Lecture 5

1. Event models
2. Neural networks

1 Event models

In the previous section we assumed that all x_i are binary, $x_i \in \{0, 1\}$. To generalize the previous model we assume that $x_i \in \{1, \ldots, k\}$, then the generative learning algorithm implies that we model

$$p(x \,|\, y) = \prod_{i=1}^{n} p(x_i \,|\, y),$$

where on the right-hand side we have multinomial probabilities (rather than Bernoulli as before). In case of continuous features this model can be also implemented if you apply the procedure of discretization:

$x_1 < 1$	$1 \leqslant x_1 < 10$	$10 \leqslant x_1 < 50$	$50 \leqslant x_1 < 100$	$100 \leqslant x_1$
0	1	2	3	4

The model with more than two possible values for each feature is called **multinomial event model** (compared to the multivariate Bernoulli event model we had earlier).

We consider the spam classification problem from different point of view. One email will be described by the vector $(x_1^{(i)}, x_2^{(i)}, \ldots, x_{d_i}^{(i)})$, where d_i is a number of words in the email i and $x_j \in \{1, 2, \ldots, n\}$, where n is a size of the bug of words and could be a huge number, for example, $n = 50\,000$. In other words, we assign some number to each word and takes these numbers to the feature vector. The main difference in such formulation is that feature vectors have different lengths for different training examples (because email lengths are different).

The joint distribution for x and y can be written as

$$p(x, y) = \left(\prod_{j=1}^{d} p(x_j \,|\, y) \right) p(y),$$

where x_j is j-th word in the email, d is a number of words in the email. We identify the parameters for this model and apply the maximum likelihood estimation to find the values for them. Parameters are defined by the equations:

$$\begin{aligned}
&\varphi_{k \,|\, y=1} = p(x_j = k \,|\, y = 1) \text{ (for any } j),\\
&\varphi_{k \,|\, y=0} = p(x_j = k \,|\, y = 0) \text{ (for any } j),\\
&\varphi_y = p(y = 1).
\end{aligned}$$

For example, the parameter $\varphi_{k \,|\, y=1}$ defines the probability of having word k in the spam email on any position j and so on.

The likelihood is defined as

$$l(\varphi_{k\,|\,y=1}, \varphi_{k\,|\,y=0}, \varphi_y) = \ln \prod_{i=1}^{m} p(x^{(i)}, y^{(i)}, \varphi_{k\,|\,y=1}, \varphi_{k\,|\,y=0}, \varphi_y) =$$
$$= \ln \prod_{i=1}^{m} p(x^{(i)} \,|\, y^{(i)}, \varphi_{k\,|\,y=1}, \varphi_{k\,|\,y=0}) \cdot p(y^{(i)}; \varphi_y)$$

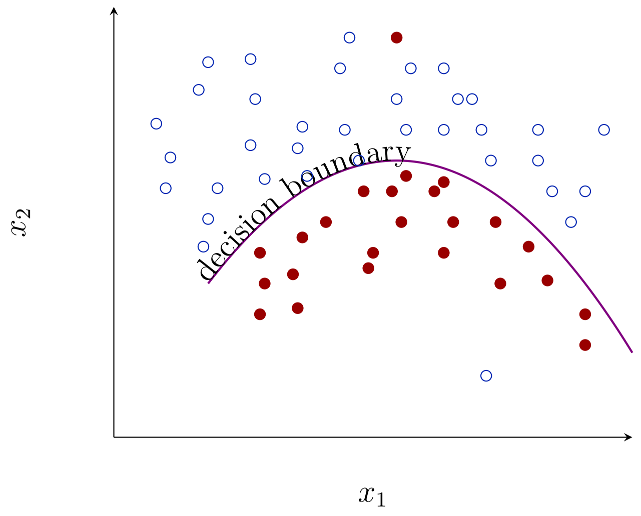
and the maximization gives the following list of parameters

$$\varphi_{k\,|\,y=1} = \frac{\sum_{i=1}^{m} \left(\mathbb{1}\{y^{(i)} = 1\} \sum_{j=1}^{d_i} \mathbb{1}\{x_j^{(i)} = k\} \right) + 1}{\sum_{i=1}^{m} \left(\mathbb{1}\{y^{(i)} = 1\} \cdot d_i \right) + n},$$
$$\varphi_{k\,|\,y=0} = \frac{\sum_{i=1}^{m} \left(\mathbb{1}\{y^{(i)} = 0\} \sum_{j=1}^{d_i} \mathbb{1}\{x_j^{(i)} = k\} \right) + 1}{\sum_{i=1}^{m} \left(\mathbb{1}\{y^{(i)} = 0\} \cdot d_i \right) + n}.$$

The meaning of the first parameter is the number of appearances of the word k in all spam emails divided by the total number of words in all spam emails. Notice that we use the general Laplace smoothing described in the previous lecture.

2 Neural networks

The following example shows that in some cases we need non linear decision boundary which means that we should build nonlinear classifiers.



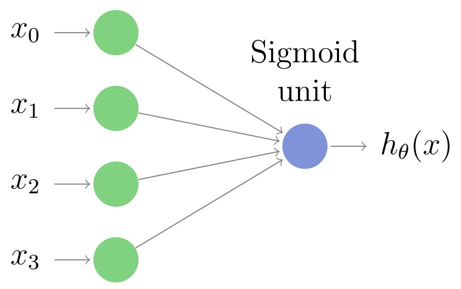
One example of nonlinear classifiers we took before was the logistic regression with hypothesis $h_\theta(x) = \dfrac{1}{1 + e^{-\theta^T x}}$. Another example is generative learning algorithm with the assumptions

$$x \,|\, y = 1 \sim \text{ExpFamily}(\eta_1),$$
$$x \,|\, y = 0 \sim \text{ExpFamily}(\eta_0).$$

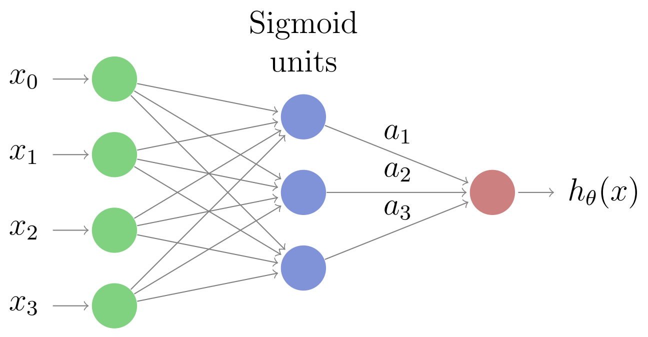
One way to build nonlinear classifier is to fit the hypothesis with higher degree function instead of $\theta^T x$. For example, we can consider the hypothesis

$$h_\theta(x) = \frac{1}{1 + e^{-\theta_0 - \theta_1 x_1 - \theta_2 x_2 - \theta_3 x_1 x_2 - \theta_4 x_1^2 x_2 - \dots}}.$$

Another way is to visualize the logistic regression algorithm in some fancy way and try to generalize the idea:



For this diagram we are going to use the biological terminology: the circles are **neurons** or **units**, the set of green circles (representing the features) is an **input layer**, the blues circle with sigmoid function is an **output layer** with **sigmoid activation function**. The natural way to generalize this diagram is to add more neurons:



On this diagram we have 3 layers: input layer (green), hidden layer (blue) with sigmoid activation functions and output layer (red) with sigmoid activation function. Let $g(z) = \dfrac{1}{1+e^{-z}}$ be a sigmoid function then

$$a_1 = g(x^T\theta_1^{(1)}),\ a_2 = g(x^T\theta_2^{(1)}),\ a_3 = g(x^T\theta_3^{(1)}),\ h_\theta(x) = g(a^T\theta^{(2)}),$$

where

$$a = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}.$$

The number of parameters (**weights** in the neural network terminology) for this network equals to the number of edges for this diagram: $12+3 = 15$. In other words, we should learn 4 vectors of parameters $\theta_1^{(1)}$, $\theta_2^{(1)}$, $\theta_3^{(1)}$ and $\theta^{(2)}$. Obviously we could add more hidden layers and our hypothesis becomes more complicated.

To fit the parameters we recall that the log-likelihood for logistic regression has a form

$$l(\theta) = \sum_{i=1}^{m} y^{(i)} \ln h_\theta(x^{(i)}) + (1 - y^{(i)}) \ln(1 - h_\theta(x^{(i)}))$$

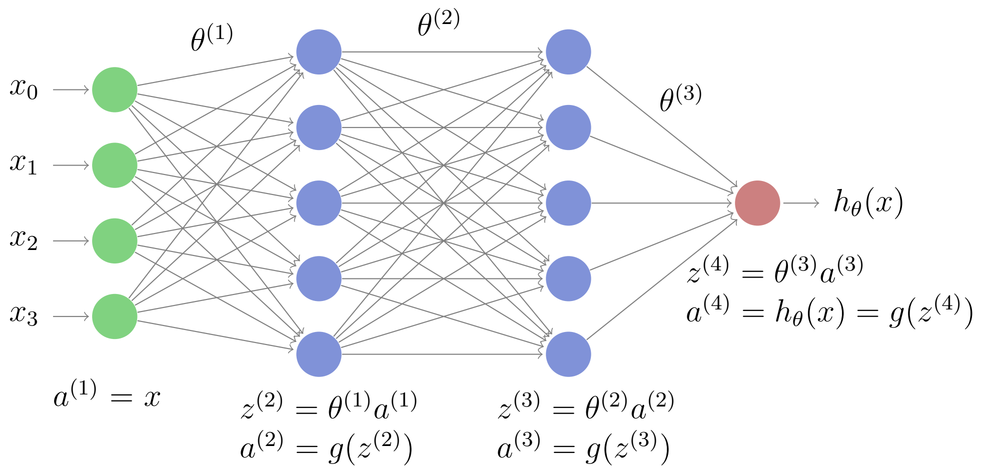
and we obtain the neural network as a generalization of logistic regression. This is the reason that for the neural network we are going to use the following loss function

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

and find the minimum of this function using the gradient descent. For the neural networks the gradient descent algorithm has a special name: **backpropagation**. The procedure of predictions in the neural network is called **forward propagation**.

To have the convenient notations we denote by $\theta^{(k)}$ the matrix of parameters from the layer k to the layer $k+1$. The element $\theta_{ij}^{(k)}$ defines the weight from the i-th neuron of layer $k+1$ to the j-th neuron of layer k. Let $a_i^{(k)}$ be the activation function of the neuron i in layer k. To use the gradient descent algorithm we should compute all derivatives $\dfrac{\partial J(\theta)}{\partial \theta_{ij}^{(k)}}$.

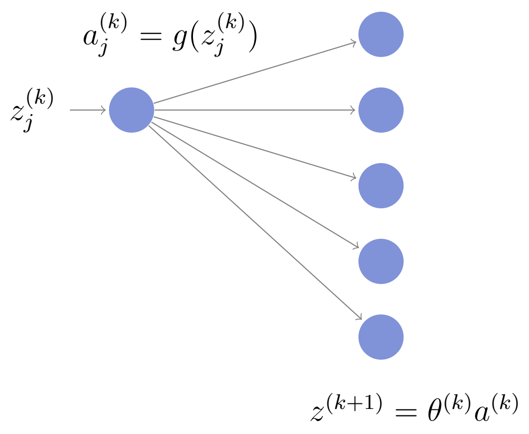
Given one training example (x, y) consider one forward propagation step for the following network configuration:


$\theta^{(1)}$
$\theta^{(2)}$
$\theta^{(3)}$
x_0
x_1
x_2
x_3
$h_\theta(x)$
$z^{(4)} = \theta^{(3)} a^{(3)}$
$a^{(4)} = h_\theta(x) = g(z^{(4)})$
$a^{(1)} = x$
$z^{(2)} = \theta^{(1)} a^{(1)}$
$a^{(2)} = g(z^{(2)})$
$z^{(3)} = \theta^{(2)} a^{(2)}$
$a^{(3)} = g(z^{(3)})$

The backpropagation procedure includes the calculation of partial derivatives

$$\delta_j^{(k)} = \frac{\partial J(\theta)}{\partial z_j^{(k)}}.$$

Consider the unit j in the layer k and the next layer $k+1$:


$a_j^{(k)} = g(z_j^{(k)})$
$z_j^{(k)}$
$z^{(k+1)} = \theta^{(k)} a^{(k)}$

Assuming that we know derivatives $\delta_i^{(k+1)} = \dfrac{\partial J(\theta)}{\partial z_i^{(k+1)}}$ and

$$z_i^{(k+1)} = \ldots + \theta_{ij}^{(k)} a_j^{(k)} + \ldots = \ldots + \theta_{ij}^{(k)} g(z_j^{(k)}) + \ldots, \tag{1}$$

where terms from other units of layer k are denoted by dots. Then the derivative $\delta_j^{(k)}$ can be calculated using Chain Rule:

$$\begin{aligned} \delta_j^{(k)} &= \frac{\partial J(\theta)}{\partial z_j^{(k)}} = \sum_i \frac{\partial J(\theta)}{\partial z_i^{(k+1)}} \cdot \frac{\partial z_i^{(k+1)}}{\partial z_j^{(k)}} = \\ &= \sum_i \frac{\partial J(\theta)}{\partial z_i^{(k+1)}} \cdot \theta_{ij}^{(k)} \cdot g'(z_j^{(k)}) = \sum_i \delta_i^{(k+1)} \theta_{ij}^{(k)} \cdot g'(z_j^{(k)}). \end{aligned}$$

More general, for all units of the layer k the formula can be written as

$$\delta^{(k)} = (\theta^{(k)})^T \delta^{(k+1)} g'(z^{(k)}) = (\theta^{(k)})^T \delta^{(k+1)} a^{(k)} (1 - a^{(k)})$$

(remember that $g(z)$ is a sigmoid function and its derivatives can be calculated in a very simple way).

When all "deltas" are calculated it is easy to calculate our target derivatives (again using Chain Rule and the equation (1)):

$$\frac{\partial J(\theta)}{\partial \theta_{ij}^{(k)}} = \frac{\partial J(\theta)}{\partial z_i^{(k+1)}} \cdot \frac{\partial z_i^{(k+1)}}{\partial \theta_{ij}^{(k)}} = \delta_i^{(k+1)} \cdot g(z_j^{(k)}) = \delta_i^{(k+1)} \cdot a_j^{(k)}.$$

Now we can summarize the backpropagation algorithm (we are using stochastic gradient descent to update weights):

Algorithm 1 Backpropagation algorithm for neural networks

```
1: Set number of hidden layers $K$ and number of neurons $m_k$ in the layer $k$
2: Initialize matrices $\theta^{(k)}, k = 0, \dots, K$ of random weights
3: repeat
4:     for all training examples $(x, y)$ do
5:         Set $a^{(0)} = x$
6:         Run the forward step to compute $a^{(k)}, k = 1, \dots, K+1$
7:         Set $\delta^{(K+1)} = a^{(K+1)} - y$ (here $a^{(K+1)}$ is a prediction)
8:         for $k = K$ downto 1 do
9:             Set $\delta^{(k)} = (\theta^{(k)})^T \delta^{(k+1)} a^{(k)} (1 - a^{(k)})$
10:            for all $i = 1, \dots, m_{k+1}$ and $j = 1, \dots, m_k$ do
11:                Perform the stochastic gradient descent step
                       $$\theta_{ij}^{(k)} := \theta_{ij}^{(k)} - \alpha \cdot \delta_i^{(k+1)} \cdot a_j^{(k)}$$
12: until convergence
```

2.1 Python implementation

Import necessary libraries:

```
In [1]: import numpy as np
        import random
        import math
        import sklearn.datasets as ds
        from matplotlib import cm
        import matplotlib.pyplot as plt

        from nnvisual import *
```

We have implemented the following functions: `sigmoid()` is an activation function, `forward()` implements forward propagation step, `backward()` implements backpropagation step. The function `plot_decision_boundary()` draws the decision boundary for two classes. The function `train_neural_network()` runs the main loop of forward and backward propagations, and the function `predict_neural_network()` runs the forward loop to obtain the predictions. Notice that we have added the regularization term, which means that instead of the loss function (we will talk about the regularization techniques later in our lectures):

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} \ln h_\theta(x^{(i)}) + (1 - y^{(i)}) \ln(1 - h_\theta(x^{(i)})) \right)$$

we minimize the function

$$J(\theta) = -\frac{1}{m} \left(\sum_{i=1}^{m} y^{(i)} \ln h_\theta(x^{(i)}) + (1 - y^{(i)}) \ln(1 - h_\theta(x^{(i)})) \right) + \frac{\beta}{2} \sum_{k=0}^{K} \sum_{i=1}^{m_{k+1}} \sum_{j=1}^{m_k} \left(\theta_{ij}^{(k)} \right)^2.$$

```
In [2]: def sigmoid(z):
            return 1.0/(1+np.exp(-z))

        def forward(X, thetas):
            Xb = np.hstack((np.ones((X.shape[0],1)), X))
            a = np.transpose(Xb)
            alist = [a]
            for i in range(L):
                a = sigmoid(np.dot(thetas[i], a))
                if i < L-1:
                    a = np.vstack((np.ones((1, a.shape[1])), a))
                alist.append(a)
            return alist

        def backward(a, y, thetas, alpha=0.1, beta=0.01, batch_size=1):
            start = 0
            end = start + batch_size
            while True:
                if end > y.shape[1]:
                    end = y.shape[1]
                # a[L] are predictions
                deltas = a[L][:,start:end] - y[:,start:end]
                for i in range(L-1, -1, -1):
                    deltas_prev = np.multiply(
                        np.multiply(
                            np.dot(thetas[i].T, deltas),
                            a[i][:,start:end]),
                        1-a[i][:,start:end])
                    step = np.sum(np.array([
                              np.outer(deltas[:,k], a[i][:,start:end][:,k])
                              for k in range(deltas.shape[1])
                              ]), axis=0)
                    thetas[i][:,1:] = thetas[i][:,1:] - alpha*step[:,1:]
                                      - beta*thetas[i][:,1:]
                    thetas[i][:,0] = thetas[i][:,0] - alpha*step[:,0]
                    deltas = deltas_prev[1:, :]
                start = end
                end = start + batch_size
                if start == y.shape[1]:
                    break
            return thetas
```

```
def plot_decision_boundary(X, y, thetas, fig, pos,
                           xmin=-0.1, xmax=1.1, ymin=-0.1, ymax=1.1):
    x1, x2 = np.meshgrid(np.linspace(xmin, xmax, 200),
                         np.linspace(ymin, ymax, 200))
    ypred = forward(np.c_[x1.ravel(), x2.ravel()], thetas)[L][0]
    ypred = ypred.reshape(x1.shape)
    extent = xmin, xmax, ymin, ymax

    fig.add_subplot(1,4,pos)
    plt.imshow(ypred, cmap=cm.bwr, alpha=.9,
               interpolation='bilinear', extent = extent,
               origin='lower')
    plt.scatter(X[:,0], X[:,1], s=40, c=y, cmap=cm.bwr)

def train_neural_network(X, y, alpha, beta, batch_size, seed, niter):
    ### initialize weights
    thetas = []
    np.random.seed(seed)
    for i in range(L):
        thetas.append(20*(np.random.rand(nn[i+1], nn[i]+1)-0.5))
    print 'Thetas:'
    print thetas

    ### the main loop of training
    fig = plt.figure(figsize=(16,32))
    count_plot = 0
    for it in range(niter):
        a = forward(X, thetas)
        thetas = backward(a, y, thetas, alpha=alpha, beta=beta,
                          batch_size=batch_size)
        if it % round(niter/4) == 0:
            count_plot = count_plot + 1
            plot_decision_boundary(X, y, thetas, fig, count_plot,
                                   xmin=np.min(X[:,0])-0.1,
                                   xmax=np.max(X[:,0])+0.1,
                                   ymin=np.min(X[:,1])-0.1,
                                   ymax=np.max(X[:,1])+0.1)
    plt.show()
    return thetas

def predict_neural_network(X, thetas):
    return forward(X, thetas)[L][0]
```

We use our neural network to solve the famous XOR problem.

$\mathbf{x_1}$	$\mathbf{x_2}$	XOR $(\mathbf{x_1}, \mathbf{x_2})$
1	1	1
1	0	0
0	1	0
0	0	1

The XOR function detects if x_1 and x_2 are the same. The known fact is that it is impossible to solve this problem using logistic regression and we need at least one hidden layer to get correct solution. First, we define our dataset: 4 training examples with XOR as a target variable y.

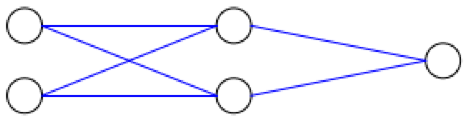
```
In [3]: X_train = np.array([[1, 1], [1, 0], [0, 1], [0, 0]]).astype(np.float64)
        y_train = np.array([1, 0, 0, 1]).reshape((1, -1))
```

Define the neural network configuration (the last number is the dimension of y):

```
In [4]: nn = np.array([X_train.shape[1], 2, y_train.shape[0]]).astype(int)
        L = len(nn)-1 # L = number of hidden layers + 1
```

We are using nnvisual.py code to show how my neural network looks like. This code is modified code from here: https://github.com/miloharper/visualise-neural-network (thanks to miloharper for it). I have changed the orientation of the neural network in the original file.

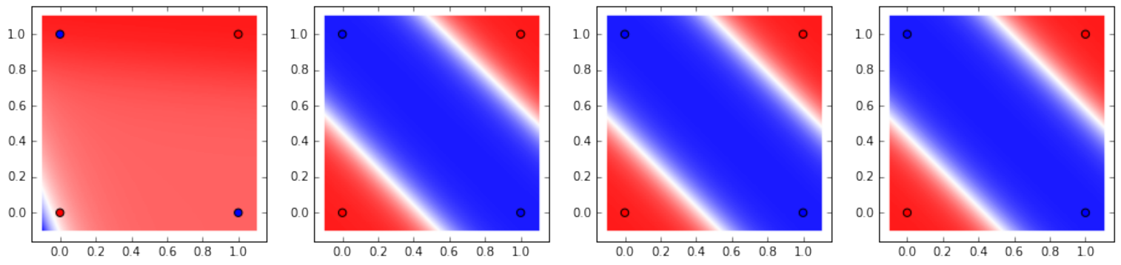
```
In [5]: network = NeuralNetwork()
        for l in nn:
            network.add_layer(l)
        network.draw()
```



Now we check different initializations and different parameters:

```
In [6]: thetas = train_neural_network(X_train, y_train,
                                      alpha=0.1, beta=0.0001,
                                      batch_size=2, seed=131214, niter=10000)
        print "Predictions:"
        predict_neural_network(X_train, thetas)
```

```
Thetas:
[array([[1.67236953, 7.19523078, 2.85081229],
       [-8.56741528, 0.63708182, 8.25788355]]),
 array([[5.62926985, 5.52618743, 5.01061296]])]
```



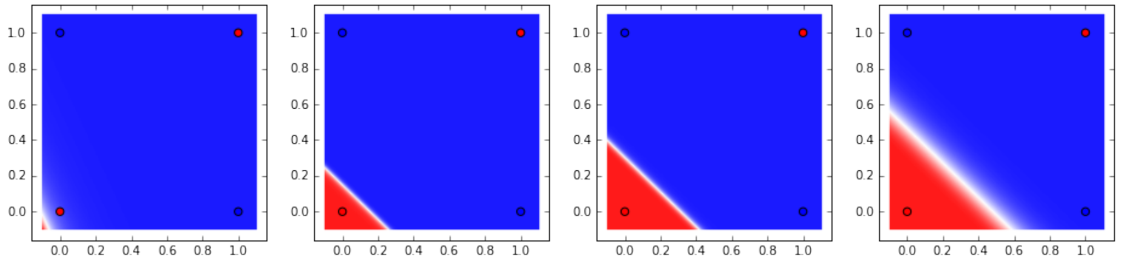
```
Predictions:

Out[6]: array([0.97663021, 0.02344909, 0.0234481, 0.97639398])
```

Here the influence of the learning rate alpha. We can see that our neural network cannot find the correct minimum with big learning rate. Notice that in this case all predictions are very close to each other.

```
In [7]: thetas = train_neural_network(X_train, y_train,
                                      alpha=10.0, beta=0.0001,
                                      batch_size=2, seed=131214, niter=10000)
        print "Predictions:"
        predict_neural_network(X_train, thetas)
```

```
Thetas:
[array([[1.67236953, 7.19523078, 2.85081229],
       [-8.56741528, 0.63708182, 8.25788355]]),
 array([[5.62926985, 5.52618743, 5.01061296]])]
```



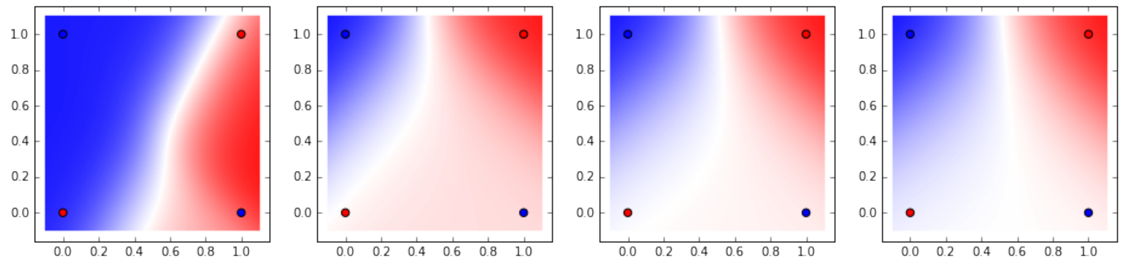
```
Predictions:

Out[7]: array([0.999996, 0.999996, 0.999996, 1.])
```

The initialization weights are also very important. The different seed can give different results:

```
In [8]: thetas = train_neural_network(X_train, y_train,
                                      alpha=0.01, beta=0.0001,
                                      batch_size=2, seed=114, niter=10000)
        print "Predictions:"
        predict_neural_network(X_train, thetas)
```

```
Thetas:
[array([[-6.96736066, 7.37490561, 8.51767526],
       [-0.23173645, 5.30985877, -2.76355142]]),
 array([[-7.335906, 0.43388152, 4.42122827]])]
```



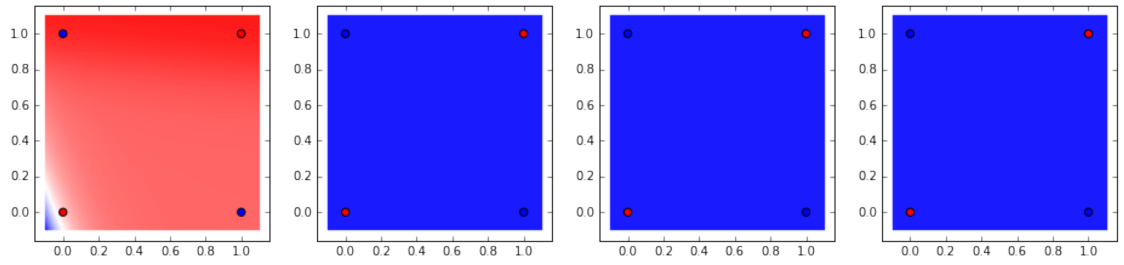
```
Predictions:

Out[8]: array([0.8485083, 0.52529168, 0.15702184, 0.46377799])
```

The regularization parameter beta is also important. For wrong values of beta the neural network can fail as well:

```
In [9]: thetas = train_neural_network(X_train, y_train,
                                      alpha=0.1, beta=0.01,
                                      batch_size=2, seed=131214, niter=10000)
        print "Predictions:"
        predict_neural_network(X_train, thetas)
```

```
Thetas:
[array([[1.67236953, 7.19523078, 2.85081229],
       [-8.56741528, 0.63708182, 8.25788355]]),
 array([[5.62926985, 5.52618743, 5.01061296]])]
```

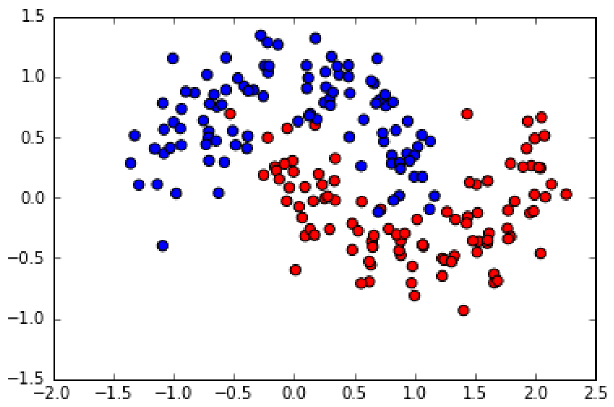


```
Predictions:

Out[9]: array([0.5, 0.5, 0.5, 0.5])
```

As an another example we consider the `make_moons` dataset from `scikit-learn` package:

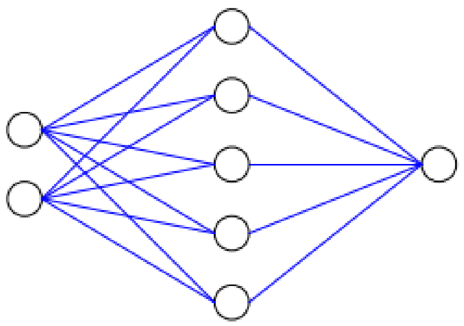
```
In [10]: np.random.seed(0)
         X_train, y_train = ds.make_moons(200, noise=0.20)
         y_train = y_train.reshape((1, -1))
         plt.scatter(X_train[:,0], X_train[:,1], s=40, c=y_train, cmap=cm.bwr)
         plt.show()
```



Initialize our neural network and visualize it:

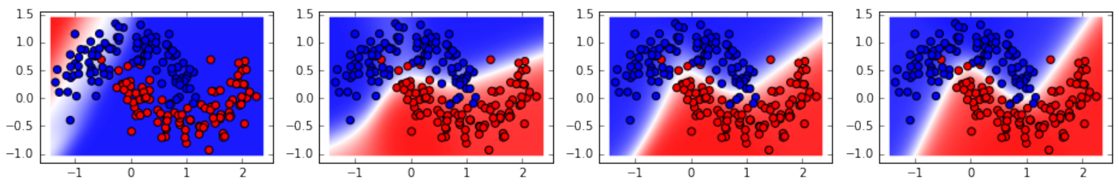
```
In [11]: nn = np.array([X_train.shape[1], 5, y_train.shape[0]]).astype(int)
         L = len(nn)-1 # L = number of hidden layers + 1
```

```
In [12]: network = NeuralNetwork()
         for l in nn:
             network.add_layer(l)
         network.draw()
```



```
In [13]: thetas = train_neural_network(X_train, y_train,
                                       alpha=0.01, beta=0.0001,
                                       batch_size=10, seed=1104, niter=1000)
```

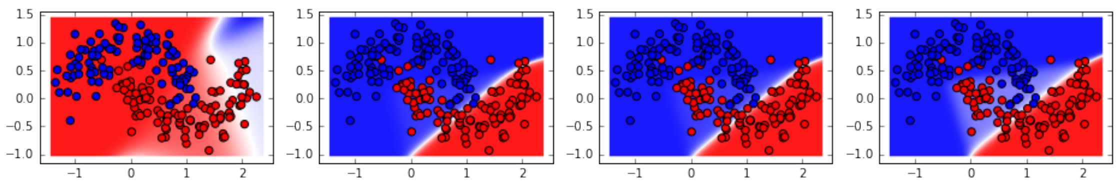
```
Thetas:
[array([[1.22741958, -1.53385074, 4.2008582 ],
       [-3.76905412, 6.57801497, 1.09413825],
       [ 0.05969385, -3.51291599, 6.83412737],
       [-8.07433437, -3.75662311, 2.07674147],
       [-1.92150795, -4.73883283, 2.7290175]]),
 array([[-9.78380152, 1.20918211, -8.0590909, -2.00016833, 9.60846917,
         9.05818477]])]
```



Wrong learning rate:

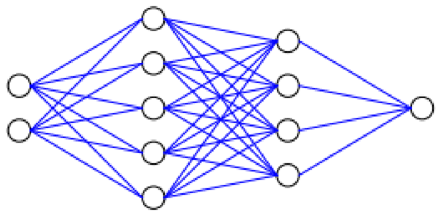
```
In [14]: thetas = train_neural_network(X_train, y_train,
                                       alpha=0.1, beta=0.0001,
                                       batch_size=10, seed=1104, niter=1000)
```

```
Thetas:
[array([[1.22741958, -1.53385074, 4.2008582 ],
       [-3.76905412, 6.57801497, 1.09413825],
       [ 0.05969385, -3.51291599, 6.83412737],
       [-8.07433437, -3.75662311, 2.07674147],
       [-1.92150795, -4.73883283, 2.7290175]]),
 array([[-9.78380152, 1.20918211, -8.0590909, -2.00016833, 9.60846917,
         9.05818477]])]
```



Add more layers to the network:

```
In [15]: nn = np.array([X_train.shape[1], 5, 4, y_train.shape[0]]).astype(int)
         L = len(nn)-1 # L = number of hidden layers + 1
         network = NeuralNetwork()
         for l in nn:
             network.add_layer(l)
         network.draw()
```



```
In [16]: thetas = train_neural_network(X_train, y_train,
                                       alpha=0.1, beta=0.0001,
                                       batch_size=10, seed=1104, niter=1000)

Thetas:
[array([[1.2274, -1.5338, 4.2008],
       [-3.7690, 6.5780, 1.0941],
       [ 0.0596, -3.5129,  6.8341],
       [-8.0743, -3.7566, 2.0767],
       [-1.9215, -4.7388, 2.7290]]),
 array([[-9.7838, 1.2091, -8.0590, -2.0001, 9.6084, 9.0581],
        [2.7658, 6.1862, 5.8740, 3.1083, 4.1896, -9.6227],
        [7.9800, -5.7013, -2.8169, 1.3702, 1.6314, 2.6471],
        [0.9133, 5.7996, -6.8791, 3.4554, -4.2520, 8.1307]]),
 array([[1.5958, -6.1230, -9.7194, -6.3800, 1.2106]])]
```