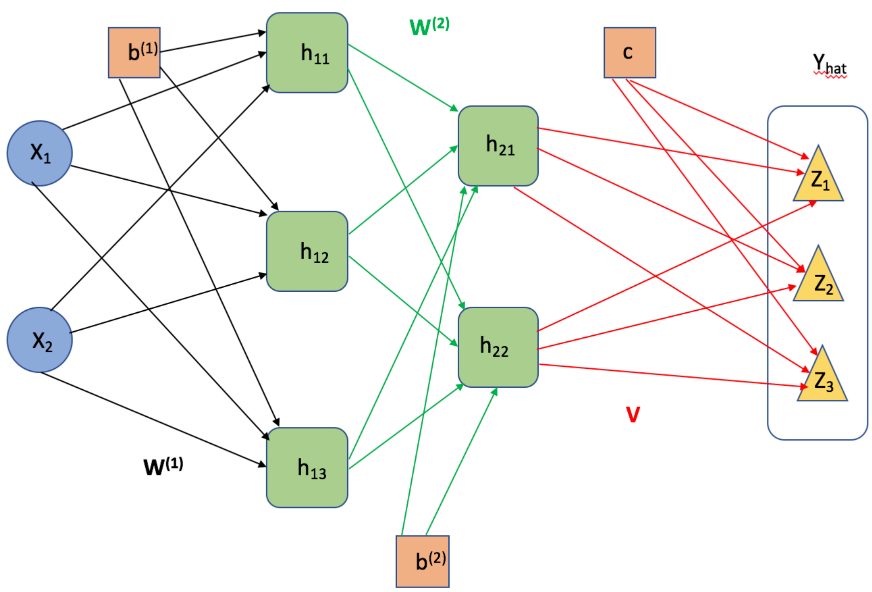
Brody Vogel Homework # 1

October 12, 2018

In [332]:
```
import numpy as np
import matplotlib.pyplot as plt
```



1.1

$h_{11} = \max(0, \mathbf{W^{(1)}}_{\mathbf{11}}X_1 + \mathbf{W^{(1)}}_{\mathbf{21}}X_2 + b^{(1)}{}_1)$
$h_{12} = \max(0, \mathbf{W^{(1)}}_{\mathbf{12}}X_1 + \mathbf{W^{(1)}}_{\mathbf{22}}X_2 + b^{(1)}{}_2)$
$h_{13} = \max(0, \mathbf{W^{(1)}}_{\mathbf{13}}X_1 + \mathbf{W^{(1)}}_{\mathbf{23}}X_2 + b^{(1)}{}_3)$

$h_{21} = \max(0, \mathbf{W^{(2)}}_{\mathbf{11}}h_{11} + \mathbf{W^{(2)}}_{\mathbf{21}}h_{12} + \mathbf{W^{(2)}}_{\mathbf{31}}h_{13} + b^{(2)}{}_1)$
$h_{22} = \max(0, \mathbf{W^{(2)}}_{\mathbf{12}}h_{11} + \mathbf{W^{(2)}}_{\mathbf{22}}h_{12} + \mathbf{W^{(2)}}_{\mathbf{32}}h_{13} + b^{(2)}{}_2)$

$Z_1 = \max(0, \mathbf{V_{11}}h_{21} + \mathbf{V_{21}}h_{22} + c_1)$
$Z_2 = \max(0, \mathbf{V_{12}}h_{21} + \mathbf{V_{22}}h_{22} + c_2)$
$Z_3 = \max(0, \mathbf{V_{13}}h_{21} + \mathbf{V_{23}}h_{22} + c_3)$

$Y_{hat} = [(e^{\mathbf{Z1}}) / (sum_k(e^{Zk}) , (e^{\mathbf{Z2}}) / (sum_k(e^{Zk}) , (e^{\mathbf{Z3}}) / (sum_k(e^{Zk})]$

Or, with matrix algebra:

$\mathbf{h_1} = \max(0, \mathbf{XW^{(1)}} + \mathbf{b^{(1)}})$
$\mathbf{h_2} = \max(0, \mathbf{h_1W^{(2)}} + \mathbf{b^{(2)}})$
$\mathbf{Z} = \max(0, \mathbf{h_2V} + \mathbf{c})$
$\mathbf{Y_{hat}} = (e^{\mathbf{Z}i}) / (sum(e^{Z}))$

1.2

In [333]:
```python
# 1.3

# ReLU activation function
def ReLu(vector):
    return(np.maximum(0, vector))


# softmax output function
def softmax(vector):
    # took some of this from StackOverflow (didn't know about 'np.exp')
    e_x = np.exp(vector)
    return e_x / e_x.sum(axis = 0)

# the neural net
def ff_nn_2_ReLu(input_vector, weights1, weights2, weights3, bias1, bias2, bias3):
    # compute the output step-by-step
        # h1 = W1X + b1^T (3 x 3)
    h1 = ReLu(weights1.dot(input_vector) + bias1)
        # h2 = W2h1 + b2^T (2 x 3)
    h2 = ReLu(weights2.dot(h1) + bias2)
        # Z = Vh2 + c^T (3 x 3)
    z = weights3.dot(h2) + bias3
        # y_hat
    y_hat = softmax(z)

    return(y_hat)
```

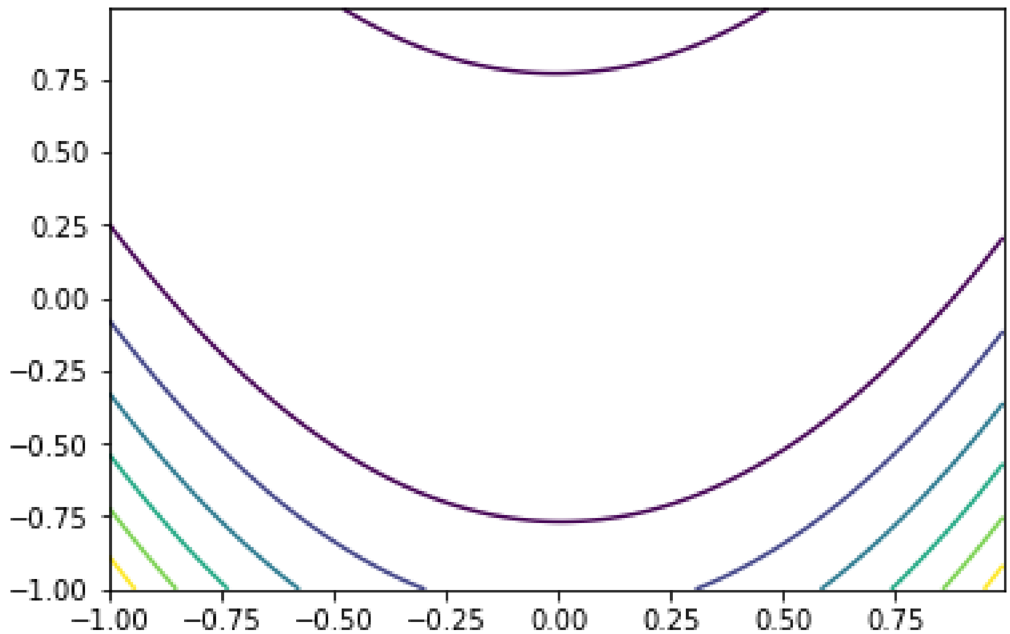
In [334]:
```python
# 1.4
```

```
        # compute the specific output for the supplied input
        X = np.array([[1, 0, 0],
                      [-1, -1, 1]])

        W1 = np.array([[1, 0],
                       [-1, 0],
                       [0, .5]])

        W2 = np.array([[1, 0, 0],
                       [-1, -1, 0]])

        V = np.array([[1, 1],
                      [0, 0],
                      [-1, -1]])

        b1 = np.array([0, 0, 1]).reshape(3 ,1)
        b2 = np.array([1, -1]).reshape(2, 1)
        c = np.array([1, 0, 0]).reshape(3, 1)

        ff_nn_2_ReLu(X, W1, W2, V, b1, b2, c)
```

```
Out[334]: array([[ 0.94649912,  0.84379473,  0.84379473],
                 [ 0.04712342,  0.1141952 ,  0.1141952 ],
                 [ 0.00637746,  0.04201007,  0.04201007]])
```

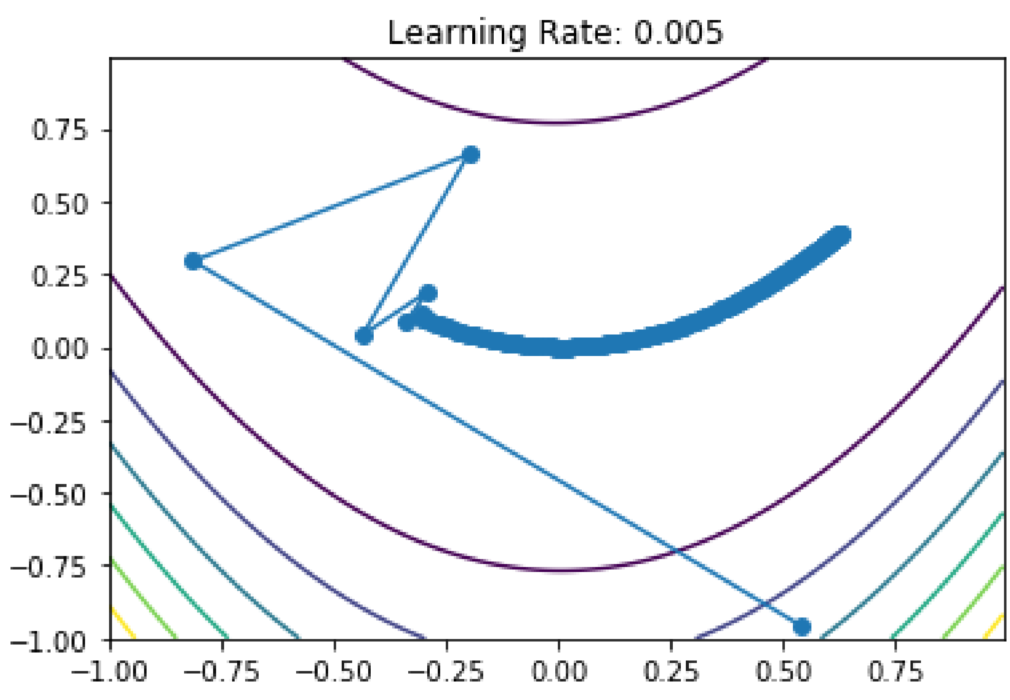
2.1
$\frac{\partial}{\partial x}(1-x)^2 + 100(y-x^2)^2 = 2(1-x)(-1) + 200(y-x^2)(2x) = -2 + 2x - 400(xy - x^3)$
$\frac{\partial}{\partial y}(1-x)^2 + 100(y-x^2)^2 = 200(y-x^2)(1) = 200(y-x^2)$

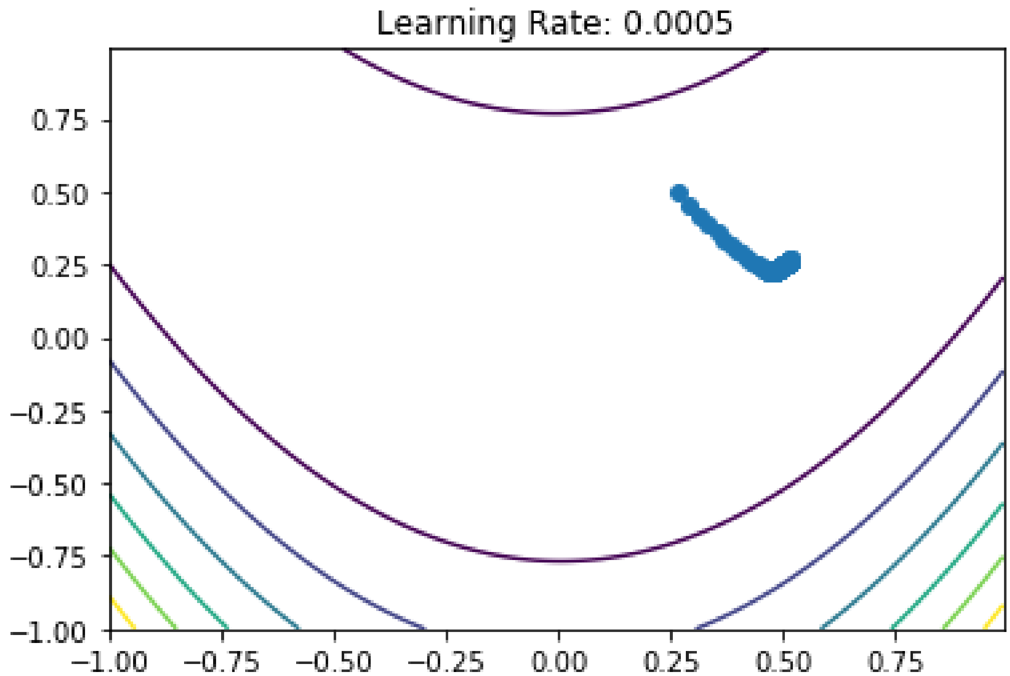
```
In [335]: # 2.2
              # adapted from the Gradient_Descent notebook
          # contours are hard to capture, so we have to zoom way in on delta
          delta = .01
          x = np.arange(-1, 1, delta)
          y = np.arange(-1, 1, delta)
          X, Y = np.meshgrid(x, y)
              # here's the Rosenbrock Function
          Z = (1-X)**2 + 100*(Y-X**2)**2
          fig, ax = plt.subplots()
          CS = ax.contour(X, Y, Z)
```

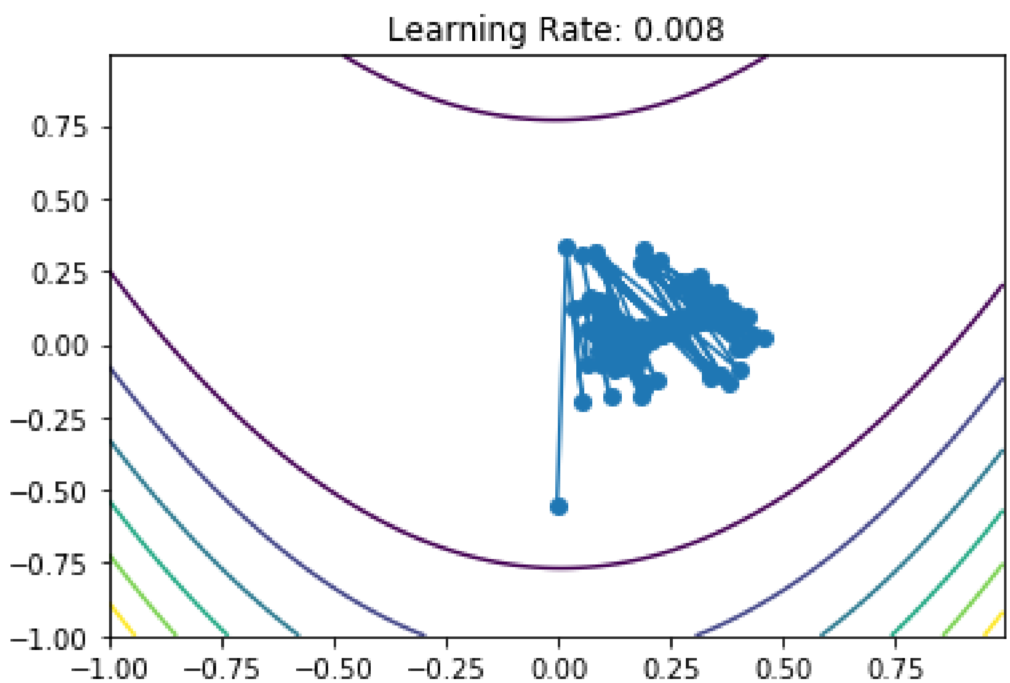


In [336]:
```
# 2.3

# function for computing the instantaneous gradient
# also adapted from Gradient_Descent notebook
def grad_f(vector):
    x, y = vector
    # partial derivative with respect to x
    df_dx = -2 + 2*x - 400*(x*y-x**3)
    # partial derivative with respect to y
    df_dy = 200*(y - x**2)
    # gradient of the two partial derivatives
    return np.array([df_dx, df_dy])

# function for the gradient descent algorithm
# again, adapted from the Gradient_Descent notebook
def grad_descent(starting_point=None, iterations=20, learning_rate=12):
    if starting_point:
        point = starting_point
    else:
        # have to start in a small interval to keep things in check
        point = np.random.uniform(-1,1,size=2)
    trajectory = [point]

    for i in range(iterations):
        grad = grad_f(point)
        point = point - learning_rate * grad
```

```
        trajectory.append(point)
    return np.array(trajectory)

np.random.seed(10)
# learning rate of .005
traj = grad_descent(iterations=200, learning_rate = .005)
traj1 = grad_descent(iterations=200, learning_rate = .0005)
traj2 = grad_descent(iterations=200, learning_rate = .008)

trajs = [traj, traj1, traj2]
lrates = [.005, .0005, .008]

for num in range(3):

    t = trajs[num]
    lr = lrates[num]
    fig, ax = plt.subplots()
    CS = ax.contour(X, Y, Z)
    x= t[:,0]
    y= t[:,1]
    plt.title("Learning Rate: " + str(lr))
    plt.plot(x,y,'-o')

# Each of the learning rates - with 200 iterations - end up in about the
# same place, albeit after much different paths.
# The largest tested learning rate (.008), especially, showed erratic
# behavior, which makes me think it overshoots the minimum in the beginning.
```



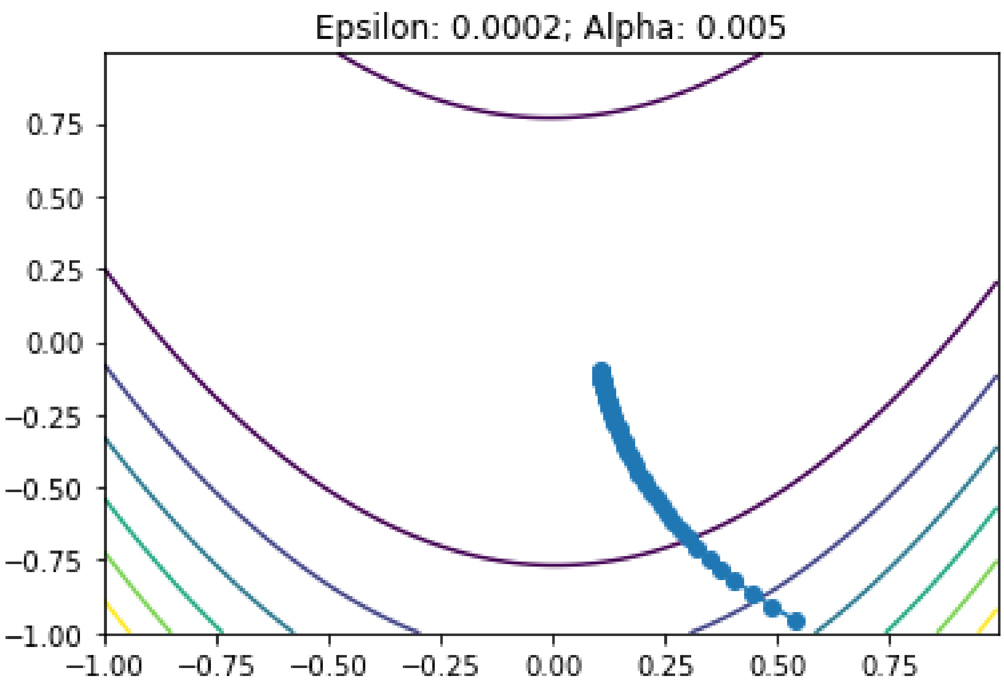

Learning Rate: 0.0005
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75

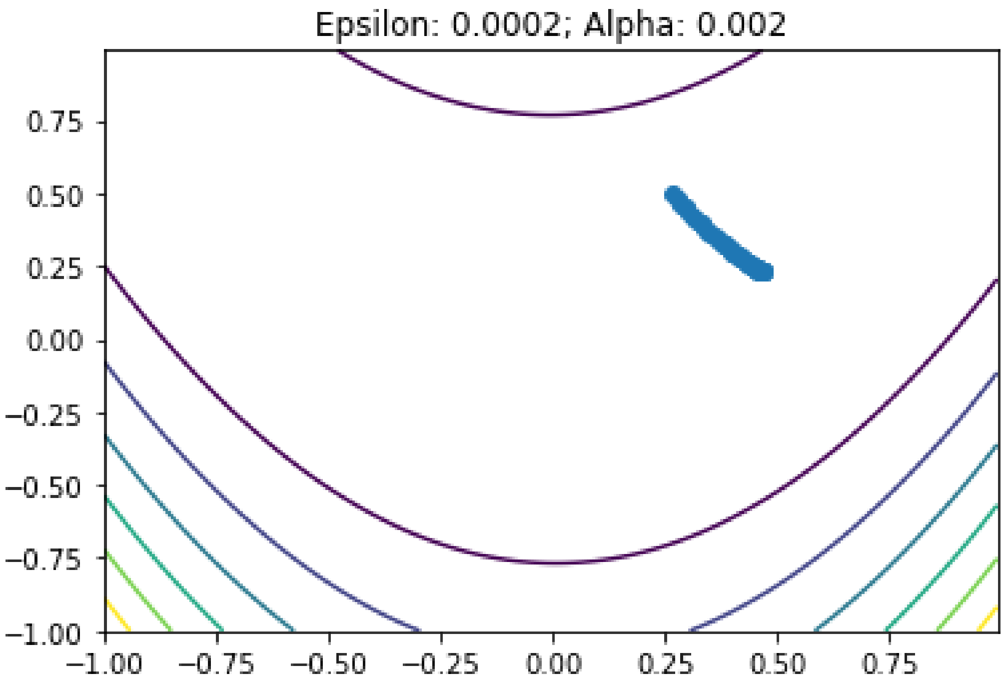

Learning Rate: 0.008
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75

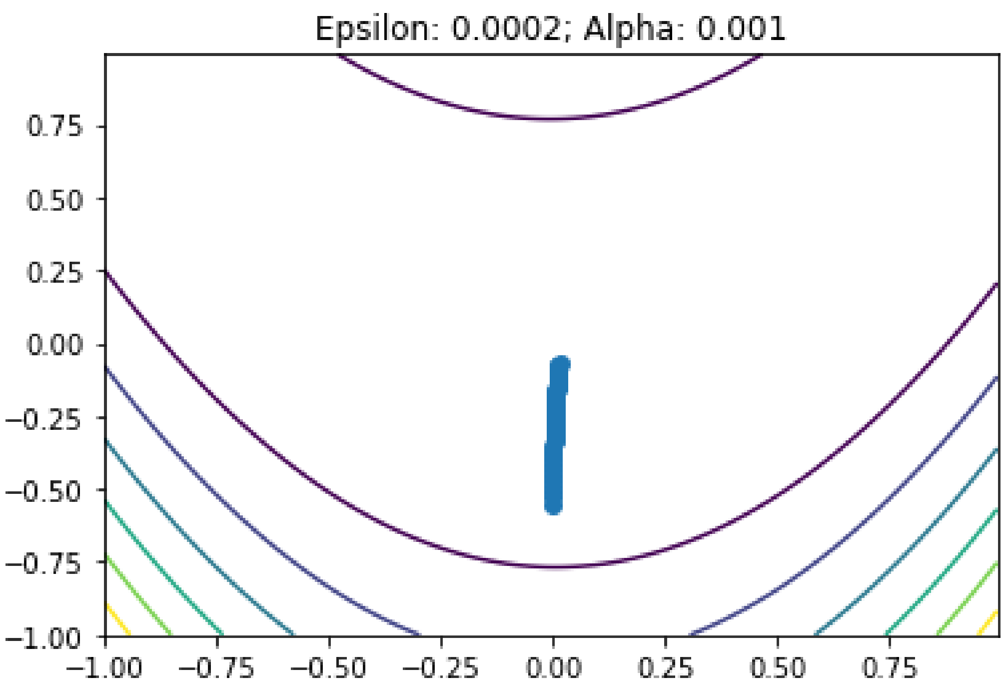
In [337]:
```
# 2.4

# gradient descent with momentum algorithm
    # adapted from the Gradient_Descent notebook
def grad_descent_with_momentum(starting_point=None, iterations=10, alpha=.9, epsilon=
    if starting_point:
        point = starting_point
    else:
        point = np.random.uniform(-1,1,size=2)
    trajectory = [point]
    # initialize the velocity vector
    v = np.zeros(point.size)

    for i in range(iterations):
        # call the same instantaneous-gradient-finding function
        grad = grad_f(point)
        # create the update to the point in the gradient descent with momentum
        # algorithm by summing the instantaneous gradient and the past gradients
        # in the velocity vector
        v = alpha*v + epsilon*grad
        # update the point in the gradient descent trajectory
        point = point - v
        trajectory.append(point)
    return np.array(trajectory)

# test different values for epsilon and alpha

np.random.seed(10)
traj = grad_descent_with_momentum(iterations=50, epsilon=.0002, alpha=.005)
traj1 = grad_descent_with_momentum(iterations=50, epsilon=.0002, alpha=.002)
traj2 = grad_descent_with_momentum(iterations=50, epsilon=.0002, alpha=.001)
traj3 = grad_descent_with_momentum(iterations=50, epsilon=.0002, alpha=.005)
traj4 = grad_descent_with_momentum(iterations=50, epsilon=.0005, alpha=.005)
traj5 = grad_descent_with_momentum(iterations=50, epsilon=.001, alpha=.005)

trajs = [traj, traj1, traj2, traj3, traj4, traj5]
eps = [.0002, .0002, .0002, .0002, .0005, .001]
alphs = [.005, .002, .001, .005, .005, .005]


for num in range(6):
    t = trajs[num]
    e = eps[num]
    a = alphs[num]

    fig, ax = plt.subplots()
    CS = ax.contour(X, Y, Z)
    x= t[:,0]
```

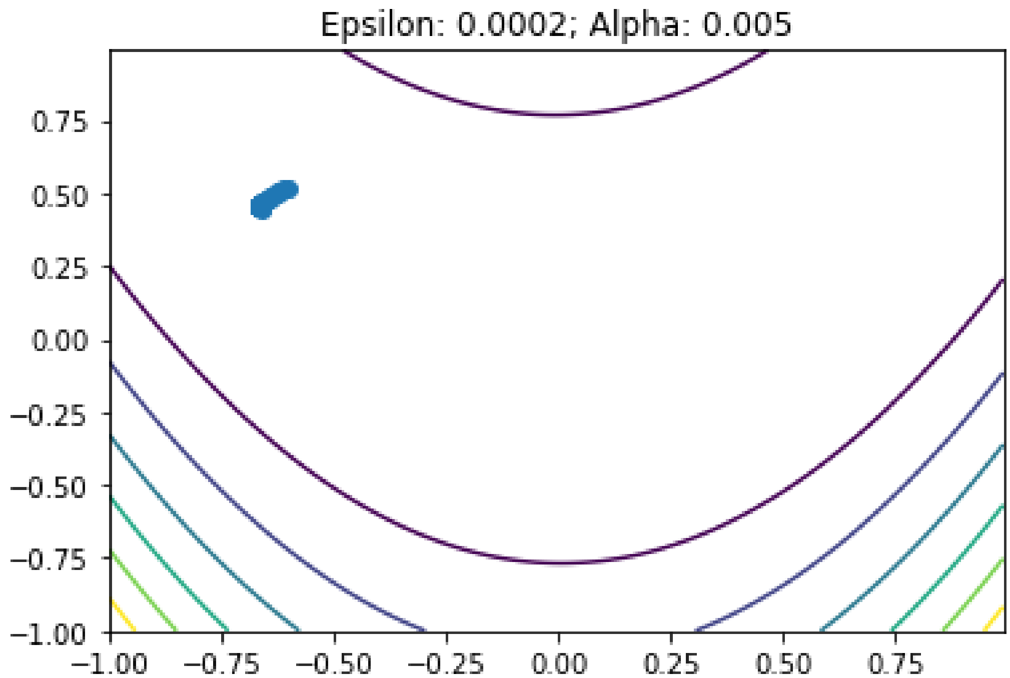
```
    y= t[:,1]
    plt.title("Epsilon: " + str(e) + "; Alpha: " + str(a))
    plt.plot(x,y,'-o')
```

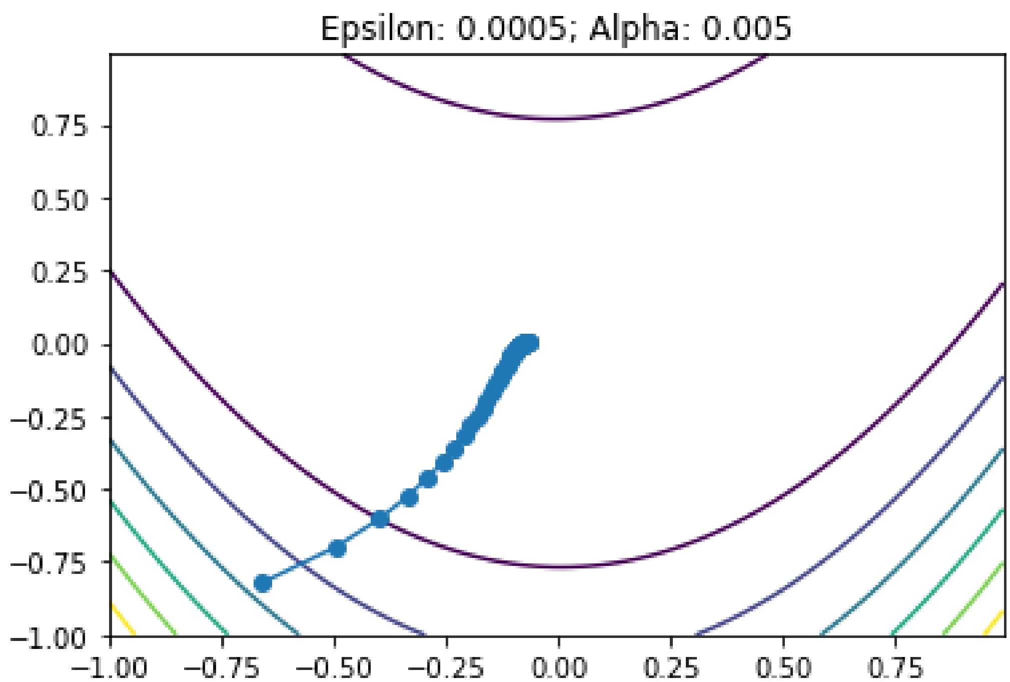
```
# Each of the different tested hyperparameters for tuning the weight of the
# current gradient (alpha) end up sending the trajectory in the same
# direction, but the larger learning rates look like they make it head there
# faster.

# The same can be said for the tested epsilon paratmeters that control
# the weight given to the velocity vector. It looks like they're all
# headed towards a minimum, but the ones that use larger parameters look
# like they'd get there faster.
```

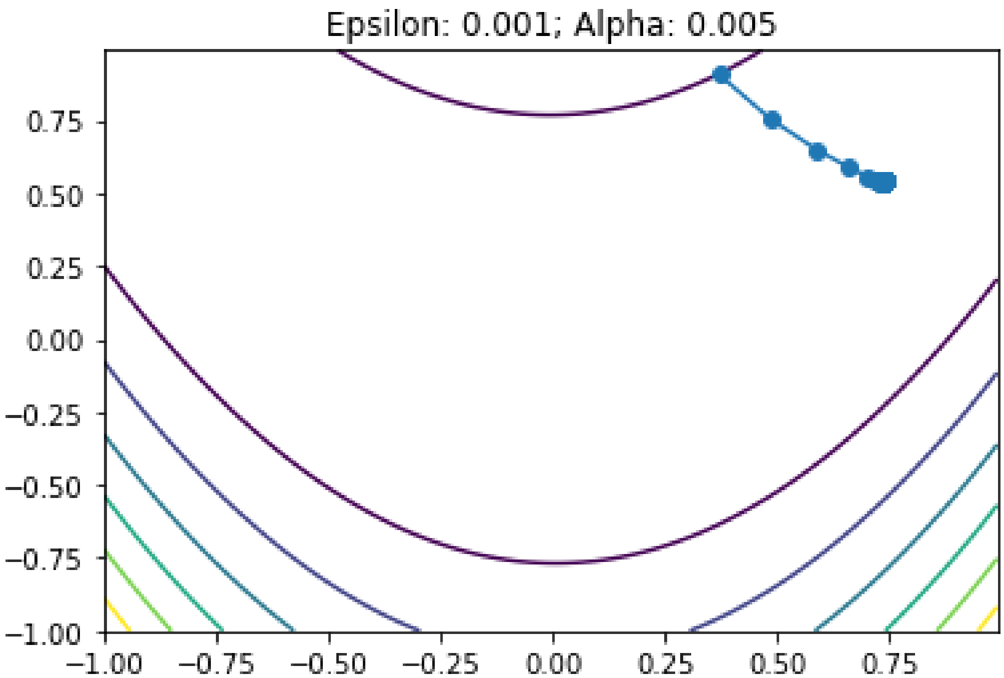



Epsilon: 0.0002; Alpha: 0.002
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75


Epsilon: 0.0002; Alpha: 0.001
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75


Epsilon: 0.0002; Alpha: 0.005
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75


Epsilon: 0.0005; Alpha: 0.005
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75



3.1
Had to rearrange things a bit from Nielsen's notation to account for the data.
$\frac{\partial L}{\partial c} = \sum(\hat{y} - y)$
$\frac{\partial L}{\partial V} = (\hat{y} - y) \cdot out_2^T$
$\frac{\partial L}{\partial b2} = \sum(V^T \cdot (\hat{y} - y)) \circ out_2$
$\frac{\partial L}{\partial W2} = (V^T \cdot (\hat{y} - y)) \circ out_2 \cdot out_1^T$
$\frac{\partial L}{\partial b1} = \sum W2^T \cdot (V^T \cdot (\hat{y} - y)) \circ out_2) \circ out_1$
$\frac{\partial L}{\partial W1} = (W2^T \cdot (V^T \cdot (\hat{y} - y)) \circ out_2) \circ out_1) \cdot X^T$
$out_1 = ReLU(W1 \cdot X + b1)$
$out_2 = ReLU(W2 \cdot out_1 + b2)$

In [338]:
```
# 3.2
def grad_f(param_vec, x, y):

    # unpack the parameters
    W1 = param_vec[0:6].reshape(3, 2)
    W2 = param_vec[6:12].reshape(2, 3)
    v = param_vec[12:18].reshape(3, 2)
    b1 = param_vec[18:21].reshape(3, 1)
    b2 = param_vec[21:23].reshape(2, 1)
    c = param_vec[23:26].reshape(3, 1)

    # forward pass
    a1 = W1.dot(x) + b1
    H1 = ReLu(a1)
```

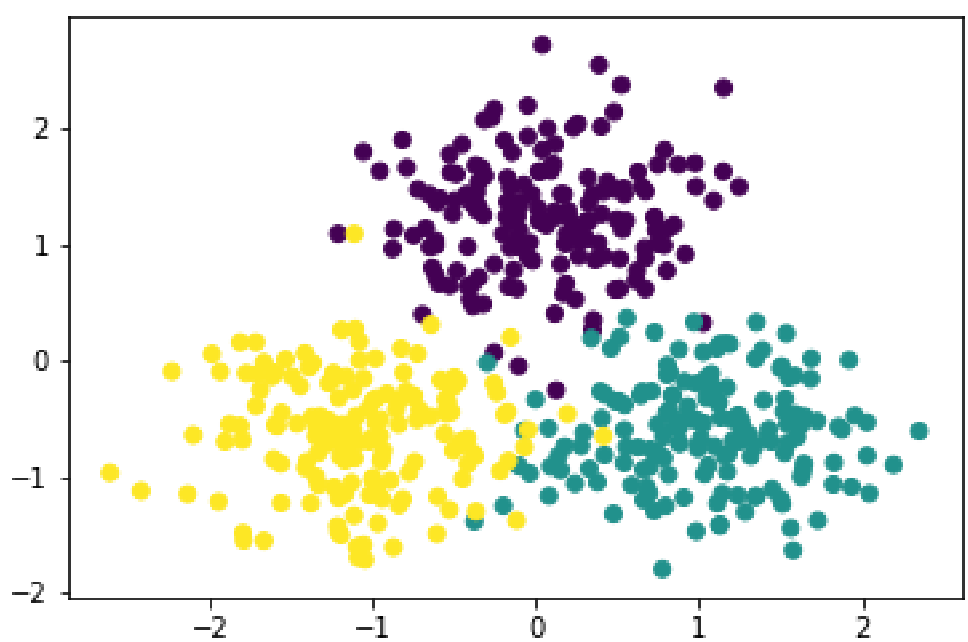
```
    a2 = W2.dot(H1) + b2
    H2 = ReLu(a2)
    Z = v.dot(H2) + c
    y_hat = softmax(Z)

    # the partials, as described above
    d_v = (y_hat - y).dot(H2.T) # d_V
    d_c = (y_hat - y).sum(axis = 1) # d_c
    d_W2 = ((V.T.dot((y_hat - y))) * (H2 > 0)).dot(H1.T) # d_W2
    d_b2 = ((V.T.dot((y_hat - y))) * (H2 > 0)).sum(axis = 1) # d_b2
    d_W1 = (W2.T.dot((V.T.dot((y_hat - y))) * (H2 > 0)) * (H1 > 0)).dot(x.T)  # d_W1
    d_b1 = (W2.T.dot((V.T.dot((y_hat - y))) * (H2 > 0)) * (H1 > 0)).sum(axis = 1) # 

    # repack the parameters
    z = [d_W1, d_W2, d_b1, d_b2, d_v, d_c]
    z = [param.flatten() for param in z]
    z = np.concatenate(z)

    # return the gradient
    return(z)


def grad_descent(x, y,  iterations=10, learning_rate=1e-2):
    # initialize the weights and biases
    v = np.random.uniform(-.1, .1, size = 6).reshape(3, 2)
    w2 = np.random.uniform(-.1, .1, size = 6).reshape(2, 3)
    w1 = np.random.uniform(-.1, .1, size = 6).reshape(3, 2)
    c = np.random.uniform(-.1, .1, size = 3).reshape(3, 1)
    b2 = np.random.uniform(-.1, .1, size = 2).reshape(2, 1)
    b1 = np.random.uniform(-.1, .1, size = 3).reshape(3, 1)

    # pack the parameters
    point = [param.flatten() for param in [w1, w2, v, b1, b2, c]]
    point = np.concatenate(point)

    # initiate the trajectory and loss vectors
    trajectory = [point]
    losses = [loss(y, y_hat(x, point))]

    # perform the gradient descent algorithm
    for i in range(iterations):
        grad = grad_f(point, x, y)
        point = point - learning_rate * grad
        trajectory.append(point)
        losses.append(loss(y, y_hat(x, point)))
    return (np.array(trajectory), losses)

# categorical cross-entropy loss function
```

```
def loss(y, y_hat):
    # cross entropy
    tot = y * np.log(y_hat)
    return  -tot.sum()

# function for computing a forward pass
def y_hat(input_vector, param_vec):
    # unpack the parameters
    weights1 = param_vec[0:6].reshape(3, 2)
    weights2 = param_vec[6:12].reshape(2, 3)
    weights3 = param_vec[12:18].reshape(3, 2)
    bias1 = param_vec[18:21].reshape(3, 1)
    bias2 = param_vec[21:23].reshape(2, 1)
    bias3 = param_vec[23:26].reshape(3, 1)

        # compute the output step-by-step
        # h1 = W1X + b1^T (3 x 3)
    h1 = ReLu(weights1.dot(input_vector) + bias1)
        # h2 = W2h1 + b2^T (2 x 3)
    h2 = ReLu(weights2.dot(h1) + bias2)
        # Z = Vh2 + c^T (3 x 3)
    z = weights3.dot(h2) + bias3
        # y_hat
    y_hat = softmax(z)
    # return the estimate
    return(y_hat)
```

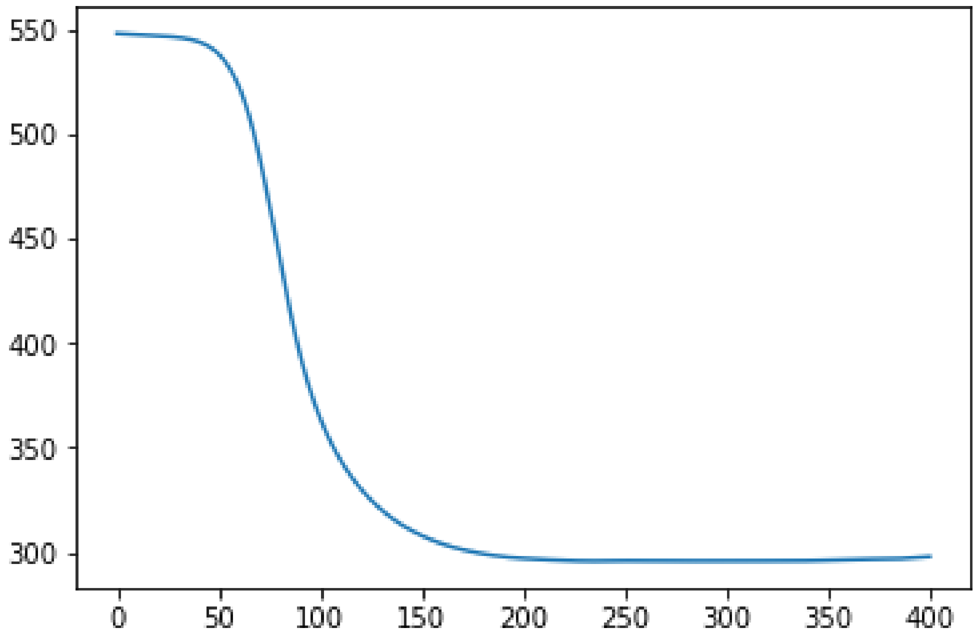
```
In [339]: # 3.3
# function for generating the Gaussian data, taken from the example notebook

import pandas as pd
def gen_gmm_data(n = 999, plot=False):
    # Fixing seed for repeatability
    np.random.seed(123)

    # Parameters of a normal distribuion
    mean_1 = [0, 2] ; mean_2 = [2, -2] ; mean_3 = [-2, -2]
    mean = [mean_1, mean_2, mean_3] ; cov = [[1, 0], [0, 1]]

    # Setting up the class probabilities
    n_samples = n
    pr_class_1 = pr_class_2 = pr_class_3 = 1/3.0
    n_class = (n_samples * np.array([pr_class_1,pr_class_2, pr_class_3])).astype(int

    # Generate sample data
    for i in range(3):
        x1,x2 = np.random.multivariate_normal(mean[i], cov, n_class[i]).T
        if (i==0):
```

```
            xs = np.array([x1,x2])
            cl = np.array([n_class[i]*[i]])
        else:
            xs_new = np.array([x1,x2])
            cl_new = np.array([n_class[i]*[i]])
            xs = np.concatenate((xs, xs_new), axis = 1)
            cl = np.concatenate((cl, cl_new), axis = 1)

        # One hot encoding classes
    y = pd.Series(cl[0].tolist())
    y = pd.get_dummies(y).as_matrix()

    # Normalizing data (prevents overflow errors)
    mu = xs.mean(axis = 1)
    std = xs.std(axis = 1)
    xs = (xs.T - mu) / std

    return xs, y, cl
```

In [342]:
```
x,y,cl = gen_gmm_data(500)
plt.scatter(x[:,0], x[:,1], c=cl)

# I transposed x and y to make the computations easier, since the dimensions in the
# assignment are a little different from those in the example.
x = x.T
y = y.T
```



```
In [354]: # 3.4
          traj, losses = grad_descent(x, y, iterations=400,learning_rate=.0002)
          # here's the plot of the losses along the paramters trajectory - looks like it's lea
          # and I had to play with the learning rate a lot
          plt.plot(losses)

Out[354]: [<matplotlib.lines.Line2D at 0x119eb0550>]
```




```
In [356]: # 3.5

          # another gradient descent with momentum function
          def grad_descent_with_momentum_2(x, y,  iterations=10, alpha=.9, epsilon=10):
              # initialize the weights and biases
              v = np.random.uniform(-.1, .1, size = 6).reshape(3, 2)
              w2 = np.random.uniform(-.1, .1, size = 6).reshape(2, 3)
              w1 = np.random.uniform(-.1, .1, size = 6).reshape(3, 2)
              c = np.random.uniform(-.1, .1, size = 3).reshape(3, 1)
              b2 = np.random.uniform(-.1, .1, size = 2).reshape(2, 1)
              b1 = np.random.uniform(-.1, .1, size = 3).reshape(3, 1)

              # pack the parameters
              point = [param.flatten() for param in [w1, w2, v, b1, b2, c]]
              point = np.concatenate(point)
              # initialize the velocity vector
              v = np.zeros(point.size)
```

```
    # initiate the trajectory and loss vectors
    trajectory = [point]
    losses = [loss(y, y_hat(x, point))]

    # perform the gradient descent algorithm
    for i in range(iterations):
        grad = grad_f(point, x, y)
        # create the update to the point in the gradient descent with momentum
        # algorithm by summing the instantaneous gradient and the past gradients
        # in the velocity vector
        v = alpha*v + epsilon*grad
        # update the point in the gradient descent trajectory
        point = point - v
        trajectory.append(point)
        losses.append(loss(y, y_hat(x, point)))
    return (np.array(trajectory), losses)
```

In [359]:
```
# It looks like, with momentum, it learns about the same, and
# maybe a little bit less efficiently.
# That is, it took longer to start descending down the loss
# slope, and it didn't traverse
# down the slope as quickly. Although, again,
# this took a lot of parameter "tuning" (guessing)
traj1, losses1 = grad_descent_with_momentum_2(x, y, iterations = 400, alpha = .001, e
plt.plot(losses1)
```

Out[359]: [<matplotlib.lines.Line2D at 0x11b19d9b0>]