Neural Networks - Prolog 2 - Logistic Regression

September 13, 2016

Marcin Junczys-Dowmunt Machine Translation Marathon 2016

1 Introduction to Neural Networks

1.1 Prolog 2: Logistic Regression

2 The Iris Data Set

Iris setosa
Iris virginica
Iris vericolor

In [5]:
```
import pandas
data = pandas.read_csv("iris.csv", header=None,
                       names=["Sepal length", "Sepal width",
                              "Petal length", "Petal width",
                              "Species"])
data[:8]
```

Out[5]:

	Sepal length	Sepal width	Petal length	Petal width	Species
0	5.2	3.4	1.4	0.2	Iris-setosa
1	5.1	3.7	1.5	0.4	Iris-setosa
2	6.7	3.1	5.6	2.4	Iris-virginica
3	6.5	3.2	5.1	2.0	Iris-virginica
4	4.9	2.5	4.5	1.7	Iris-virginica
5	6.0	2.7	5.1	1.6	Iris-versicolor
6	5.7	2.6	3.5	1.0	Iris-versicolor
7	5.0	2.0	3.5	1.0	Iris-versicolor

$$\text{Set of classes:} \qquad C = \{c_1, c_2, \cdots, c_k\} \qquad |C| = k$$

$$\text{Training data:} X = \begin{bmatrix} 1 & x_1^{(1)} & \cdots & x_n^{(1)} \\ 1 & x_1^{(2)} & \cdots & x_n^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & \cdots & x_n^{(m)} \end{bmatrix} \qquad \vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix} = \begin{bmatrix} c_2 \\ c_1 \\ \vdots \\ c_1 \end{bmatrix}$$

$$\dim X = m \times (n+1) \qquad \dim \vec{y} = m \times 1$$

$$\text{Set of classes:} \qquad C = \{c_1, c_2, \cdots, c_k\} \qquad |C| = k$$

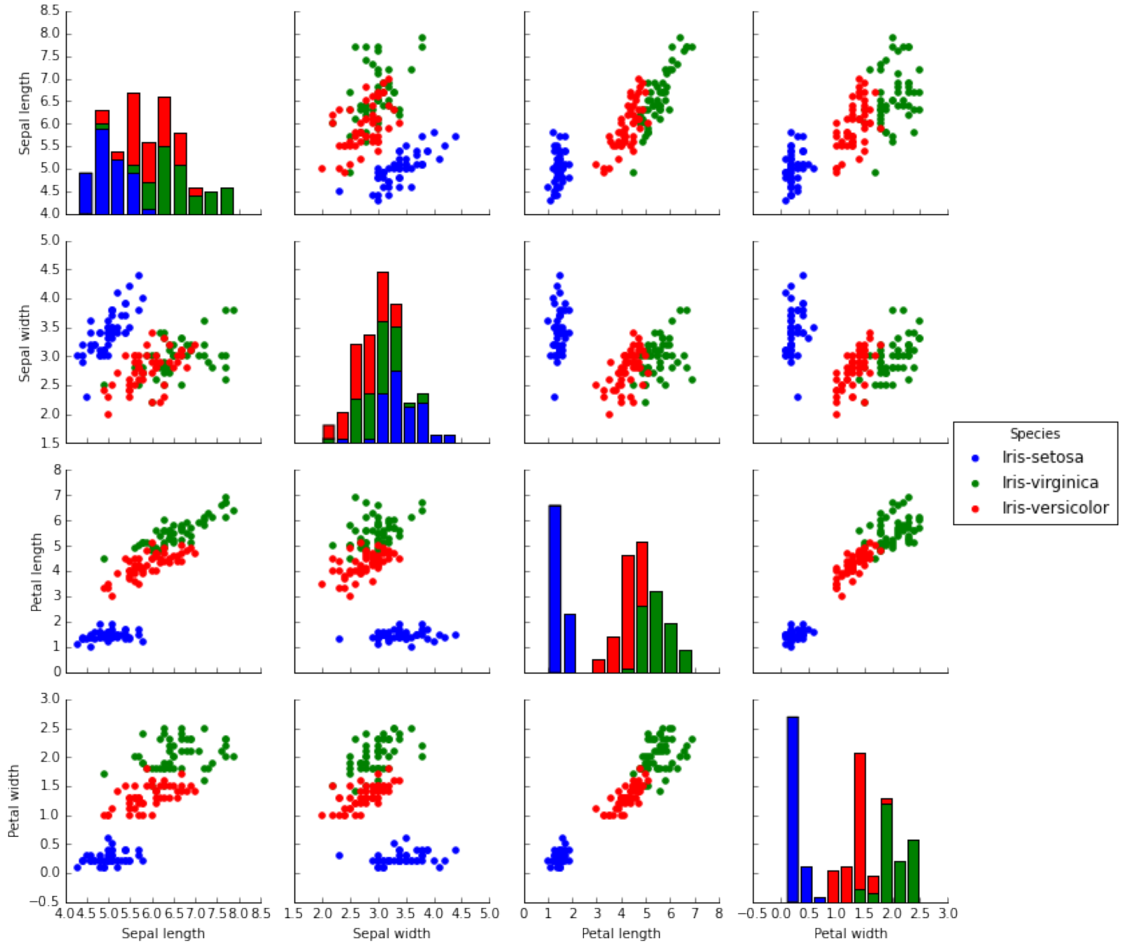
Training data (with indicator matrix): $X = \begin{bmatrix} 1 & x_1^{(1)} & \cdots & x_n^{(1)} \\ 1 & x_1^{(2)} & \cdots & x_n^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & \cdots & x_n^{(m)} \end{bmatrix}$ $\quad Y = \begin{bmatrix} \delta(c_1, y^{(1)}) & \ldots & \delta(c_k, y^{(1)}) \\ \delta(c_1, y^{(2)}) & \ldots & \delta(c_k, y^{(2)}) \\ \vdots & \ddots & \vdots \\ \delta(c_1, y^{(m)}) & \ldots & \delta(c_k, y^{(m)}) \end{bmatrix}$

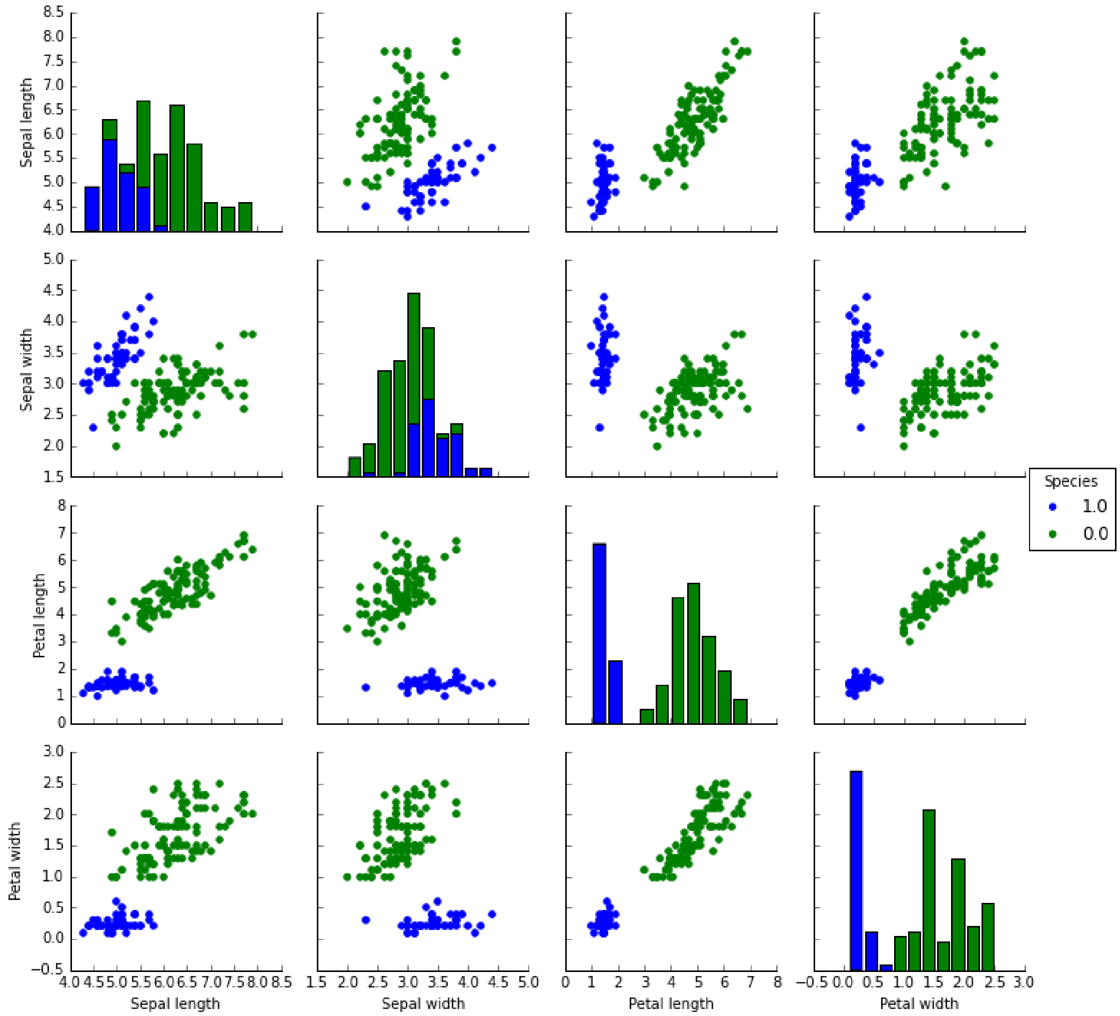
$$\delta(x, y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$$

$$\dim X = m \times (n+1) \qquad \dim Y = m \times k$$

In [6]:
```
import numpy as np

m = len(data)
X = np.matrix(data[["Sepal length", "Sepal width",
                    "Petal length", "Petal width"]])
X = np.hstack((np.ones(m).reshape(m,1), X))

y = np.matrix(data[["Species"]]).reshape(m,1)

def mapY(y, c):
    n = len(y)
    yBi = np.matrix(np.zeros(n)).reshape(n, 1)
    yBi[y == c] = 1.
    return yBi

def indicatorMatrix(y):
    classes = np.unique(y.tolist())
    Y = mapY(y, classes[0])
    for c in classes[1:]:
        Y = np.hstack((Y, mapY(y, c)))
    Y = np.matrix(Y).reshape(len(y), len(classes))
    return Y

Y = indicatorMatrix(y)
print(Y[:10])
```

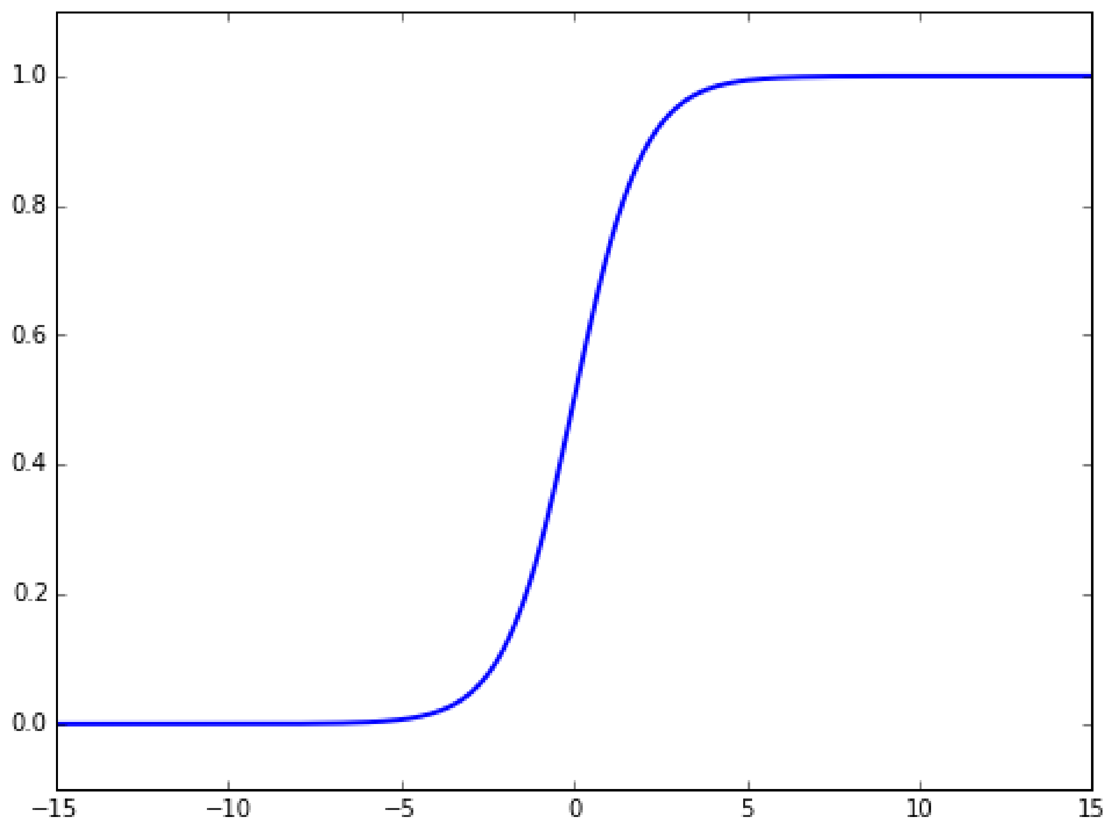
```
[[ 1.  0.  0.]
 [ 1.  0.  0.]
 [ 0.  0.  1.]
 [ 0.  0.  1.]
 [ 0.  0.  1.]
 [ 0.  1.  0.]
 [ 0.  1.  0.]
 [ 0.  1.  0.]
 [ 1.  0.  0.]
 [ 1.  0.  0.]]
```


Sepal length
Sepal width
Petal length
Petal width
Species
Iris-setosa
Iris-virginica
Iris-versicolor



2.1 Logistic function:

$$g(x) = \frac{1}{1 + e^{-x}}$$



2.2 Logistic regression model

- For a single feature vector:

$$h_\theta(x) = g(\sum_{i=0}^{n} \theta_i x_i) = \frac{1}{1 + e^{-\sum_{i=0}^{n} \theta_i x_i}}$$

- More compact in matrix form (batched):

$$h_\theta(X) = g(X\theta) = \frac{1}{1 + e^{-X\theta}}$$

2.3 The cost function (binary cross-entropy)

- Computed across the training batch:

$$J(\theta) = -\frac{1}{m}[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))]$$

- Essentially the same in matrix form with vectorized elementary functions

2.4 And its gradient

- Computed across the training batch for a single parameter:

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

- In matrix form:

$$\nabla J(\theta) = \frac{1}{|\vec{y}|}X^T\left(h_\theta(X) - \vec{y}\right)$$

- Looks the same as for linear regression, how come?

In [10]:
```
def h(theta, X):
    return 1.0/(1.0 + np.exp(-X*theta))

def J(h,theta,X,y):
    m = len(y)
    s1 = np.multiply(y, np.log(h(theta,X)))
    s2 = np.multiply((1 - y), np.log(1 - h(theta,X)))
    return -np.sum(s1 + s2, axis=0)/m

def dJ(h,theta,X,y):
    return 1.0/len(y)*(X.T*(h(theta,X)-y))
```

In [14]:
```
# Divide data into train and test set
XTrain, XTest = X[:100], X[100:]
YTrain, YTest = Y[:100], Y[100:]

# Initialize theta with zeroes
theta = np.zeros(5).reshape(5,1)

# Select only first column for binary classification (setosa vs. rest)
YTrain0 = YTrain[:,0]

print(J(h, theta, XTrain, YTrain0))
print(dJ(h, theta, XTrain, YTrain0))
```

```
[[ 0.69314718]]
[[ 0.18  ]
 [ 1.3125]
 [ 0.431 ]
 [ 1.3965]
 [ 0.517 ]]
```

2.5 Let's plug them into SGD

In [15]:
```
def mbSGD(h, fJ, fdJ, theta, X, y,
          alpha=0.01, maxSteps=10000, batchSize = 10):
    i = 0
    b = batchSize
    m = X.shape[0]
    while i < maxSteps:
        start = (i*b) % m
        end   = ((i+1)*b) % m
        if(end <= start):
```

```
                end = m
            Xbatch = X[start:end]
            Ybatch = y[start:end]
            theta = theta - alpha * fdJ(h, theta, Xbatch, Ybatch)
            i += 1
        return theta

    thetaBest = mbSGD(h, J, dJ, theta, XTrain, YTrain0,
                      alpha=0.01, maxSteps=10000, batchSize = 10)
    print(thetaBest)
```

```
[[ 0.3435233 ]
 [ 0.50679616]
 [ 1.85179133]
 [-2.83461652]
 [-1.26239494]]
```

2.6 Let's calculate test set probabilities

```
In [16]: probs = h(thetaBest, XTest)
         probs[:10]
```

```
Out[16]: matrix([[  2.21629846e-05],
                 [  9.81735045e-01],
                 [  5.12821349e-05],
                 [  3.93568995e-04],
                 [  8.34175760e-05],
                 [  9.93162028e-01],
                 [  5.84320330e-03],
                 [  8.82797896e-02],
                 [  9.71041386e-05],
                 [  8.59359714e-05]])
```

2.7 Are we done already?

2.8 The decision function (binary case)

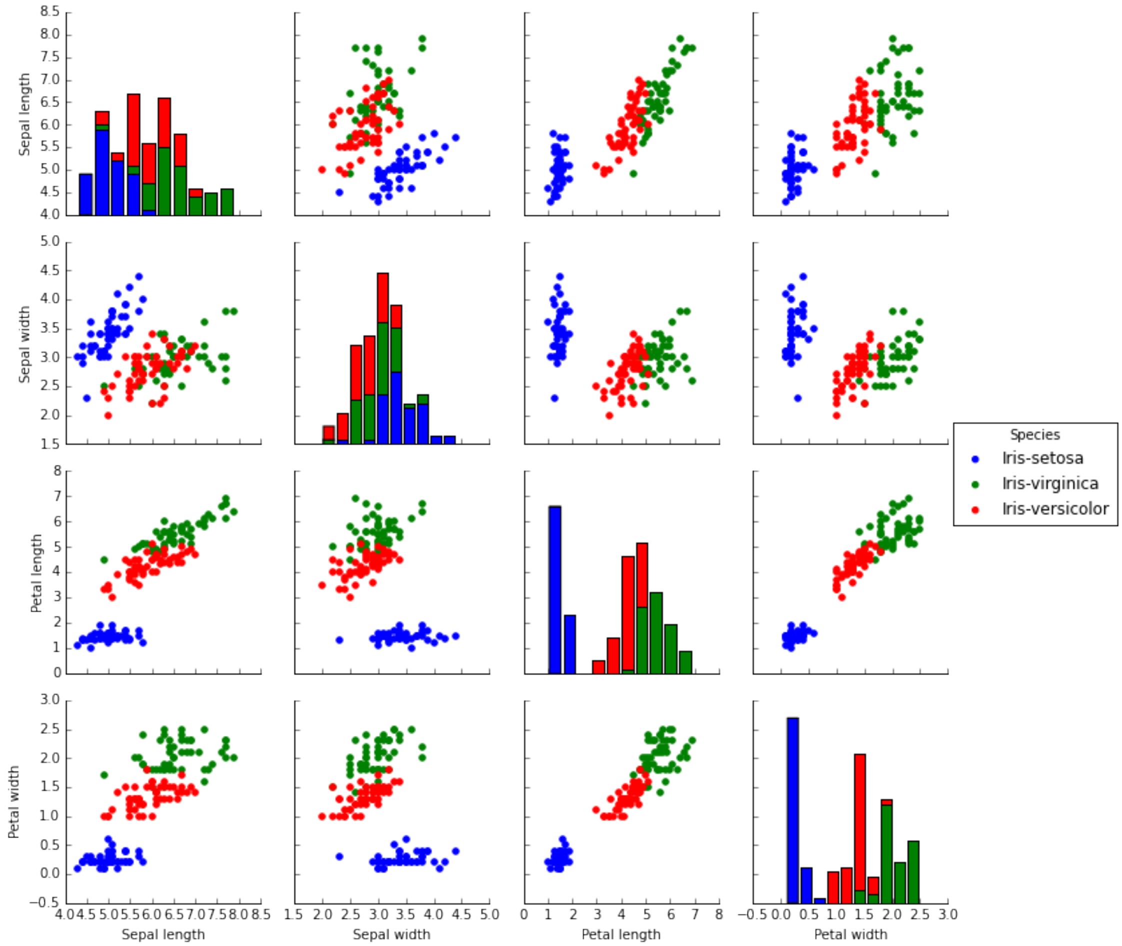
$$c = \begin{cases} 1 & \text{if } P(y=1|x;\theta) > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

$$P(y=1|x;\theta) = h_\theta(x)$$

```
In [17]: YTestBi = YTest[:,testClass]

         def classifyBi(X):
             prob = h(thetaBest, X).item()
             return (1, prob) if prob > 0.5 else (0, prob)

         acc = 0.0
         for i, rest in enumerate(YTestBi):
             cls, prob = classifyBi(XTest[i])
             print(int(YTestBi[i].item()), "<=>", cls, "-- prob:", round(prob, 4))
             acc += cls == YTestBi[i].item()
         print("Accuracy:", acc/len(XTest))
```

```
0 <=> 0 -- prob: 0.0
1 <=> 1 -- prob: 0.9817
0 <=> 0 -- prob: 0.0001
0 <=> 0 -- prob: 0.0004
0 <=> 0 -- prob: 0.0001
1 <=> 1 -- prob: 0.9932
0 <=> 0 -- prob: 0.0058
0 <=> 0 -- prob: 0.0883
0 <=> 0 -- prob: 0.0001
0 <=> 0 -- prob: 0.0001
1 <=> 1 -- prob: 0.9867
1 <=> 1 -- prob: 0.9876
0 <=> 0 -- prob: 0.0126
0 <=> 0 -- prob: 0.0
0 <=> 0 -- prob: 0.0071
1 <=> 1 -- prob: 0.9945
1 <=> 1 -- prob: 0.9851
0 <=> 0 -- prob: 0.0003
1 <=> 1 -- prob: 0.9851
0 <=> 0 -- prob: 0.0018
0 <=> 0 -- prob: 0.0008
1 <=> 1 -- prob: 0.9938
0 <=> 0 -- prob: 0.0001
0 <=> 0 -- prob: 0.034
0 <=> 0 -- prob: 0.0001
1 <=> 1 -- prob: 0.9988
0 <=> 0 -- prob: 0.0009
1 <=> 1 -- prob: 0.9916
0 <=> 0 -- prob: 0.0005
1 <=> 1 -- prob: 0.9956
0 <=> 0 -- prob: 0.0
0 <=> 0 -- prob: 0.0096
0 <=> 0 -- prob: 0.0
1 <=> 1 -- prob: 0.9765
0 <=> 0 -- prob: 0.0003
1 <=> 1 -- prob: 0.9437
0 <=> 0 -- prob: 0.0
0 <=> 0 -- prob: 0.0
1 <=> 1 -- prob: 0.9903
0 <=> 0 -- prob: 0.0136
0 <=> 0 -- prob: 0.0037
1 <=> 1 -- prob: 0.998
0 <=> 0 -- prob: 0.0036
1 <=> 1 -- prob: 0.9921
0 <=> 0 -- prob: 0.0001
1 <=> 1 -- prob: 0.9991
1 <=> 1 -- prob: 0.9937
0 <=> 0 -- prob: 0.0
0 <=> 0 -- prob: 0.0002
0 <=> 0 -- prob: 0.0001
Accuracy: 1.0
```

3 Multi-class logistic regression



3.1 Method 1: One-against-all

- We create three binary models: $h_{\theta_1}, h_{\theta_2}, h_{\theta_3}$, one for each class;
- We select the class with the highest probability.

- Is this property true?

$$\sum_{c=1,\dots,3} h_{\theta_c}(x) = \sum_{c=1,\dots,3} P(y = c|x; \theta_c) = 1$$

3.2 Softmax

- Multi-class version of logistic function is the softmax function

$$\text{softmax}(k, x_1, \dots, x_n) = \frac{e^{x_k}}{\sum_{i=i}^{n} e^{x_i}}$$

$$P(y = c|x; \theta_1, \dots, \theta_k) = \text{softmax}(c, \theta_1^T x, \dots, \theta_k^T x)$$

- Do we now have the following property?

$$\sum_{c=1,\ldots,3} P(y = c|x; \theta_c) = 1$$

```
In [20]: def softmax(X):
             return softmaxUnsafe(X - np.max(X, axis=1))

         def softmaxUnsafe(X):
             return np.exp(X) / np.sum(np.exp(X), axis=1)

         X = np.matrix([2.1, 0.5, 0.8, 0.9, 3.2]).reshape(1,5)
         P = softmax(X)

         print(X)
         print("Suma X =", np.sum(X, axis=1), "\n")
         print(P)
         print("Suma P =", np.sum(P, axis=1))
```

```
[[ 2.1  0.5  0.8  0.9  3.2]]
Suma X = [[ 7.5]]

[[ 0.20921428  0.04223963  0.05701754  0.06301413  0.62851442]]
Suma P = [[ 1.]]
```

```
In [21]: def trainMaxEnt(X, Y):
             n = X.shape[1]
             thetas = []
             for c in range(Y.shape[1]):
                 print("Training classifier for class %d" % c)
                 YBi = Y[:,c]
                 theta = np.matrix(np.random.normal(0, 0.1,n)).reshape(n,1)
                 thetaBest = mbSGD(h, J, dJ, theta, X, YBi,
                                   alpha=0.01, maxSteps=10000, batchSize = 10)
                 print(thetaBest)
                 thetas.append(thetaBest)
             return thetas

         thetas = trainMaxEnt(XTrain, YTrain);
```

```
Training classifier for class 0
[[ 0.40857659]
 [ 0.49964544]
 [ 1.84599176]
 [-2.8487475 ]
 [-1.22299215]]
Training classifier for class 1
[[ 0.76129434]
 [ 0.33037374]
 [-1.52574278]
 [ 0.99708116]
 [-2.08792796]]
Training classifier for class 2
[[-1.42873609]
 [-1.79503896]
```

```
[-2.23000675]
 [ 2.82362858]
 [ 3.19525365]]
```

3.3 Multi-class decision function

$$
\begin{aligned}
c &= \arg\max_{i \in \{1,\ldots,k\}} P(y = i | x; \theta_1, \ldots, \theta_k) \\
&= \arg\max_{i \in \{1,\ldots,k\}} \mathrm{softmax}(i, \theta_1^T x, \ldots, \theta_k^T x)
\end{aligned}
$$

```
In [22]: def classify(thetas, X):
             regs = np.matrix([(X*theta).item() for theta in thetas])
             probs = softmax(regs)
             return np.argmax(probs), probs

         print(YTest[:10])
         YTestCls = YTest * np.matrix((0,1,2)).T
         print(YTestCls[:10])

         acc = 0.0
         for i in range(len(YTestCls)):
             cls, probs = classify(thetas, XTest[i])
             correct = int(YTestCls[i].item())
             print(correct, "<=>", cls, " - ", correct == cls, np.round(probs, 4).tolist())
             acc += correct == cls
         print("Accuracy =", acc/float(len(XTest)))
```

```
[[ 0.  0.  1.]
 [ 1.  0.  0.]
 [ 0.  0.  1.]
 [ 0.  0.  1.]
 [ 0.  0.  1.]
 [ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]
 [ 0.  0.  1.]]
[[ 2.]
 [ 0.]
 [ 2.]
 [ 2.]
 [ 2.]
 [ 0.]
 [ 1.]
 [ 1.]
 [ 2.]
 [ 2.]]
2 <=> 2  -  True [[0.0, 0.2194, 0.7806]]
0 <=> 0  -  True [[0.9957, 0.0043, 0.0]]
2 <=> 2  -  True [[0.0, 0.0774, 0.9226]]
2 <=> 2  -  True [[0.0001, 0.1211, 0.8789]]
2 <=> 2  -  True [[0.0, 0.3455, 0.6545]]
0 <=> 0  -  True [[0.9995, 0.0005, 0.0]]
1 <=> 1  -  True [[0.0097, 0.8617, 0.1286]]
1 <=> 1  -  True [[0.1576, 0.8177, 0.0247]]
2 <=> 2  -  True [[0.0, 0.1076, 0.8924]]
```

```
2 <=> 2  -  True [[0.0, 0.3269, 0.6731]]
0 <=> 0  -  True [[0.9965, 0.0035, 0.0]]
0 <=> 0  -  True [[0.9975, 0.0025, 0.0]]
1 <=> 1  -  True [[0.0149, 0.9362, 0.0489]]
2 <=> 2  -  True [[0.0, 0.0673, 0.9327]]
1 <=> 1  -  True [[0.0082, 0.852, 0.1398]]
0 <=> 0  -  True [[0.9992, 0.0008, 0.0]]
0 <=> 0  -  True [[0.9948, 0.0052, 0.0]]
2 <=> 2  -  True [[0.0001, 0.1249, 0.875]]
0 <=> 0  -  True [[0.9948, 0.0052, 0.0]]
1 <=> 1  -  True [[0.0006, 0.9079, 0.0915]]
1 <=> 1  -  True [[0.0003, 0.6839, 0.3158]]
0 <=> 0  -  True [[0.9992, 0.0008, 0.0]]
2 <=> 2  -  True [[0.0, 0.0277, 0.9723]]
1 <=> 1  -  True [[0.0353, 0.9182, 0.0465]]
2 <=> 2  -  True [[0.0, 0.0257, 0.9743]]
0 <=> 0  -  True [[0.9999, 0.0001, 0.0]]
1 <=> 1  -  True [[0.0006, 0.6052, 0.3941]]
0 <=> 0  -  True [[0.9984, 0.0016, 0.0]]
1 <=> 1  -  True [[0.0001, 0.6143, 0.3855]]
0 <=> 0  -  True [[0.9994, 0.0006, 0.0]]
2 <=> 2  -  True [[0.0, 0.0125, 0.9875]]
1 <=> 1  -  True [[0.0113, 0.907, 0.0818]]
2 <=> 2  -  True [[0.0, 0.0633, 0.9367]]
0 <=> 0  -  True [[0.9928, 0.0072, 0.0]]
2 <=> 2  -  True [[0.0, 0.0307, 0.9693]]
0 <=> 0  -  True [[0.9684, 0.0316, 0.0]]
2 <=> 2  -  True [[0.0, 0.0174, 0.9826]]
2 <=> 2  -  True [[0.0, 0.0019, 0.9981]]
0 <=> 0  -  True [[0.9988, 0.0012, 0.0]]
1 <=> 1  -  True [[0.0137, 0.9375, 0.0488]]
1 <=> 1  -  True [[0.0021, 0.7884, 0.2095]]
0 <=> 0  -  True [[0.9998, 0.0002, 0.0]]
1 <=> 1  -  True [[0.0023, 0.9499, 0.0478]]
0 <=> 0  -  True [[0.9983, 0.0017, 0.0]]
2 <=> 2  -  True [[0.0, 0.0843, 0.9157]]
0 <=> 0  -  True [[0.9999, 0.0001, 0.0]]
0 <=> 0  -  True [[0.9992, 0.0008, 0.0]]
2 <=> 2  -  True [[0.0, 0.0163, 0.9837]]
2 <=> 2  -  True [[0.0001, 0.1279, 0.872]]
2 <=> 2  -  True [[0.0, 0.06, 0.94]]
Accuracy = 1.0
```

3.4 Method 2: Multi-class training

$$\Theta = (\theta^{(1)}, \ldots, \theta^{(c)})$$

$$h_\Theta(x) = [P(k|x, \Theta)]_{k=1,\ldots,c} = \text{softmax}(\Theta x)$$

3.4.1 The cost function (categorial cross-entropy)

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{c} \delta(y^{(i)}, k) \log P(k|x^{(i)}, \Theta)$$

$$\delta(x, y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$$

3.4.2 And its gradient

$$\frac{\partial J(\Theta)}{\partial \Theta_{j,k}} = -\frac{1}{m} \sum_{i=1}^{m} (\delta(y^{(i)}, k) - P(k|x^{(i)}, \Theta)) \, x_j^{(i)}$$

$$\nabla J(\Theta) = \frac{1}{m} X^T \left(h_\theta(X) - Y\right) \qquad Y - \text{Indicator matrix}$$

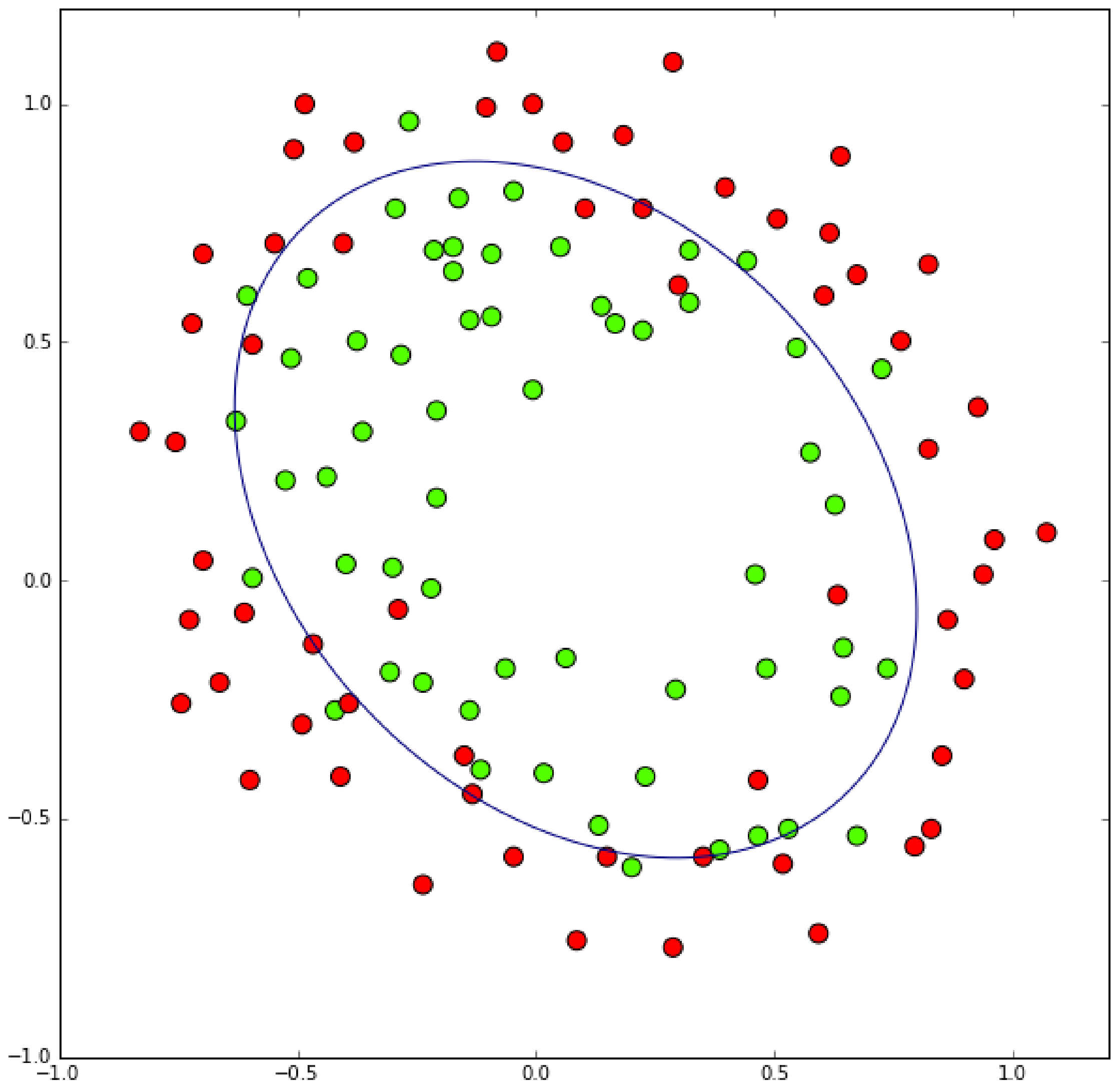
In [23]:
```
def h_mc(Theta, X):
    return softmax(X*Theta)

def J_mc(h, Theta, X, Y):
    return 0

def dJ_mc(h, Theta, X, Y):
    return 1.0/len(y) * (X.T*(h(Theta, X) - Y))

n = XTrain.shape[1]
k = YTrain.shape[1]

Theta = np.matrix(np.random.normal(0, 0.1, n*k)).reshape(n,k)
ThetaBest = mbSGD(h_mc, J_mc, dJ_mc, Theta, XTrain, YTrain,
                  alpha=0.01, maxSteps=50000, batchSize = 10)
print(ThetaBest)

def classify(Theta, X):
    probs = h_mc(Theta, X)
    return np.argmax(probs, axis=1), probs

YTestCls = YTest * np.matrix((0,1,2)).T

acc = 0.0
for i in range(len(YTestCls)):
    cls, probs = classify(ThetaBest, XTest[i])
    correct = int(YTestCls[i].item())
    print(correct, "<=>", cls, " - ", correct == cls, np.round(probs, 4).tolist())
    acc += correct == cls
print("Accuracy =", acc/float(len(XTest)))
```

```
[[ 0.06220749  0.19218432 -0.47822033]
 [ 0.47632816  0.20882749 -0.84773206]
 [ 1.13531687 -0.2500473  -1.0685813 ]
 [-1.67853691  0.25156471  1.55159505]
 [-0.83497163 -0.56058931  1.41419519]]
2 <=> [[2]]  -  [[ True]] [[0.0003, 0.2713, 0.7284]]
0 <=> [[0]]  -  [[ True]] [[0.9257, 0.0741, 0.0002]]
2 <=> [[2]]  -  [[ True]] [[0.0006, 0.2332, 0.7663]]
2 <=> [[2]]  -  [[ True]] [[0.0027, 0.3039, 0.6934]]
2 <=> [[2]]  -  [[ True]] [[0.0007, 0.3095, 0.6898]]
```

```
0 <=> [[0]]  -  [[ True]] [[0.9653, 0.0347, 0.0]]
1 <=> [[1]]  -  [[ True]] [[0.0344, 0.7488, 0.2168]]
1 <=> [[1]]  -  [[ True]] [[0.1798, 0.7252, 0.095]]
2 <=> [[2]]  -  [[ True]] [[0.0008, 0.2432, 0.756]]
2 <=> [[2]]  -  [[ True]] [[0.0011, 0.3871, 0.6118]]
0 <=> [[0]]  -  [[ True]] [[0.9363, 0.0636, 0.0001]]
0 <=> [[0]]  -  [[ True]] [[0.9413, 0.0586, 0.0002]]
1 <=> [[1]]  -  [[ True]] [[0.0535, 0.7927, 0.1538]]
2 <=> [[2]]  -  [[ True]] [[0.0004, 0.2425, 0.7571]]
1 <=> [[1]]  -  [[ True]] [[0.0338, 0.7137, 0.2525]]
0 <=> [[0]]  -  [[ True]] [[0.9653, 0.0347, 0.0]]
0 <=> [[0]]  -  [[ True]] [[0.9276, 0.0722, 0.0001]]
2 <=> [[2]]  -  [[ True]] [[0.0026, 0.3274, 0.67]]
0 <=> [[0]]  -  [[ True]] [[0.9276, 0.0722, 0.0001]]
1 <=> [[1]]  -  [[ True]] [[0.0116, 0.7071, 0.2812]]
1 <=> [[1]]  -  [[ True]] [[0.006, 0.5489, 0.4451]]
0 <=> [[0]]  -  [[ True]] [[0.9642, 0.0357, 0.0]]
2 <=> [[2]]  -  [[ True]] [[0.0004, 0.154, 0.8456]]
1 <=> [[1]]  -  [[ True]] [[0.0886, 0.7515, 0.16]]
2 <=> [[2]]  -  [[ True]] [[0.0007, 0.153, 0.8463]]
0 <=> [[0]]  -  [[ True]] [[0.9879, 0.0121, 0.0]]
1 <=> [[1]]  -  [[ True]] [[0.0079, 0.5814, 0.4107]]
0 <=> [[0]]  -  [[ True]] [[0.9535, 0.0464, 0.0001]]
1 <=> [[1]]  -  [[ True]] [[0.004, 0.5082, 0.4879]]
0 <=> [[0]]  -  [[ True]] [[0.9704, 0.0296, 0.0]]
2 <=> [[2]]  -  [[ True]] [[0.0, 0.0706, 0.9294]]
1 <=> [[1]]  -  [[ True]] [[0.0422, 0.7556, 0.2022]]
2 <=> [[2]]  -  [[ True]] [[0.0001, 0.172, 0.8279]]
0 <=> [[0]]  -  [[ True]] [[0.9091, 0.0906, 0.0003]]
2 <=> [[2]]  -  [[ True]] [[0.0016, 0.1911, 0.8072]]
0 <=> [[0]]  -  [[ True]] [[0.8405, 0.1584, 0.001]]
2 <=> [[2]]  -  [[ True]] [[0.0004, 0.1495, 0.8501]]
2 <=> [[2]]  -  [[ True]] [[0.0001, 0.0624, 0.9374]]
0 <=> [[0]]  -  [[ True]] [[0.9536, 0.0462, 0.0001]]
1 <=> [[1]]  -  [[ True]] [[0.053, 0.7838, 0.1633]]
1 <=> [[1]]  -  [[ True]] [[0.0182, 0.6332, 0.3486]]
0 <=> [[0]]  -  [[ True]] [[0.9823, 0.0177, 0.0]]
1 <=> [[1]]  -  [[ True]] [[0.0221, 0.7951, 0.1828]]
0 <=> [[0]]  -  [[ True]] [[0.9536, 0.0463, 0.0001]]
2 <=> [[2]]  -  [[ True]] [[0.0013, 0.2712, 0.7275]]
0 <=> [[0]]  -  [[ True]] [[0.99, 0.01, 0.0]]
0 <=> [[0]]  -  [[ True]] [[0.9643, 0.0356, 0.0001]]
2 <=> [[2]]  -  [[ True]] [[0.0003, 0.1311, 0.8686]]
2 <=> [[2]]  -  [[ True]] [[0.0023, 0.3379, 0.6598]]
2 <=> [[2]]  -  [[ True]] [[0.0012, 0.2482, 0.7506]]
Accuracy = [[ 1.]]
```

3.5 Feature engineering

In [24]:
```
n = 2
sgd = True
```



3.6 A more difficult example: The MNIST task

3.7 Reuse all the code

In [27]:
```
mnistXTrain, mnistYTrain = toMatrix(read(dataset="training"),
                                    maxItems=60000)
mnistYTrainI = indicatorMatrix(mnistYTrain)

mnistXTest, mnistYTest = toMatrix(read(dataset="testing"))
mnistYTestI = indicatorMatrix(mnistYTest)

n = mnistXTrain.shape[1]
k = mnistYTrainI.shape[1]

mnistTheta = np.matrix(np.random.normal(0, 0.1, n*k)).reshape(n,k)
mnistThetaBest = mbSGD(h_mc, J_mc, dJ_mc, mnistTheta, mnistXTrain, mnistYTrainI,
```

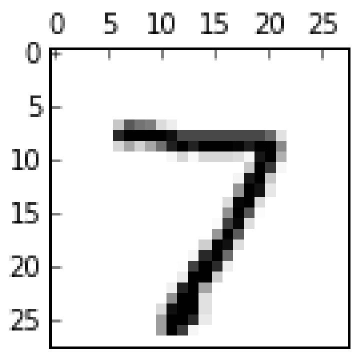
```
                                    alpha=0.1, maxSteps=20000, batchSize = 20)
```

```
In [28]: cls, probs = classify(mnistThetaBest, mnistXTest)
         print("Accuracy: ", np.sum(cls == mnistYTest)/len(mnistYTest))
```

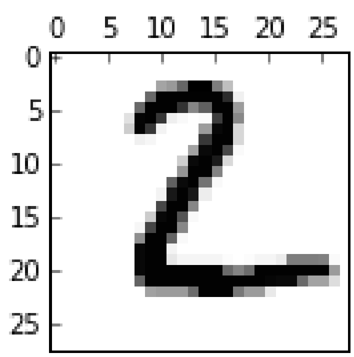
```
Accuracy:  0.905
```

3.8 Let's look at a few examples

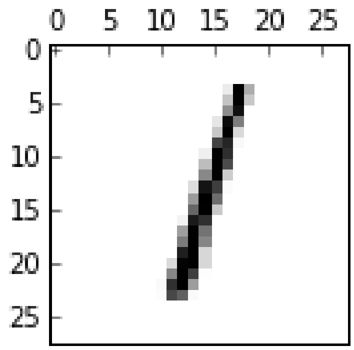
```
In [29]: np.set_printoptions(precision=2)
         for i, image in enumerate(mnistXTest[:10]):
             show(image[:,1:].reshape(28,28))
             print(cls[i], mnistYTest[i], probs[i])
```



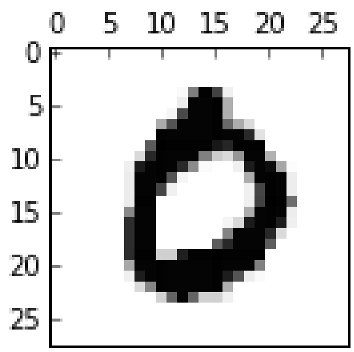

```
[[7]] [[ 7.]] [[  1.17e-04   1.45e-07   1.20e-04   1.90e-03   9.98e-06   4.82e-05
    2.72e-07   9.97e-01   6.82e-05   1.22e-03]]
```



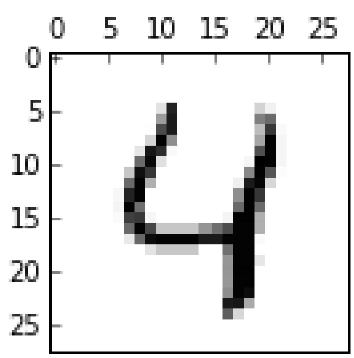

```
[[2]] [[ 2.]] [[  8.53e-03   4.78e-05   9.36e-01   9.52e-03   1.41e-08   1.29e-02
    2.98e-02   4.39e-09   3.32e-03   2.83e-07]]
```



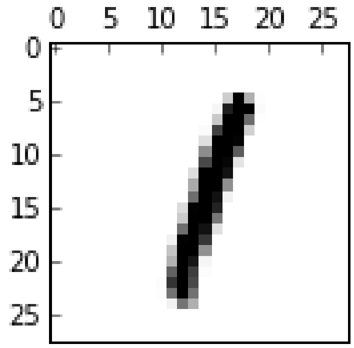

```
[[1]] [[ 1.]] [[  9.68e-05   9.55e-01   1.65e-02   6.18e-03   4.19e-04   2.43e-03
    3.97e-03   6.35e-03   8.27e-03   1.11e-03]]
```



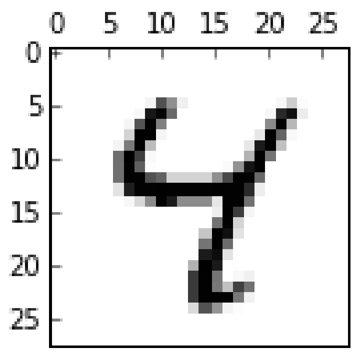

```
[[0]] [[ 0.]] [[  9.98e-01   2.60e-09   2.40e-04   3.34e-05   1.83e-07   7.01e-04
    4.49e-04   1.10e-04   1.20e-04   3.18e-05]]
```



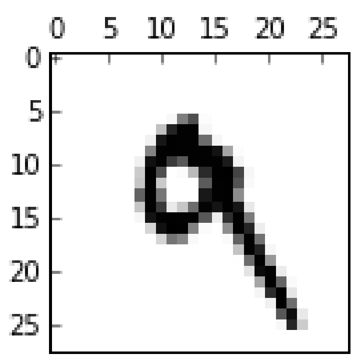

```
[[4]] [[ 4.]] [[  8.38e-04   2.12e-05   7.54e-03   2.44e-04   8.92e-01   6.17e-04
    6.37e-03   1.99e-02   9.52e-03   6.29e-02]]
```



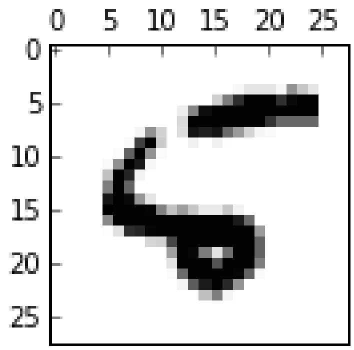
```
[[1]] [[ 1.]] [[  6.27e-06   9.84e-01   3.48e-03   3.37e-03   7.98e-05   2.37e-04
    1.31e-04   3.10e-03   5.24e-03   6.98e-04]]
```



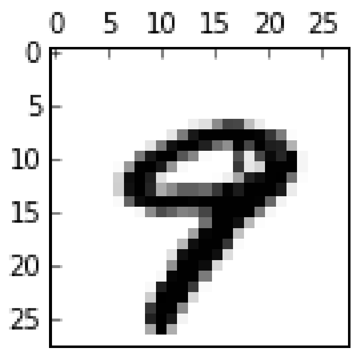
```
[[4]] [[ 4.]] [[  1.08e-05   1.90e-05   9.43e-06   8.96e-04   9.40e-01   1.18e-02
    2.69e-04   2.16e-03   2.94e-02   1.58e-02]]
```



```
[[9]] [[ 9.]] [[  8.97e-06   1.55e-03   1.96e-03   7.12e-03   2.56e-02   9.39e-03
    1.37e-03   6.14e-03   1.09e-02   9.36e-01]]
```




```
[[6]] [[ 5.]] [[  4.89e-03   5.73e-05   1.33e-02   4.94e-06   1.17e-02   7.67e-03
    9.58e-01   8.05e-06   3.87e-03   9.74e-04]]
```

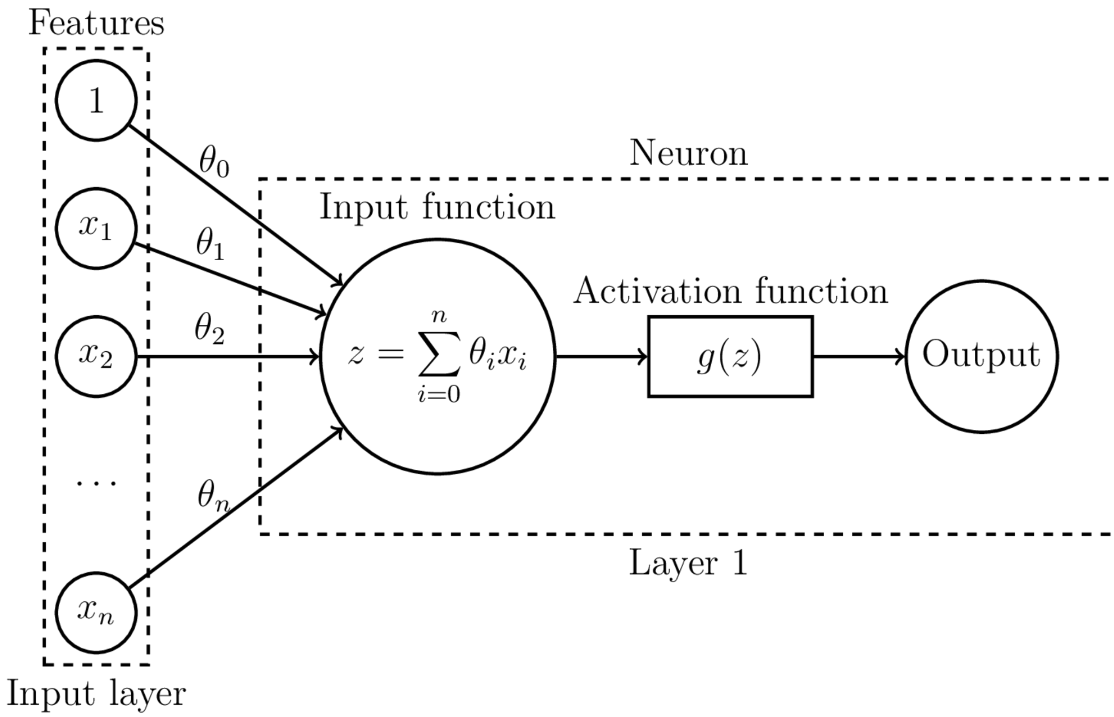



```
[[9]] [[ 9.]] [[  3.99e-05   2.81e-07   1.14e-05   1.82e-05   3.17e-02   2.37e-04
    5.44e-05   1.12e-01   8.63e-03   8.48e-01]]
```

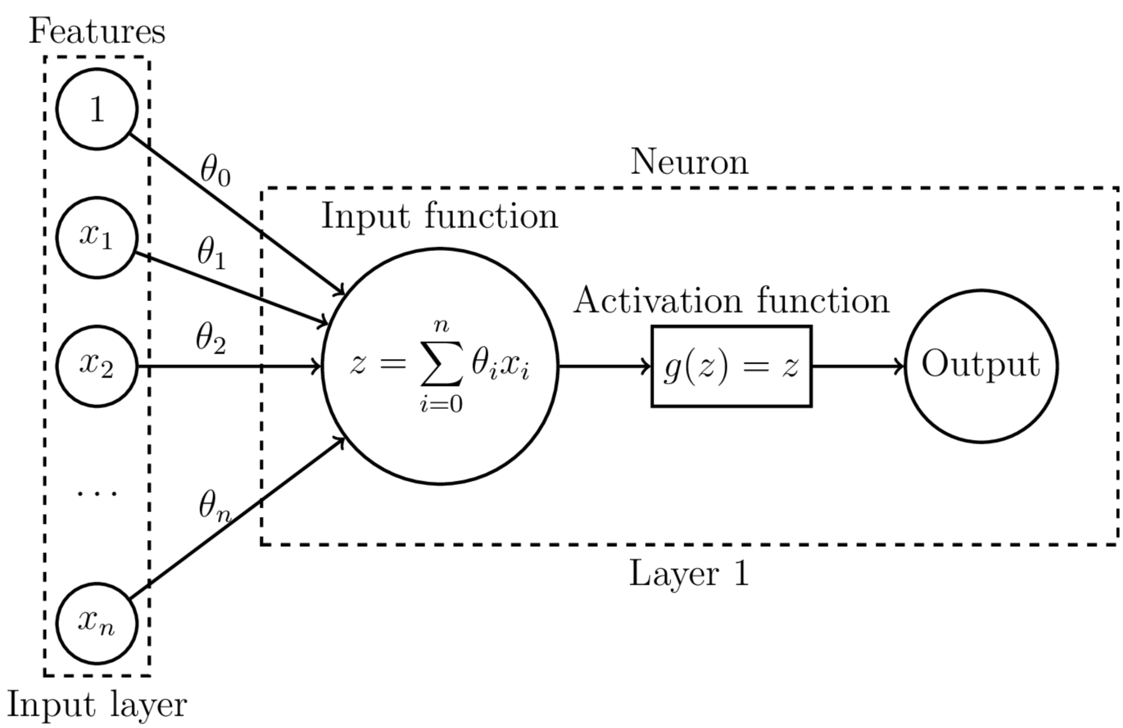
4 OK, but what about Neural Networks?

Actually, we already trained a whole bunch of neural networks during the lecture!

4.1 The Neuron



4.2 What's this?



4.3 All we need for a single-layer single-neuron regression network:

- Model:

$$h_\theta(x) = \sum_{i=0}^{n} \theta_i x_i$$

- Cost function (MSE):

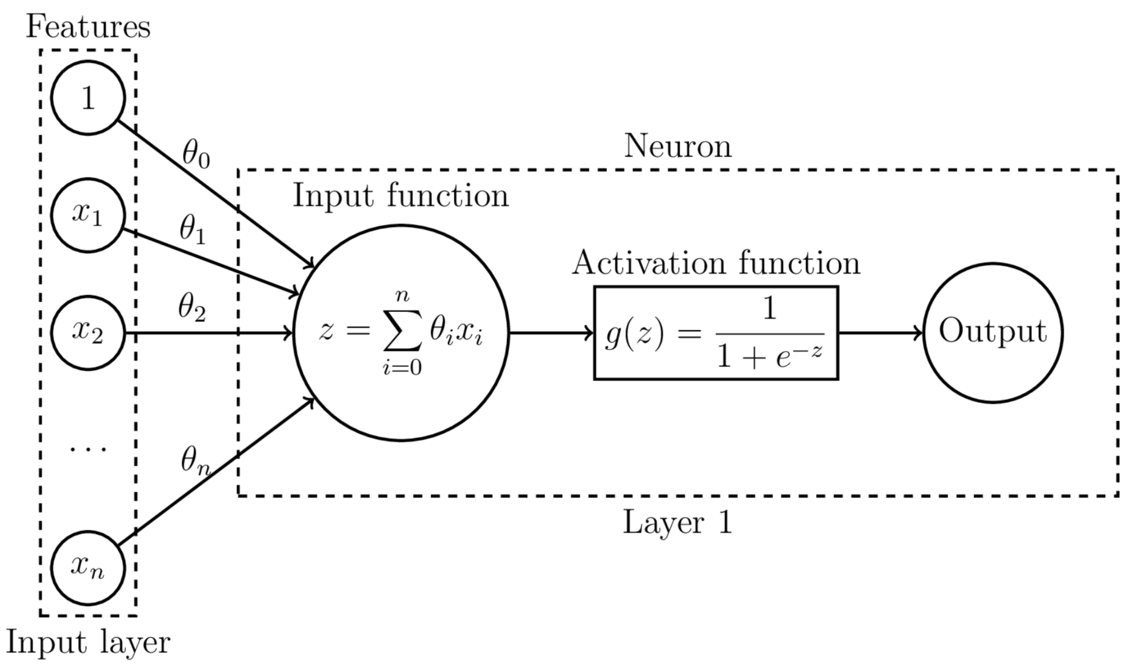
$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

- Gradient:

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

- Optimization algorithm: any variant of SGD

4.4 And that?



4.5 All we need for a single-layer single-neuron binary classifier:

- Model:

$$h_\theta(x) = \sigma(\sum_{i=0}^{n} \theta_i x_i) = P(c = 1|x, \theta)$$

- Cost function (binary cross-entropy):

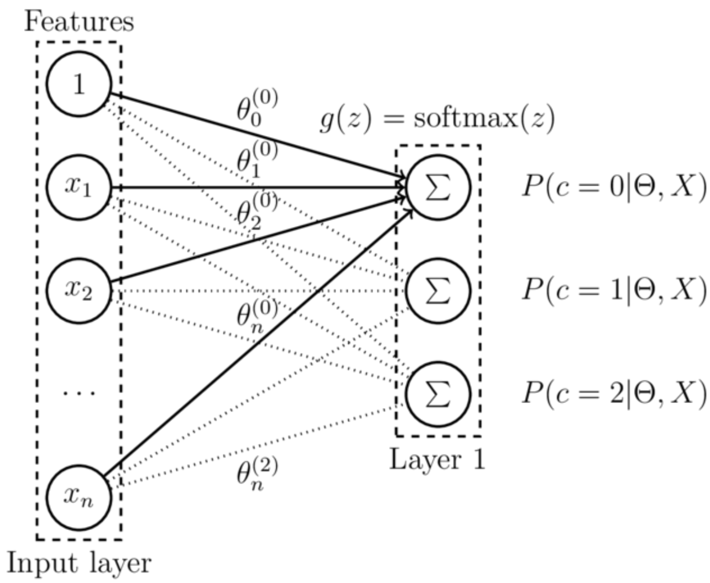
$$\begin{aligned} J(\theta) \quad = \quad & -\tfrac{1}{m} \textstyle\sum_{i=1}^{m} [y^{(i)} \log P(c = 1|x^{(i)}, \theta) \\ & +(1 - y^{(i)}) \log(1 - P(c = 1|x^{(i)}, \theta))] \end{aligned}$$

- Gradient:

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

- Optimization algorithm: any variant of SGD

4.6 And what are these?



$$\Theta = \begin{bmatrix} \theta_0^{(0)} & \theta_0^{(1)} & \theta_0^{(2)} \\ \theta_1^{(0)} & \theta_1^{(1)} & \theta_1^{(2)} \\ \vdots & \vdots & \vdots \\ \theta_n^{(0)} & \theta_n^{(1)} & \theta_n^{(2)} \end{bmatrix}$$

4.7 All we need to train a single-layer multi-class classifier

- Model:

$$h_\Theta(x) = [P(k|x, \Theta)]_{k=1,\ldots,c} \text{ where } \Theta = (\theta^{(1)}, \ldots, \theta^{(c)})$$

- Cost function $J(\Theta)$ (categorial cross-entropy):

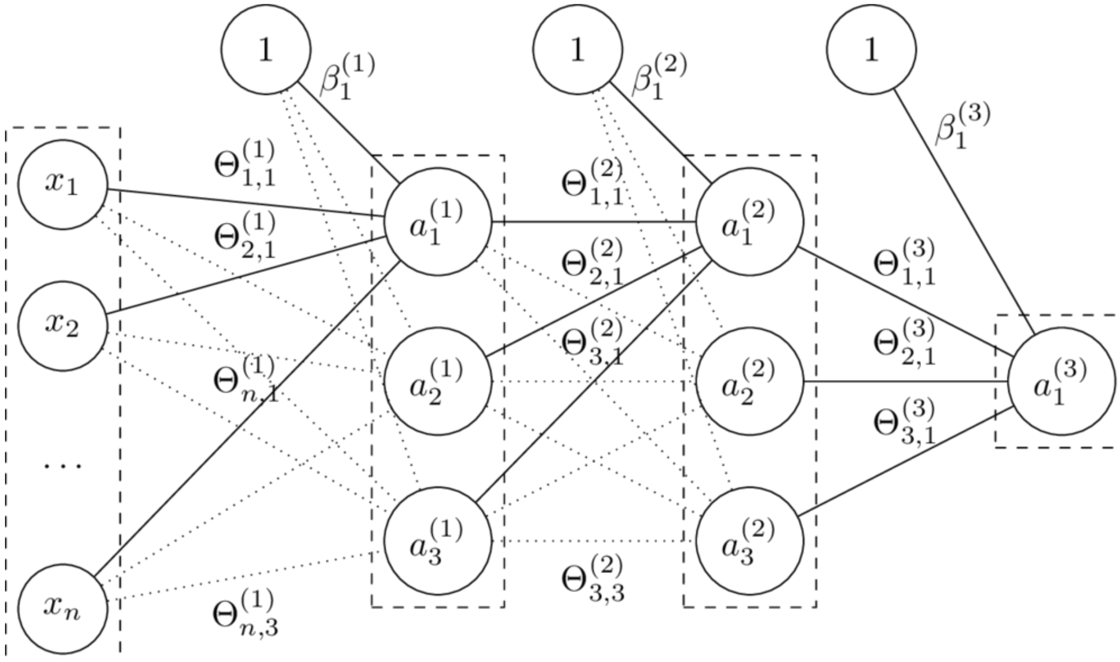
$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{c}\delta(y^{(i)}, k)\log P(k|x^{(i)}, \Theta)$$

- Gradient $\nabla J(\Theta)$:

$$\frac{\partial J(\Theta)}{\partial \Theta_{j,k}} = -\frac{1}{m}\sum_{i=1}^{m}(\delta(y^{(i)}, k) - P(k|x^{(i)}, \Theta))x_j^{(i)}$$

- Optimization algorithm: any variant of SGD

4.8 Tomorrow: How do we train this monster?



4.9 Or something like this: