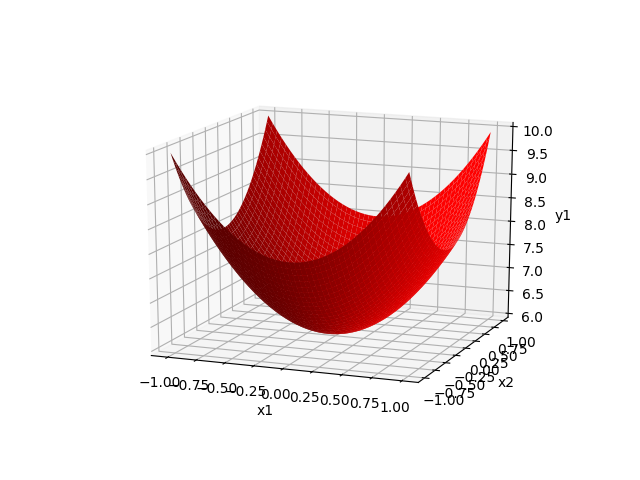
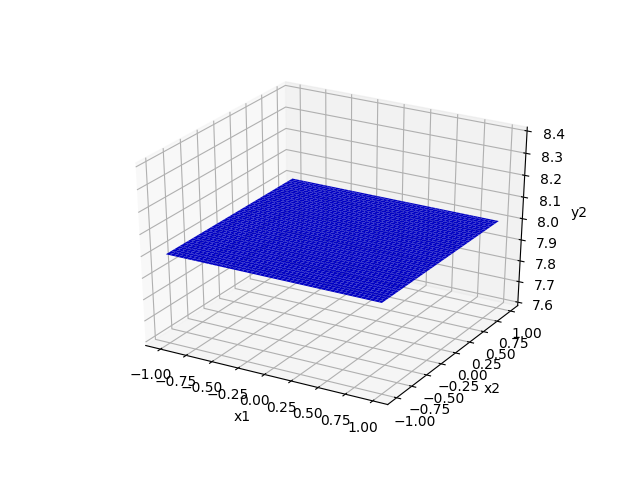
Question 2

1)

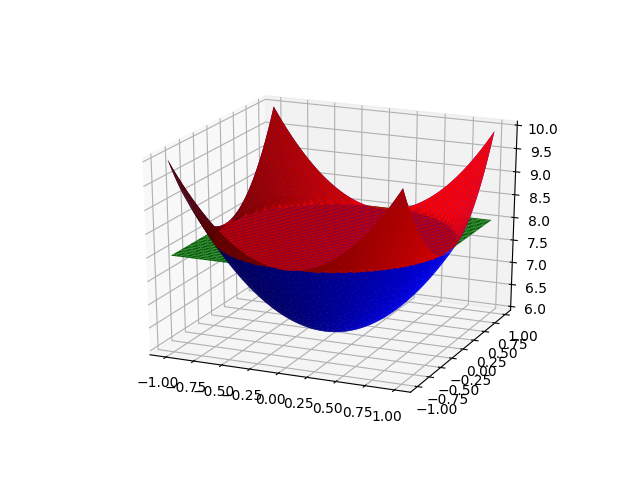


2)

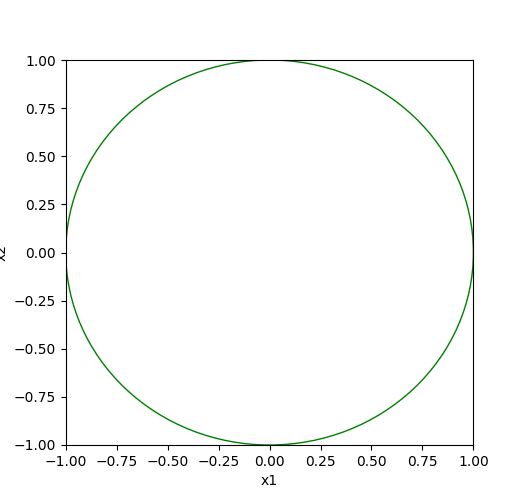


3)

The decision region in 3D:

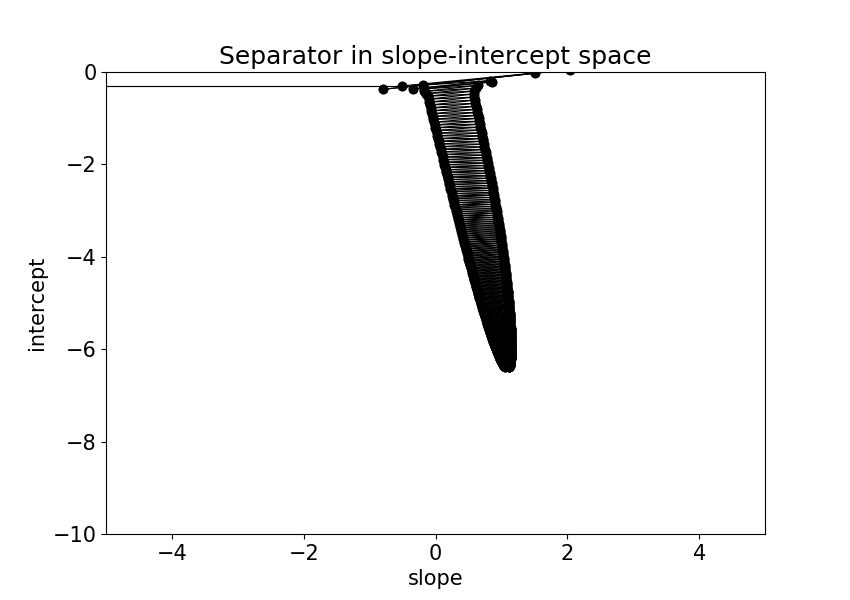


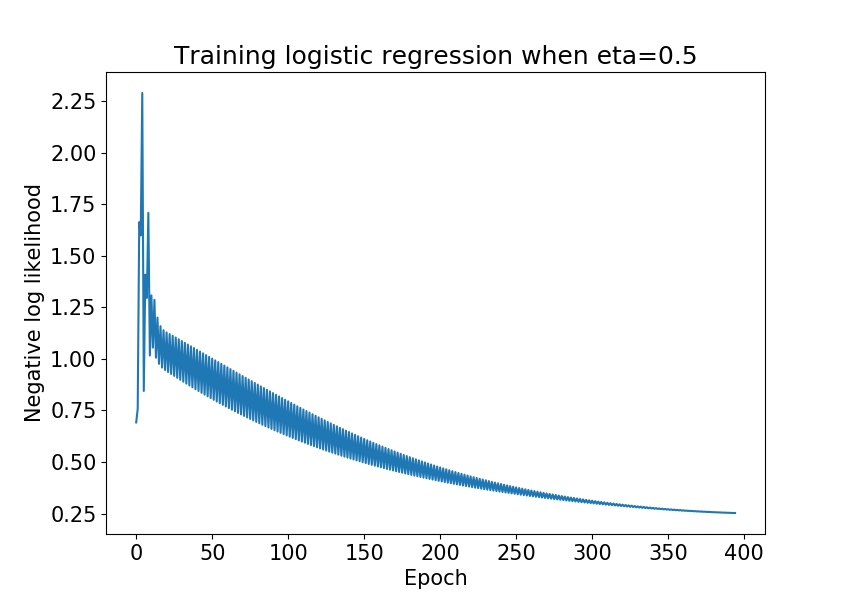
The decision boundary in 2D is a circle with radius=1:



Question 5

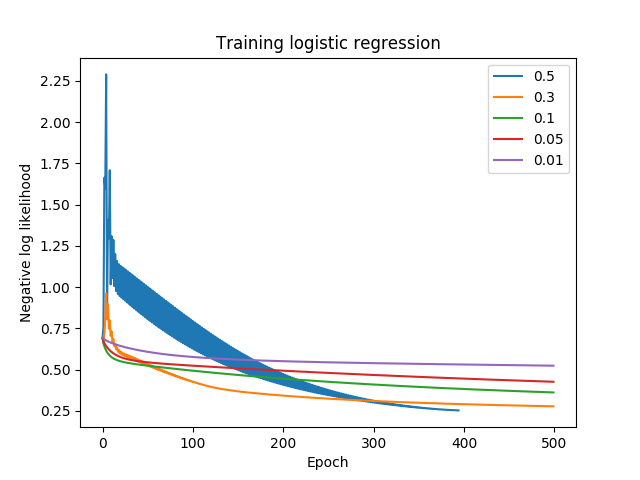
1)





The oscillating of the error curve could be caused by the relatively large learning rate eta=0.5. The error decrease in the direction of the gradient but if the learning rate is too large we could step over the local minimum and jump to some instances with even higher error.

2)



The relatively small learning rate 0.3, 0.1, 0.05, 0.01 gives less oscillating curve since the step we move each time in gradient decent now is smaller.

The relatively large 0.5 learning rate gave us the lowest error overall and it achieves the tolerance level most quickly, in other words before 500 epochs.

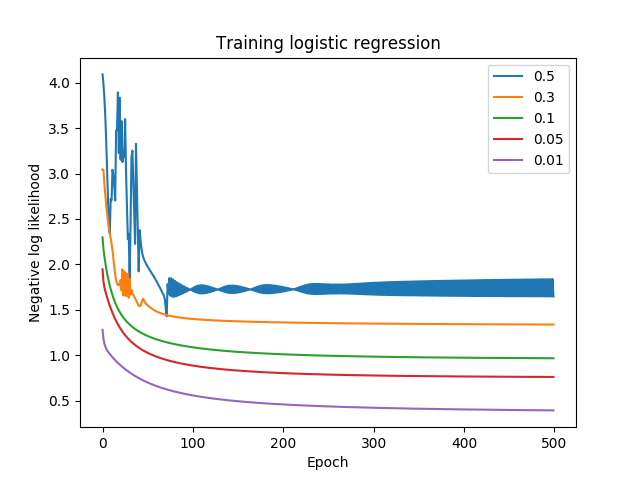
The 0.1 learning rate gave us the low error most quickly as you can see on the graph that green line is the steepest before 100 epochs.

So I think there’s no fixed answer for the learning rate choice in this case. It all depends on the experiment setting.

3)

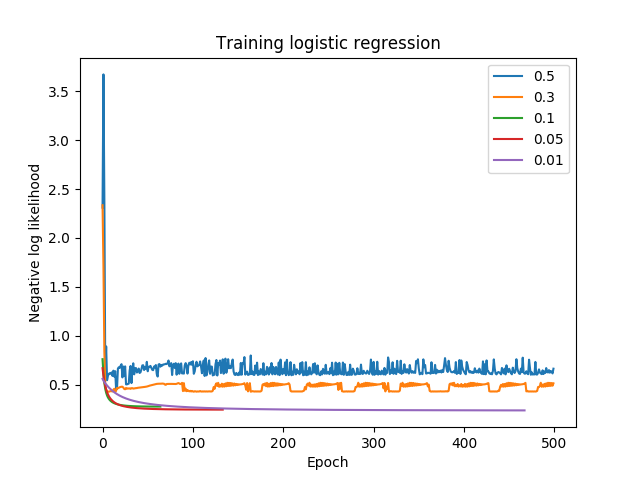
I have talked with TA about this question since I noticed that shuffle made a big difference for the result at different place (shuffle for each epoch/eta or shuffle at the beginning). The TA explained to me that usually shuffle doesn’t affect the model a lot and the most reasonable place to shuffle is for each epoch since it increases the possibility to find the right w the most. But for our case, the dataset given this time is special so that without shuffle the SGD doesn’t make the optimization faster and also with shuffle at different place we will see different effects. The following graphs will illustrate this clearer.

**Without Shuffle:**



The SGD isn’t faster than gradient decent since all of the learning rate didn’t achieve the tolerance level before 500 epochs.

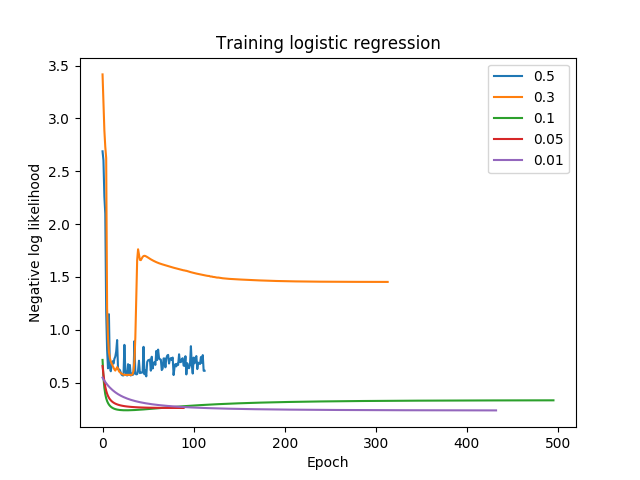
**One instance (Each time the graph is different due to shuffle) with shuffle at the beginning (before the iteration):**



Large learning rate resulted in oscillation and kind-of increasing error curve.

With shuffle at the beginning, the SGD is faster than gradient decent since it generates a low error within 500 epochs.

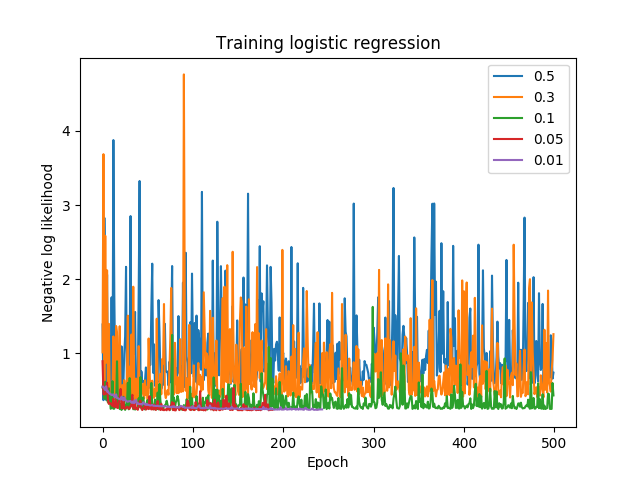
**One instance with shuffle for each eta:**

****

Large learning rate resulted in oscillation and increasing error curve.

With shuffle for each eta, the SGD is faster than gradient decent since it generates a low error within 500 epochs.

**One instance with shuffle for each epoch:**



The SDG is also faster but the oscillation is huge now.

Question 6(Option3):

To run this code on GPU I used the cuda library:

 device = torch.device('cuda:0' if torch.cuda.is\_available() else 'cpu')

and send the corresponding tasks to the gpu memory so that it could be run parallelly in gpu.

model = cifar\_resnet20().to(device)

inputs, labels = data[0].to(device),data[1].to(device)

To apply L2 regularization to the coefficients we can add a weight\_decay argument in optimizer so the code is:

optimizer = optim.SGD(list(model.fc.parameters()), lr=0.001, momentum=0.9, weight\_decay=0.2)

The weight\_decay arugment here decided the value of regularizer.

And in the code, I also added a tester for testing data and print a corresponding accuracy on 10000 test images.

Without Regularization:

Accuracy of the network on the 10000 test images: 67%

With Regularization = 0.1:

Accuracy of the network on the 10000 test images: 65%

With Regularization = 0.2:

Accuracy of the network on the 10000 test images: 64%

We can see that for our model as we increase the regularizer(), the model accuracy is decreasing. The regularization is actually decreasing the model’s prediction power. In assignment 1 question 3.3 we have discussed a similar scenario. In most cases the testing error for regularized regression is lower than unregularized regression if we have a high degree/deep network which is highly likely to cause overfitting.

But the lower error is not guaranteed, if we have a weak/underfitting/luckily proper model then the regularization will slash the predictive power even more and make the testing error larger.

**Question2**

#!/usr/bin/env python  
  
import numpy as np  
import matplotlib.pyplot as plt  
from mpl\_toolkits import mplot3d  
  
  
def y1(x1, x2):  
 return 6 + 2 \* x1 \*\* 2 + 2 \* x2 \*\* 2  
  
  
def y2(x1, x2):  
 return 8 + 0 \* x1 \*\* 2 + 0 \* x2 \*\* 2  
  
# Draw y1(x)  
fig = plt.figure(1)  
ax = plt.axes(projection='3d')  
x1 = x2 = np.arange(-1.0, 1.0, 0.01)  
(x1,x2)=np.meshgrid(x1,x2)  
y1 = y1(x1,x2)  
ax.plot\_surface(x1, x2, y1, color='red')  
ax.set\_xlabel('x1')  
ax.set\_ylabel('x2')  
ax.set\_zlabel('y1')  
plt.show()  
  
  
# Draw y2(x)  
fig = plt.figure(2)  
ax = plt.axes(projection='3d')  
x1 = x2 = np.arange(-1.0, 1.0, 0.01)  
(x1,x2)=np.meshgrid(x1,x2)  
y2 = y2(x1,x2)  
ax.plot\_surface(x1, x2, y2, color='blue')  
ax.set\_xlabel('x1')  
ax.set\_ylabel('x2')  
ax.set\_zlabel('y2')  
plt.show()  
  
# Draw decision boundary  
fig=plt.figure(3)  
plt.axis([-1,1,-1,1])  
ax=fig.add\_subplot(1,1,1)  
circ=plt.Circle((0,0), radius=1, color='g', fill=False)  
ax.add\_patch(circ)  
ax.set\_xlabel('x1')  
ax.set\_ylabel('x2')  
plt.show()  
  
# Draw decision boundary  
fig = plt.figure(4)  
ax = plt.axes(projection='3d')  
x1 = x2 = np.arange(-1.0, 1.0, 0.01)  
(x1,x2)=np.meshgrid(x1,x2)  
y2 = y2(x1,x2)  
y1 = y1(x1,x2)  
y3 = np.maximum(y1,y2)  
ax.plot\_surface(x1, x2, y1, color='blue')  
ax.plot\_surface(x1, x2, y2, color='green')  
ax.plot\_surface(x1, x2, y3, color='red')  
plt.show()

**Logistic\_regression\_mod.py**

#!/usr/bin/env python  
  
import numpy as np  
import scipy.special as sps  
import matplotlib.pyplot as plt  
import assignment2 as a2  
  
# Maximum number of iterations. Continue until this limit, or when error change is below tol.  
max\_iter = 500  
tol = 0.00001  
  
# Step size for gradient descent.  
etas = [0.5, 0.3, 0.1, 0.05, 0.01]  
  
# Load data.  
data = np.genfromtxt('data.txt')  
  
# Data matrix, with column of ones at end.  
X = data[:, 0:3]  
  
# Target values, 0 for class 1, 1 for class 2.  
t = data[:, 3]  
  
# For plotting data  
class1 = np.where(t == 0)  
X1 = X[class1]  
class2 = np.where(t == 1)  
X2 = X[class2]  
  
  
# Initialize legend  
legend = []  
  
for eta in etas:  
 # We need to put initialization of w in the for loop o.w the w is getting smaller and smaller  
 w = np.array([0.1, 0, 0])  
 e\_all = []  
 legend.append(str(eta))  
 for iter in range(0, max\_iter):  
  
 # Compute output using current w on all data X.  
 y = sps.expit(np.dot(X, w))  
  
 # e is the error, negative log-likelihood (Eqn 4.90)  
 e = -np.mean(np.multiply(t, np.log(y)) + np.multiply((1 - t), np.log(1 - y)))  
  
 # Add this error to the end of error vector.  
 e\_all.append(e)  
  
 # Gradient of the error, using Eqn 4.91  
 grad\_e = np.mean(np.multiply((y - t), X.T), axis=1)  
  
 # Update w, \*subtracting\* a step in the error derivative since we're minimizing  
 w\_old = w  
 w = w - eta \* grad\_e  
  
 # Print some information.  
 print('epoch {0:d}, negative log-likelihood {1:.4f}, w={2}'.format(iter, e, w.T))  
  
 # Stop iterating if error doesn't change more than tol.  
 if iter > 0:  
 if np.absolute(e - e\_all[iter - 1]) < tol:  
 break  
  
 plt.plot(e\_all)  
  
# Plot error over iterations  
plt.ylabel('Negative log likelihood')  
plt.title('Training logistic regression')  
plt.xlabel('Epoch')  
plt.legend(legend)  
plt.show()

**Logistic\_regression\_sgd.py**

#!/usr/bin/env python  
  
import numpy as np  
import scipy.special as sps  
import matplotlib.pyplot as plt  
import assignment2 as a2  
  
tol = 0.00001  
max\_iter = 500  
etas = [0.5, 0.3, 0.1, 0.05, 0.01]  
data = np.genfromtxt('data.txt')  
legend=[]  
np.random.shuffle(data)  
X = data[:, 0:3]  
t = data[:, 3]  
  
for eta in etas:  
 w = np.array([0.1, 0, 0])  
 e\_all = []  
 legend.append(str(eta))  
 for itr in range(0, max\_iter):  
 for i in range(0, len(X)):  
 y = sps.expit(np.dot(X[i], w))  
 grad\_e = np.multiply((y - t[i]), X[i,:].T)  
 w = w - eta \* grad\_e  
  
 y = sps.expit(np.dot(X, w))  
 e = -np.mean(np.multiply(t, np.log(y+1e-5)) + np.multiply((1 - t), np.log(1 - y+1e-5)))  
 e\_all.append(e)  
 if itr > 0:  
 if np.absolute(e\_all[itr] - e\_all[itr - 1]) < tol:  
 break  
 plt.plot(e\_all)  
  
plt.ylabel('Negative log likelihood')  
plt.title('Training logistic regression')  
plt.xlabel('Epoch')  
plt.legend(legend)  
plt.show()

**Question 6**

class cifar\_resnet20(nn.Module):

    def \_\_init\_\_(self):

        super(cifar\_resnet20, self).\_\_init\_\_()

        ResNet20 = CifarResNet(BasicBlock, [3, 3, 3])

        url = 'https://github.com/chenyaofo/CIFAR-pretrained-models/releases/download/resnet/cifar100-resnet20-8412cc70.pth'

        ResNet20.load\_state\_dict(model\_zoo.load\_url(url))

        modules = list(ResNet20.children())[:-1]

        backbone = nn.Sequential(\*modules)

        self.backbone = nn.Sequential(\*modules)

        self.fc = nn.Linear(64, 10)

    def forward(self, x):

        out = self.backbone(x)

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

        return self.fc(out)

if \_\_name\_\_ == '\_\_main\_\_':

    device = torch.device('cuda:0' if torch.cuda.is\_available() else 'cpu')

    model = cifar\_resnet20().to(device)

    transform = transforms.Compose([transforms.ToTensor(),

                                    transforms.Normalize(mean=(0.4914, 0.4822, 0.4465),

                                                         std=(0.2023, 0.1994, 0.2010))])

    trainset = datasets.CIFAR10('./data', download=True, transform=transform)

    trainloader = torch.utils.data.DataLoader(trainset, batch\_size=32,

                                          shuffle=True, num\_workers=2)

    testset = datasets.CIFAR10(root='./data', train=False,

                                       download=True, transform=transform)

    testloader = torch.utils.data.DataLoader(testset, batch\_size=32,

                                         shuffle=False, num\_workers=2)

    classes = ('plane', 'car', 'bird', 'cat',

           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

    criterion = nn.CrossEntropyLoss().to(device)

    optimizer = optim.SGD(list(model.fc.parameters()), lr=0.001, momentum=0.9, weight\_decay=0.1 )

    ## Do the training

    for epoch in range(NUM\_EPOCH):  # loop over the dataset multiple times

        running\_loss = 0.0

        for i, data in enumerate(trainloader, 0):

            # get the inputs

            inputs, labels = data[0].to(device),data[1].to(device)

            # zero the parameter gradients

            optimizer.zero\_grad()

            # forward + backward + optimize

            outputs = model(inputs)

            loss = criterion(outputs, labels)

            loss.backward()

            optimizer.step()

            running\_loss += loss.item()

            if i % 20 == 19:    # print every 20 mini-batches

                print('[%d, %5d] loss: %.3f' %

                    (epoch + 1, i + 1, running\_loss / 20))

                running\_loss = 0.0

    print('Finished Training')

    correct = 0

    total = 0

    with torch.no\_grad():

        for data in testloader:

            images, labels = data[0].to(device),data[1].to(device)

            outputs = model(images)

            \_, predicted = torch.max(outputs.data, 1)

            total += labels.size(0)

            correct += (predicted == labels).sum().item()

    print('Accuracy of the network on the 10000 test images: %d %%' % (

        100 \* correct / total))