture, we need to show $f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y)$ to prove convexity. Since we know a norm follows absolute scalability and triangle inequality, the condition holds true and all norm functions are convex.

b. *[3 points]* $\{x \in \mathbb{R}^n : ||x|| \leq 1\}$ is essentially a ball around the origin which is intuitively convex. To prove we can show:

$$||\lambda x + (1-\lambda)y|| \leq \lambda||x|| + (1-\lambda)||y|| \leq \lambda + (1-\lambda) = 1$$

The first inequality is due to the triangle inequality of norm and the second due to the condition of the set. The inequality holds and all values are within the limits of the set and so the set is convex.

c. *[2 points]* The defined set is not convex as a line between two points can lie outside of the set. (note: the 'tips' of the plot go all the way to 4, but the values are too small to be accurately represented)



Source Code

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

sns.set()
x = np.linspace(0, 4, 100)
y = (2 - np.sqrt(x))**2
x = np.sort(np.append(x, -x[1:len(x)]))
y = np.append(np.flip(y[1:len(y)]), y)

plt.fill_between(x, y, -y)
plt.xlabel('x1')
plt.ylabel('x2')
plt.show()
```

B.3

a. *[3 points]* If f, g are convex functions, then their sum is also convex:

$$
\begin{aligned}
f(\lambda x + (1-\lambda)y) &\leq \lambda f(x) + (1-\lambda) f(y) \\
g(\lambda x + (1-\lambda)y) &\leq \lambda g(x) + (1-\lambda) g(y) \\
f(\lambda x + (1-\lambda)y) + g(\lambda x + (1-\lambda)y) &\leq \lambda f(x) + (1-\lambda) f(y) + \lambda g(x) + (1-\lambda) g(y) \\
f(\lambda x + (1-\lambda)y) + g(\lambda x + (1-\lambda)y) &\leq \lambda (f(x) + g(x)) + (1-\lambda)(f(y) + g(y)) \\
h(x) &= f(x) + g(x) \\
h(\lambda x + (1-\lambda)y) &\leq \lambda h(x) + (1-\lambda) h(y)
\end{aligned}
$$

Since $\ell_i(w)$ is given to be convex over $w \in \mathbb{R}^d$ and a norm is always convex, their sum is convex over $w \in \mathbb{R}^d$ as well.

b. *[1 points]* If the function is convex the local minima is the global minima which allows us to find the absolute minimum loss.

Logistic Regression

B.4

a. *[5 points]* The partial derivative can be computed for a specific w then extrapolated to matrix form for W. If we set $\hat{\mathbf{y}}_i^{(\mathbf{w}^{(m)})} = \texttt{softmax}(\mathbf{w}^{(m)} \cdot \mathbf{x}_i)$:

$$\begin{aligned}\nabla_{\mathbf{w}^{(m)}} \mathcal{L} &= \frac{\partial}{\partial \mathbf{w}^{(m)}} - \sum_{i=1}^{n} \sum_{\ell=i}^{k} \mathbf{1}\{y_i = \ell\} \log(\hat{\mathbf{y}}_i^{(\mathbf{w}^{(\ell)})}) \\ &= -\sum_{i=1}^{n} \sum_{\ell=i}^{k} \mathbf{1}\{y_i = \ell\} \frac{1}{\hat{\mathbf{y}}_i^{(\mathbf{w}^{(\ell)})}} \frac{\partial \hat{\mathbf{y}}_i^{(\mathbf{w}^{(\ell)})}}{\partial \mathbf{w}^{(m)}}\end{aligned}$$

To solve for the partial derivative of the `softmax` function, in the first case if $m = \ell$ we have:

$$\begin{aligned}\frac{\partial \hat{\mathbf{y}}_i^{(\mathbf{w}^{(\ell)})}}{\partial \mathbf{w}^{(m)}} &= \frac{\mathbf{x}_i \exp(\mathbf{w}^{(\ell)} \mathbf{x}_i) \sum_{j=1}^{k} \exp(\mathbf{w}^{(j)} \mathbf{x}_i) - \mathbf{x}_i \exp(\mathbf{w}^{(m)} \mathbf{x}_i) \exp(\mathbf{w}^{(\ell)} \mathbf{x}_i)}{(\sum_{j=1}^{k} \exp(\mathbf{w}^{(j)} \mathbf{x}_i))^2} \\ &= \mathbf{x}_i \frac{\exp(\mathbf{w}^{(\ell)} \mathbf{x}_i)}{\sum_{j=1}^{k} \exp(\mathbf{w}^{(j)} \mathbf{x}_i)} \frac{\sum_{j=1}^{k} \exp(\mathbf{w}^{(j)} \mathbf{x}_i) - x_i \exp(\mathbf{w}^{(m)} \mathbf{x}_i)}{\sum_{j=1}^{k} \exp(\mathbf{w}^{(j)} \mathbf{x}_i)} \\ &= \mathbf{x}_i \hat{\mathbf{y}}_i^{(\mathbf{w}^{(\ell)})} (1 - \hat{\mathbf{y}}_i^{(\mathbf{w}^{(m)})})\end{aligned}$$

In the second case when $m \neq \ell$ we have:

$$\begin{aligned}\frac{\partial \hat{\mathbf{y}}_i^{(\mathbf{w}^{(\ell)})}}{\partial \mathbf{w}^{(m)}} &= \frac{0 - \mathbf{x}_i \exp(\mathbf{w}^{(m)} \mathbf{x}_i) \exp(\mathbf{w}^{(\ell)} \mathbf{x}_i)}{(\sum_{j=1}^{k} \exp(\mathbf{w}^{(j)} \mathbf{x}_i))^2} \\ &= \mathbf{x}_i (-\hat{\mathbf{y}}_i^{(\mathbf{w}^{(m)})})(\hat{\mathbf{y}}_i^{(\mathbf{w}^{(\ell)})})\end{aligned}$$

Combining them, we have:

$$\begin{aligned}\nabla_{\mathbf{w}^{(m)}} \mathcal{L} &= -\sum_{i=1}^{n} \mathbf{x}_i \left(\mathbf{1}\{y_i = m\}(1 - \hat{\mathbf{y}}_i^{(\mathbf{w}^{(m)})}) - \sum_{\ell \neq m}^{k} \mathbf{1}\{y_i = \ell\} \hat{\mathbf{y}}_i^{(\mathbf{w}^{(m)})} \right) \\ &= -\sum_{i=1}^{n} \mathbf{x}_i \left(\mathbf{1}\{y_i = m\} - \hat{\mathbf{y}}_i^{(\mathbf{w}^{(m)})} \sum_{\ell=1}^{k} \mathbf{1}\{y_i = \ell\} \right) \\ &= -\sum_{i=1}^{n} \mathbf{x}_i \left(\mathbf{1}\{y_i = m\} - \hat{\mathbf{y}}_i^{(\mathbf{w}^{(m)})} \right)\end{aligned}$$

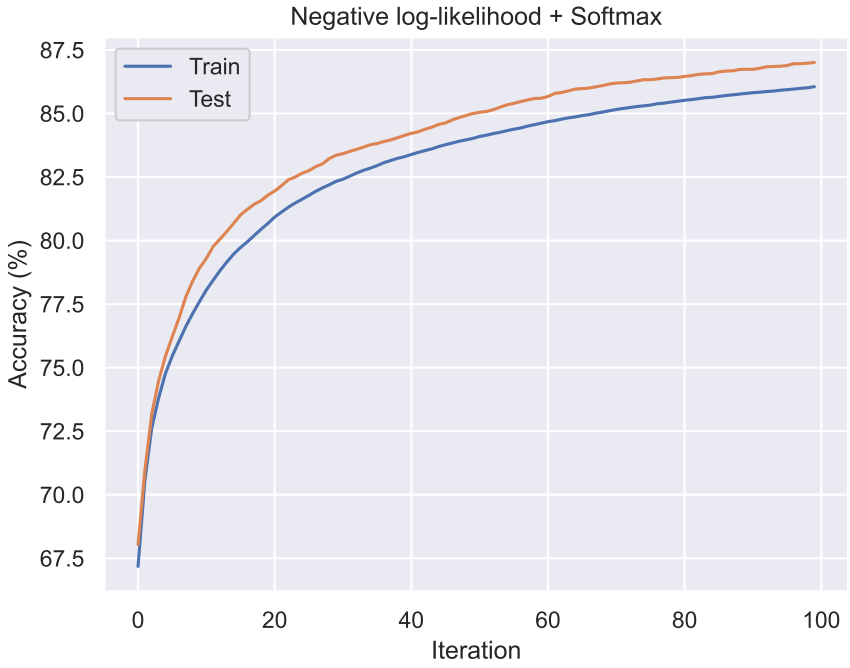
Expanding to matrix format for W we finally have:

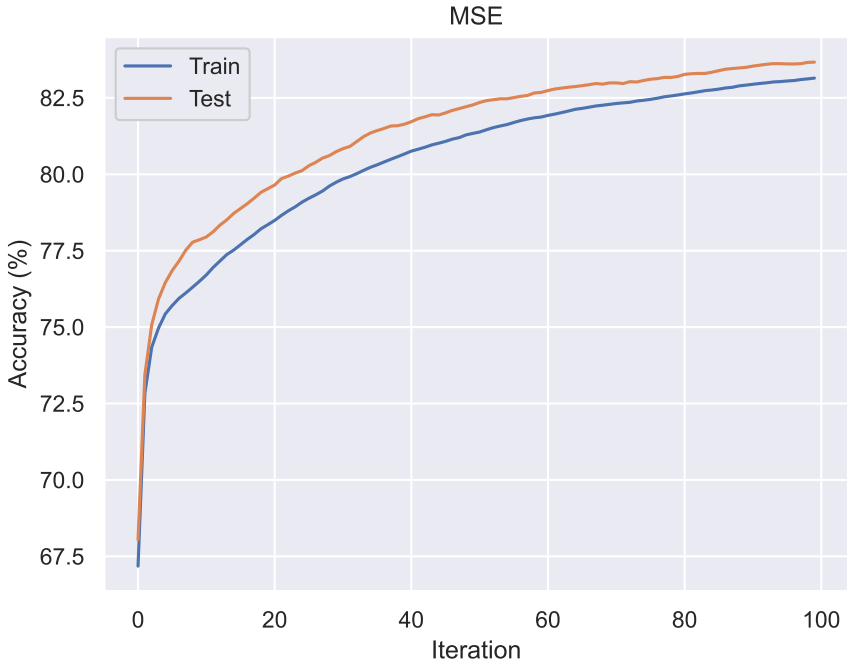
$$\nabla_W \mathcal{L} = -\sum_{i=1}^{n} \mathbf{x}_i (\mathbf{y}_i - \hat{\mathbf{y}}_i^W)^\top$$

b. *[5 points]* Since $\frac{d}{dx} ||x||_2^2 = 2x$ we have:

$$\begin{aligned}\nabla_W J(W) &= \frac{\partial}{\partial W} \frac{1}{2} \sum_{i=1}^{n} ||\mathbf{y}_i - W^\top \mathbf{x}_i||_2^2 \\ &= -\sum_{i=1}^{n} \mathbf{x}_i (\mathbf{y}_i - W^\top \mathbf{x}_i)^\top \\ &= -\sum_{i=1}^{n} \mathbf{x}_i (\mathbf{y}_i - \widetilde{\mathbf{y}}_i^{(W)})^\top\end{aligned}$$

c. *[15 points]* Using a step size of 0.1, I noticed both methods converged at around 100 iterations and so did 100 to more easily compare them. Both methods work fairly well, with NLL+softmax reaching higher accuracy compared to MSE. MSE performs better at much earlier iterations but plateaus faster.





Source Code

```
import numpy as np
import matplotlib.pyplot as plt
```

```
from mnist import MNIST
import torch
import seaborn as sns

sns.set()
# Load data
mndata = MNIST('../MNIST/data/')
X_train, labels_train = map(np.array, mndata.load_training())
X_test, labels_test = map(np.array, mndata.load_testing())

# Rearrange data
X_train = X_train/255.0
X_train = torch.from_numpy(X_train)
X_train = X_train.type(torch.FloatTensor)
X_test = X_test/255.0
X_test = torch.from_numpy(X_test)
X_test = X_test.type(torch.FloatTensor)

# for NLLLoss+softmax
Y_train = torch.from_numpy(labels_train)
Y_train = Y_train.type(torch.long)
Y_test = torch.from_numpy(labels_test)
Y_test = Y_test.type(torch.long)

## for MSE
# Y_train = np.zeros((len(X_train), 10))
# Y_train[range(0, len(X_train)), labels_train] = 1
# Y_train = torch.from_numpy(Y_train)
# Y_train = Y_train.type(torch.FloatTensor)
# Y_test = np.zeros((len(X_test), 10))
# Y_test[range(0, len(X_test)), labels_test] = 1
# Y_test = torch.from_numpy(Y_test)
# Y_test = Y_test.type(torch.FloatTensor)

W = torch.zeros(784, 10, requires_grad=True)
step_size = 0.1
max_step = 1
trainacc = []
testacc = []
for i in range(100):
    y_hat = torch.matmul(X_train, W)
    # cross entropy combines softmax calculation with NLLLoss

    loss = torch.nn.functional.cross_entropy(y_hat, Y_train) # for NLLLoss+softmax
    #loss = torch.nn.functional.mse_loss(y_hat, Y_train) # for MSE
    # computes derivatives of the loss with respect to W
    loss.backward()

    # gradient descent update
    W.data = W.data - step_size * W.grad
    max_step = W.grad.max().item()
    print(max_step)

    # .backward() accumulates gradients into W.grad instead
    # of overwriting, so we need to zero out the weights
```

```
        W.grad.zero_()

        # Save accuracy for each iteration
        y = torch.matmul(X_train, W).argmax(axis=1)
        y = sum(y == Y_train) # for NLLLoss
        #y = sum(y == Y_train.argmax(axis=1)) # for MSE
        trainacc.append(y.item()/X_train.size()[0]*100)
        print(y.item()/X_train.size()[0]*100)

        y = torch.matmul(X_test, W).argmax(axis=1)
        y = sum(y == Y_test) # for NLLLoss
        #y = sum(y == Y_test.argmax(axis=1)) # for MSE
        testacc.append(y.item()/X_test.size()[0]*100)

# Plot
plt.plot(trainacc)
plt.plot(testacc)
plt.ylabel('Accuracy (%)')
plt.xlabel('Iteration')
plt.title('Negative log-likelihood + Softmax')
plt.legend(('Train', 'Test'))
```