Anly590_HW1_ArcherA

October 15, 2018

1 Anly 590 HW 1

1.1 Alex Archer

worked with Brody Vogel and Jing Yan

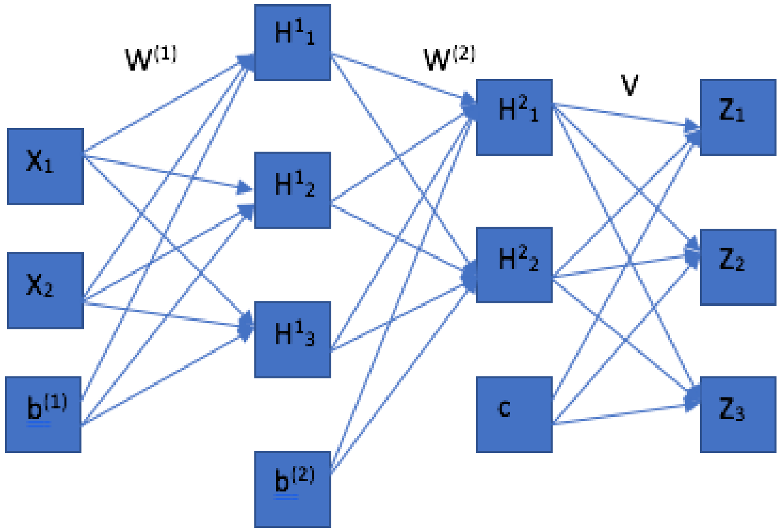
1.1.1 (1) Feedforward

1. Neural Network Diagram

2. $z_i = V_{i-1,0} relu(W^{(2)}_{0,0} relu(W^{(1)}_{0,0} x_1 + W^{(1)}_{0,1} x_2 + b^{(1)}_0) + W^{(2)}_{0,1} relu(W^{(1)}_{1,0} x_1 + W^{(1)}_{1,1} x_2 + b^{(1)}_1) + W^{(2)}_{0,2} relu(W^{(1)}_{2,0} x_1 + W^{(1)}_{2,1} x_2 + b^{(1)}_2) + b^{(2)}_0) + V_{i-1,1} relu(W^{(2)}_{1,0} relu(W^{(1)}_{0,0} x_1 + W^{(1)}_{0,1} x_2 + b^{(1)}_0) + W^{(2)}_{1,1} relu(W^{(1)}_{1,0} x_1 + W^{(1)}_{1,1} x_2 + b^{(1)}_1) + W^{(2)}_{1,2} relu(W^{(1)}_{2,0} x_1 + W^{(1)}_{2,1} x_2 + b^{(1)}_2) + b^{(2)}_1) + c_{i-1}, i = 1, 2, 3$
where $relu(x) = max(0, x)$
$\hat{y} = softmax(z) = \sigma(z)$, where $z = [z_1, z_2, z_3]$
$\hat{y} = \sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{3} e^{z_k}}$ for $j = 1, 2, 3$

3.

```
In [906]: import numpy as np
          import matplotlib.pyplot as plt
          import pandas as pd

In [907]: # activation functions

          def relu(x):
              return(np.maximum(x, 0))

          def softmax(z):
              return np.exp(z)/(np.sum(np.exp(z),axis=0))

In [908]: # weights, inputs
          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])
```




```
b2 = np.array([1,-1])
c = np.array([1,0,0])
X = np.array([[1,0,0],[-1,-1,1]])
```

In [910]:
```
# define the network

def ff_nn_2_ReLu(W1, W2, V, b1, b2, c, X):
    #z = softmax(np.dot(V,relu(np.dot(W2,relu(np.dot(W1,X)+b1)) + b2))+c)
    a1 = W1.dot(X) + b1.reshape(3,1)
    H1 = relu(a1)
    a2 = W2.dot(H1) + b2.reshape(2,1)
    H2 = relu(a2)
    a3 = V.dot(H2) + c.reshape(3,1)
    z = softmax(a3)
    return(z)
```

4.

In [911]:
```
# weights
ff_nn_2_ReLu(W1,W2,V,b1,b2,c,X)
```

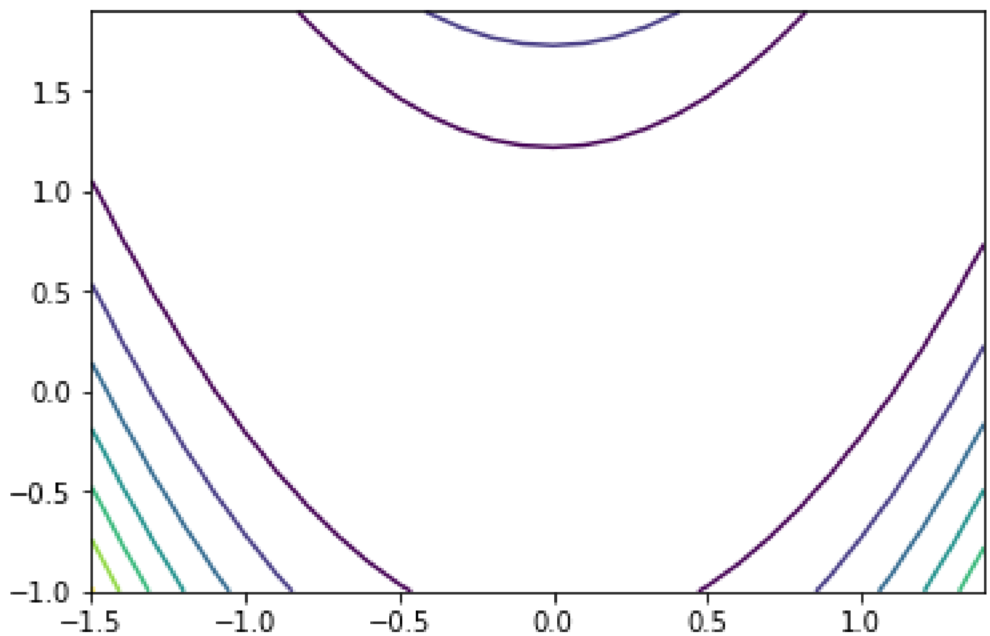
Out[911]:
```
array([[0.94649912, 0.84379473, 0.84379473],
       [0.04712342, 0.1141952 , 0.1141952 ],
       [0.00637746, 0.04201007, 0.04201007]])
```

1.1.2 (2) Gradient Descent

1. $f(x,y) = (1-x)^2 + 100(y-x^2)^2)$
$\frac{\partial f(x,y)}{\partial x} = -2(1-x) - 400x(y-x^2)$
$\frac{\partial f(x,y)}{\partial y} = 200(y-x^2)$

2.

```
In [912]: # contour plot
          delta = .1
          x = np.arange(-1.5, 1.5, delta)
          y = np.arange(-1.0, 2, delta)
          X, Y = np.meshgrid(x, y)
          Z = (1-X)**2 + 100*(Y-X**2)**2
          fig, ax = plt.subplots()
          CS = ax.contour(X, Y, Z)
```



3.

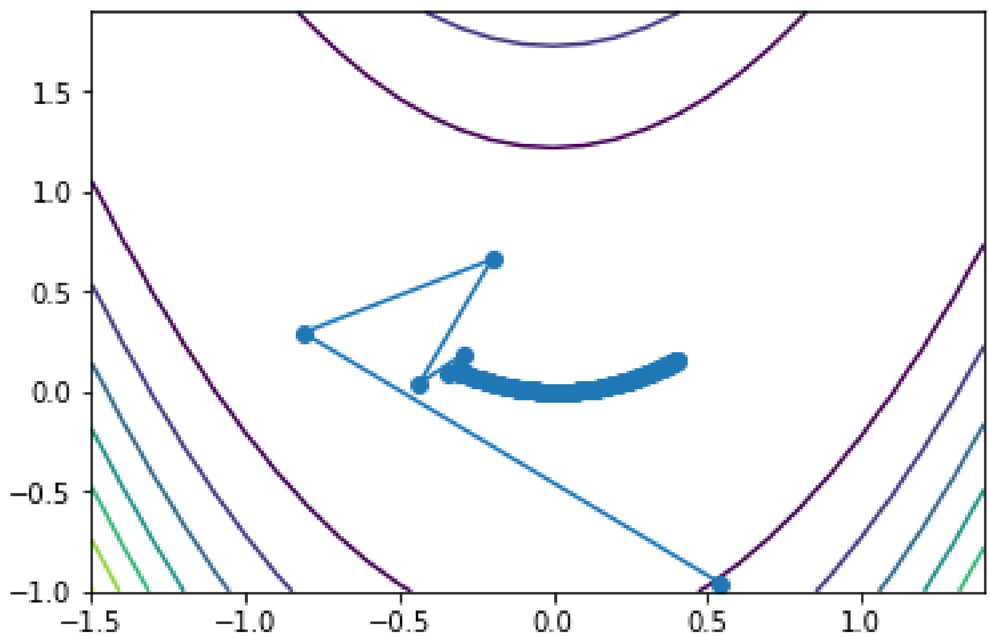
```
In [913]: # grad func
          def grad_f(vector):
              x, y = vector
              df_dx = -2 * (1-x) - 400*x * (y-x**2)
              df_dy = 200 * (y-x**2)
              return np.array([df_dx, df_dy])
```

```
In [914]: # grad descent w/o momentum
          def grad_descent(starting_point=None, iterations=100, learning_rate=.1):
```

```python
    if starting_point:
        point = starting_point
    else:
        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)
        #print(point)
    return np.array(trajectory)
```

In [915]:
```python
np.random.seed(10)
traj = grad_descent(iterations=100, learning_rate=.005)

fig, ax = plt.subplots()
CS = ax.contour(X, Y, Z)
x= traj[:,0]
y= traj[:,1]
plt.plot(x,y,'-o')
```

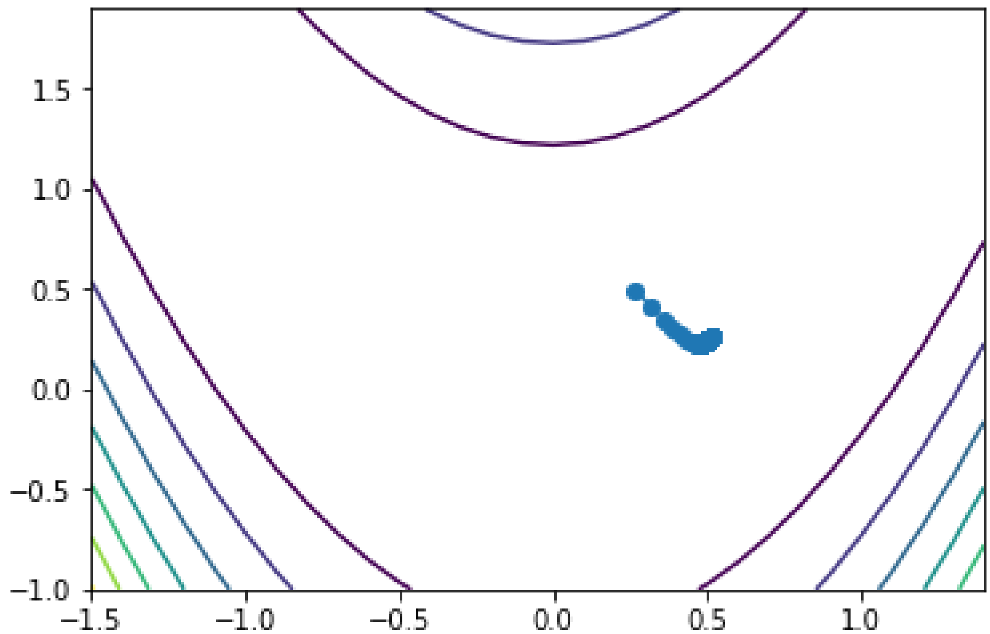
Out[915]: [<matplotlib.lines.Line2D at 0x124ae4908>]



In [916]:
```python
traj = grad_descent(iterations=100, learning_rate=.001)
```

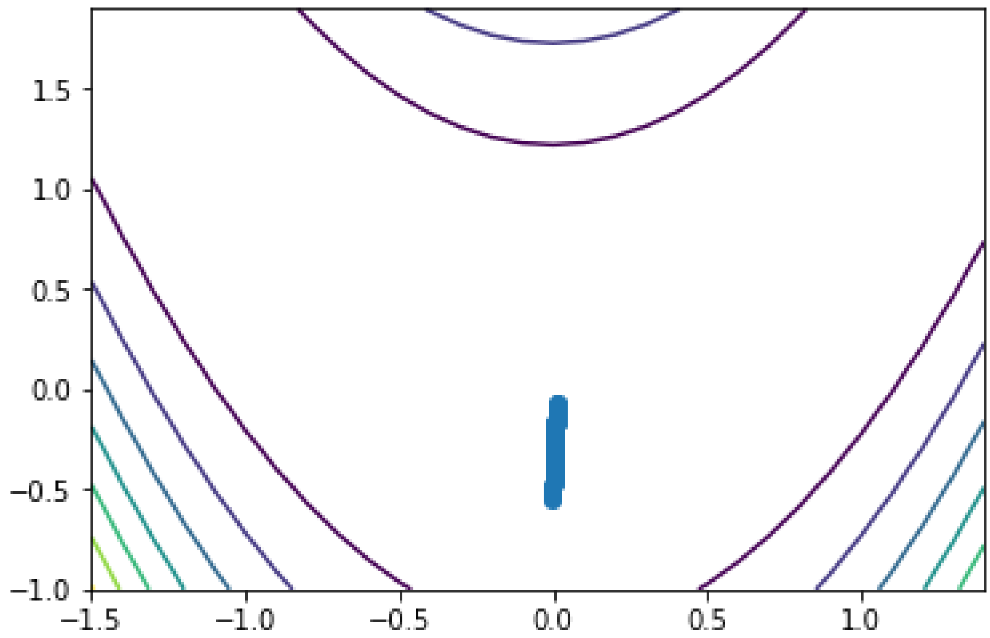
```
fig, ax = plt.subplots()
CS = ax.contour(X, Y, Z)
x= traj[:,0]
y= traj[:,1]
plt.plot(x,y,'-o')
```

Out[916]: [<matplotlib.lines.Line2D at 0x124c090f0>]



In [917]:
```
traj = grad_descent(iterations=100, learning_rate=.0001)

fig, ax = plt.subplots()
CS = ax.contour(X, Y, Z)
x= traj[:,0]
y= traj[:,1]
plt.plot(x,y,'-o')
```

Out[917]: [<matplotlib.lines.Line2D at 0x124d29e80>]

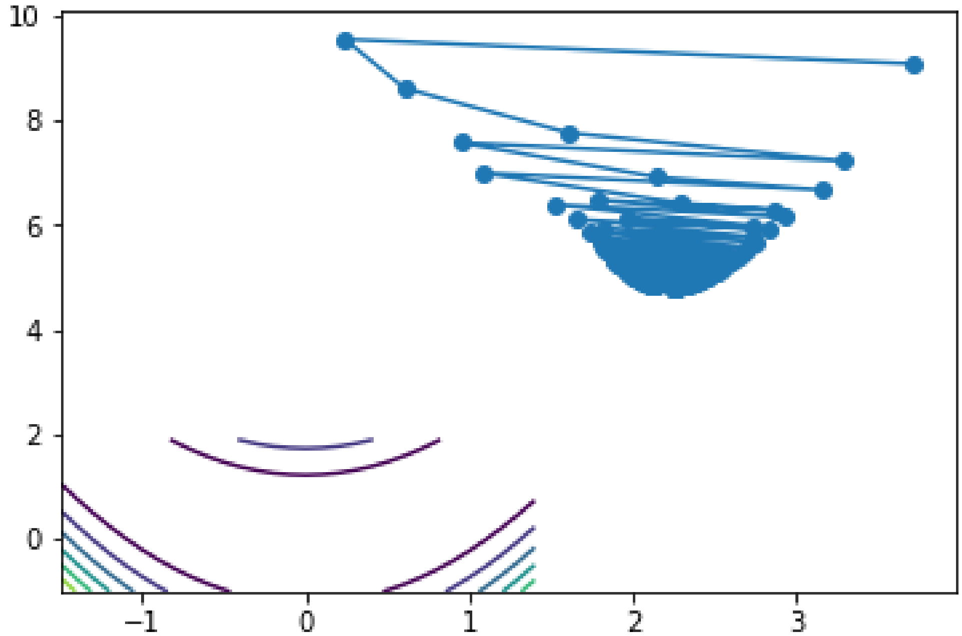


4.

In [918]:
```
# grad descent with momentum
def grad_descent_with_momentum(starting_point=None, iterations=10, alpha=.9, epsilon=
    if starting_point:
        point = starting_point
    else:
        point = np.random.uniform(-10,10,size=2)
    trajectory = [point]
    v = np.zeros(point.size)

    for i in range(iterations):
        grad = grad_f(point)
        v = alpha*v + epsilon*grad
        point = point - v
        #print(point)
        trajectory.append(point)
    return np.array(trajectory)
```

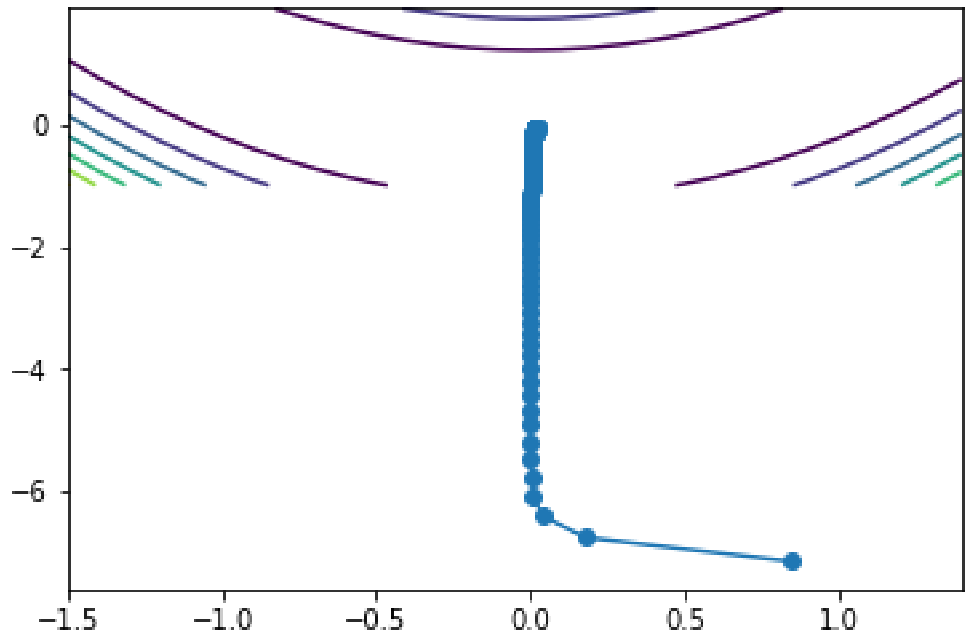
In [921]:
```
traj = grad_descent_with_momentum(iterations=100, epsilon=.0005, alpha=.025)

fig, ax = plt.subplots()
CS = ax.contour(X, Y, Z)
x= traj[:,0]
y= traj[:,1]
plt.plot(x,y,'-o')
```

Out[921]: [<matplotlib.lines.Line2D at 0x124ebb4a8>]

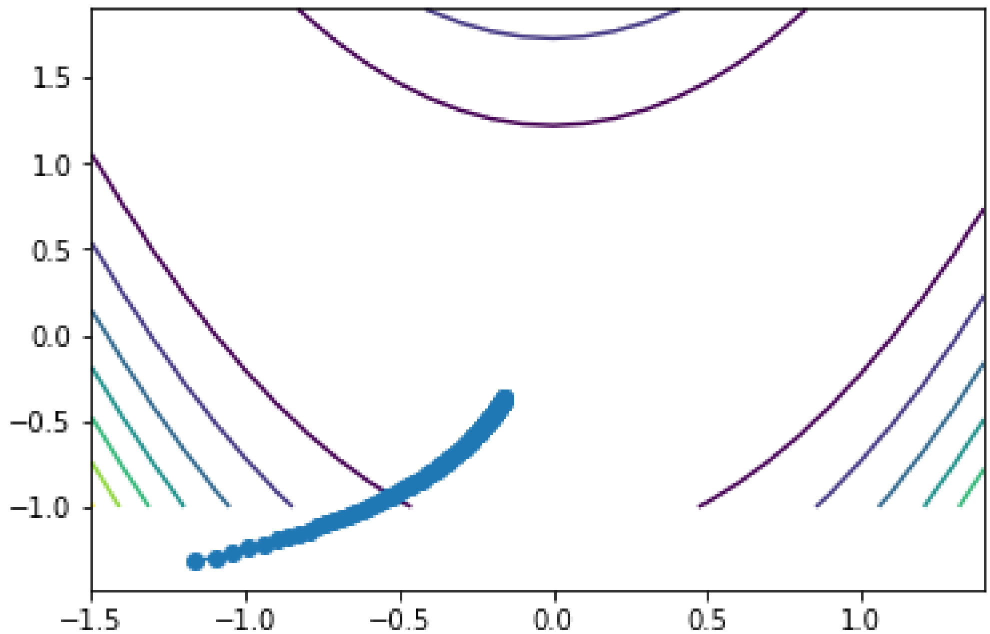



```
In [926]: traj = grad_descent_with_momentum(iterations=100, epsilon=.00025, alpha=.025)

          fig, ax = plt.subplots()
          CS = ax.contour(X, Y, Z)
          x= traj[:,0]
          y= traj[:,1]
          plt.plot(x,y,'-o')
```

Out[926]: [<matplotlib.lines.Line2D at 0x12531cf60>]




```
In [928]: traj = grad_descent_with_momentum(iterations=100, epsilon=.00005, alpha=.05)

          fig, ax = plt.subplots()
          CS = ax.contour(X, Y, Z)
          x= traj[:,0]
          y= traj[:,1]
          plt.plot(x,y,'-o')

Out[928]: [<matplotlib.lines.Line2D at 0x1254a2f28>]
```



1.1.3 (3) Backprop

1. $\delta^L = \frac{\partial L}{\partial a^L}\sigma\prime(z^L) = \overrightarrow{\hat{y}} - \overrightarrow{y}$
   $\frac{\partial L}{\partial V} = \delta^L a^{L-1} = (\overrightarrow{\hat{y}} - \overrightarrow{y}) \cdot out_2^T$
   $\frac{\partial L}{\partial c} = \frac{\partial L}{\partial a^L} = \sum(\overrightarrow{\hat{y}} - \overrightarrow{y})$
   $\frac{\partial L}{\partial W^{(2)}} = \delta^{L-1}a^{L-2} = V^T \cdot (\overrightarrow{\hat{y}} - \overrightarrow{y}) \odot \frac{\partial}{\partial in_2} relu(in_2) \cdot out_1^T$
   $\frac{\partial L}{\partial b^{(2)}} = \delta^{L-1} = \sum(V^T \cdot (\overrightarrow{\hat{y}} - \overrightarrow{y})) \odot \frac{\partial}{\partial in_2} relu(in_2)$
   $\frac{\partial L}{\partial W^{(1)}} = \delta^{L-2}X = \sum W^{(2)T} \cdot (V^T \cdot (\overrightarrow{\hat{y}} - \overrightarrow{y})) \odot \frac{\partial}{\partial in_2} relu(in_2)) \odot \frac{\partial}{\partial X} relu(X)) \cdot X^T$
   $\frac{\partial L}{\partial b^{(1)}} = \delta^{L-2}X = \sum W^{(2)T} \cdot (V^T \cdot (\overrightarrow{\hat{y}} - \overrightarrow{y})) \odot \frac{\partial}{\partial in_2} relu(in_2)) \odot \frac{\partial}{\partial X} relu(X))$
   where $relu(x) = max(x, 0), in_1 = W^{(1)} \cdot x + b^{(1)}, in_2 = W^{(2)} \cdot out_1 + b^{(2)}, out_1 = relu(in_1), out_2 = relu(in_2)$
   $\frac{\partial}{\partial x} relu(x) = \begin{cases} 1, & x > 0 \\ 0, & otherwise \end{cases}$

```
In [1941]: # new grad func for the NN from part 1
           def grad_f(params, x, y):
               # params
               W1 = params[0:6].reshape(3,2)
               W2 = params[6:12].reshape(2,3)
               V = params[12:18].reshape(3,2)
               b1 = params[18:21].reshape(3,1)
               b2 = params[21:23].reshape(2,1)
               c = params[23:26].reshape(3,1)
```

```
    # forward pass
    a1 = W1.dot(x) + b1
    H1 = relu(a1)
    a2 = W2.dot(H1) + b2
    H2 = relu(a2)
    a3 = V.dot(H2) + c
    y_hat = softmax(a3)

    # gradients
    d_V = (y_hat - y).dot(H2.T)
    d_c = (y_hat - y).sum(axis=1)
    d_W2 = ((V.T.dot((y_hat - y))) * (H2 > 0)).dot(H1.T)
    d_b2 = ((V.T.dot((y_hat - y))) * (H2 > 0)).sum(axis=1)
    d_W1 = (W2.T.dot((V.T.dot((y_hat - y))) * (H2 > 0)) * (H1 > 0)).dot(x.T)
    d_b1 = (W2.T.dot((V.T.dot((y_hat - y))) * (H2 > 0)) * (H1 > 0)).sum(axis=1)

    # return vector of gradients
    grad_vec = np.array([d_V, d_c, d_W2, d_b2, d_W1, d_b1])
    grad_vec = [p.flatten() for p in grad_vec]
    grad_vec = np.concatenate(grad_vec)
    return(grad_vec)
```

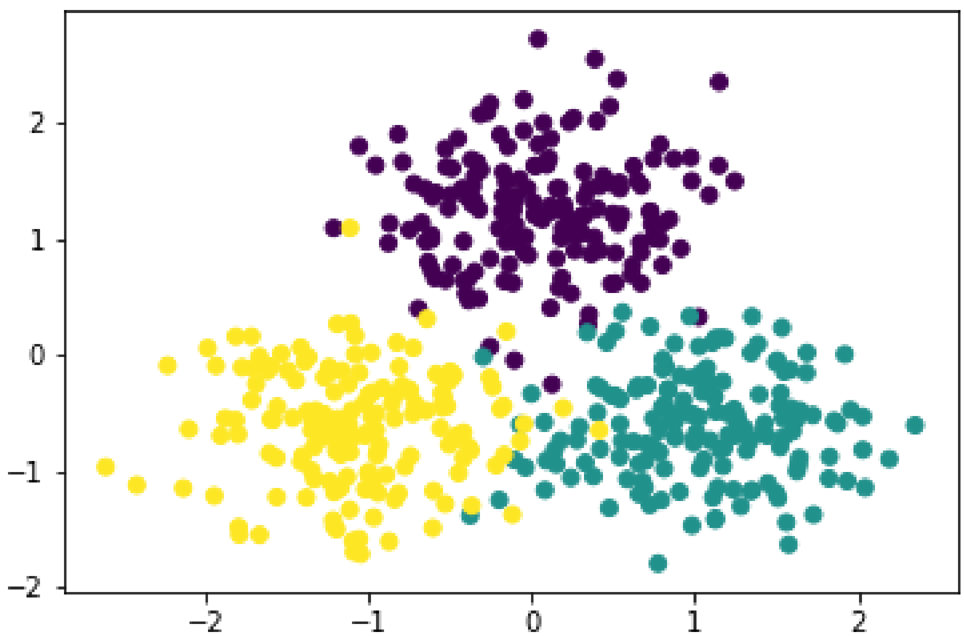
In [1913]:
```
def gen_gmm_data(n = 999, plot=False):
    # Fixing seed for repeatability
    np.random.seed(123)

    # 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(in

    # 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 [2225]:
```
x,y,cl = gen_gmm_data(500)
plt.scatter(x[:,0], x[:,1], c=cl.reshape(498,))
```

```
/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:28: FutureWarning: Method .as_mat
```

Out[2225]:
```
<matplotlib.collections.PathCollection at 0x14039e550>
```



In [1915]:
```
def loss(y, y_hat):
    # cross entropy
    tot = y * np.log(y_hat)
    return(-tot.sum())

def yhat(in_vec, params):
    # yhat helper
```

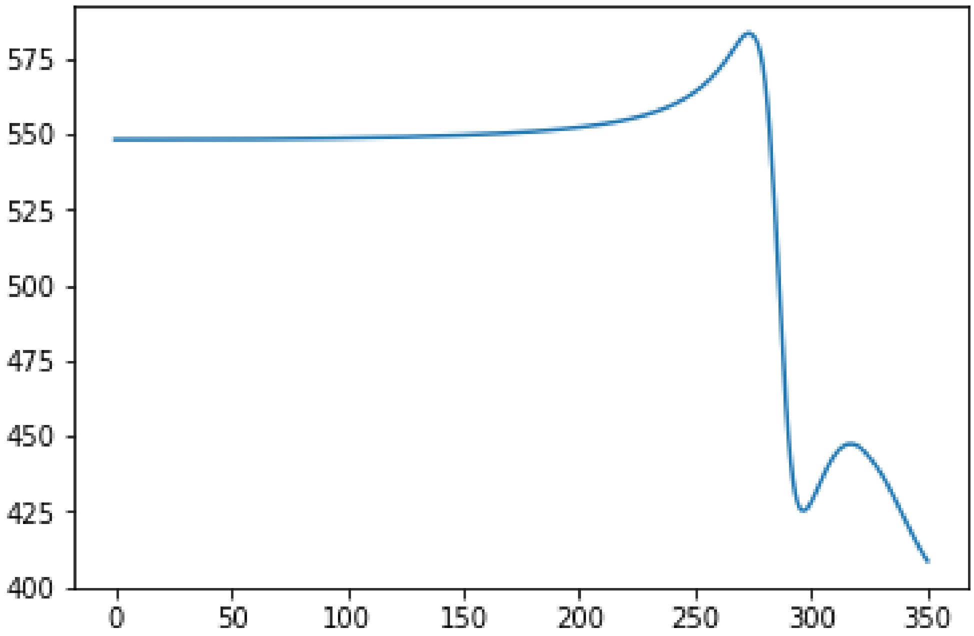
```
        w_1 = params[0:6].reshape(3,2)
        w_2 = params[6:12].reshape(2,3)
        v = params[12:18].reshape(3,2)
        b_1 = params[18:21].reshape(3,1)
        b_2 = params[21:23].reshape(2,1)
        c_0 = params[23:26].reshape(3,1)

        a_1 = w_1.dot(in_vec) + b_1
        H_1 = relu(a_1)
        a_2 = w_2.dot(H_1) + b_2
        H_2 = relu(a_2)
        a_3 = v.dot(H_2) + c_0
        y_hat = softmax(a_3)

        return(y_hat)
```

In [2296]:
```
# grad descent w/o momentum for NN in part 1
def grad_descent(x, y, iterations=10, learning_rate=.01):
    point = np.random.uniform(-.1, .1, size = 26).astype("float128")

    trajectory = [point]
    losses = [loss(y, yhat(x, point))]

    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)
```

In [2980]:
```
traj, losses = grad_descent(x.T, y.T, iterations=350,
                            learning_rate=.00025)
```

In [2981]:
```
plt.plot(losses)
```

Out[2981]:
```
[<matplotlib.lines.Line2D at 0x1512875c0>]
```

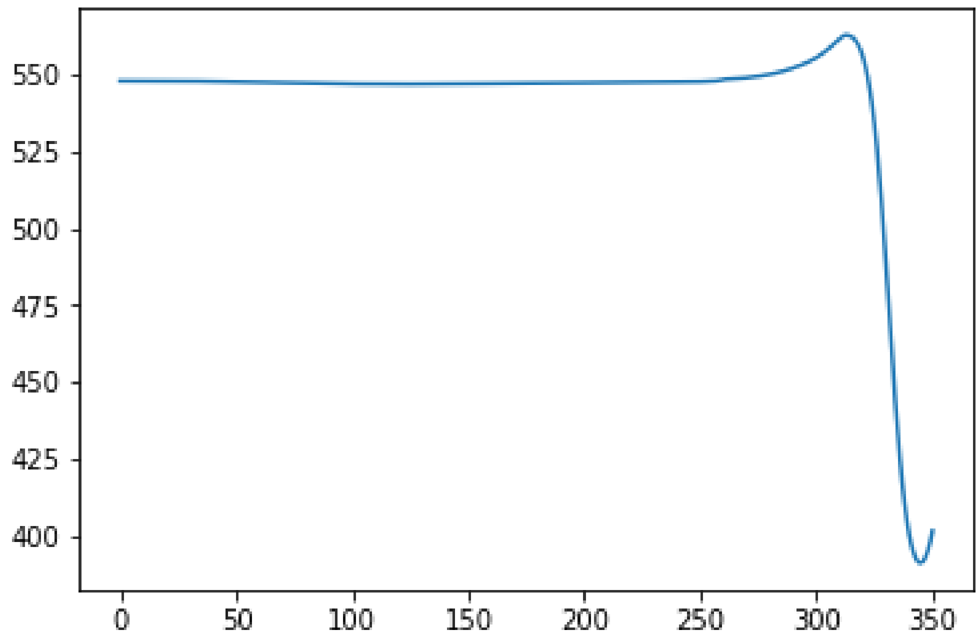



```
In [2773]: # gradient descent with momentum for NN in part 1
           def grad_descent_with_momentum2(x, y, iterations=100,
                                           alpha=.75, epsilon=.1):
               point = np.random.uniform(-.1, .1, size = 26).astype("float128")
               v = np.zeros(point.size)
               trajectory = [point]
               losses = [loss(y, yhat(x, point))]

               for i in range(iterations):
                   grad = grad_f(point, x, y)
                   v = alpha*v + epsilon*grad
                   point = point - v
                   trajectory.append(point)
                   losses.append(loss(y, y_hat(x, point)))
               return(np.array(trajectory), losses)
```

```
In [2769]: traj_m, losses_m = grad_descent_with_momentum2(x.T, y.T, iterations=350,
                                                          alpha=.001, epsilon=.0005)
```

```
In [2770]: plt.plot(losses_m)
```

```
Out[2770]: [<matplotlib.lines.Line2D at 0x14c3ea128>]
```



The gradient descent with momentum loss appears to be similar to gradient descent without momentum. They converge at a similar rate. The parameters likely need to be tuned a bit more to find the optimal results.