Introduction to Theano

In [ ]:
```
Tomasz Golan
MLVF Meeting, 03.16.2016
```

Linear Regression

Notation

- Features:

$$x = (x_1, \ldots, x_n)$$

- Training example:

$$(x^{(i)}, y^{(i)})$$

- Training set:

$$\left\{(x^{(i)}, y^{(i)}); i = 1, \ldots, N\right\}$$

Notation

- Weights:

$$w = (w_0, w_1, \ldots, w_n)$$

- For convenience:

$$x_0 = 1$$

- Hypothesis:

$$h(x) = w_0 + w_1 x_1 + \ldots + w_n x_n = \sum_{i=0}^{n} w_i x_i = w^T x$$

Learning

- Learning means finding w
- Lets define cost function as:

$$f(w) = \frac{1}{2} \sum_{i=1}^{N} \left(h(x^{(i)}) - y^{(i)} \right)^2$$

- Minimum can be find using gradient descent method:

$$w_j = w_j - \alpha \frac{\partial f(w)}{\partial w_j} = w_j + \alpha \sum_{i=1}^{N} \left(y^{(i)} - h(x^{(i)}) \right) x_j$$

- Where α is learning rate

Generate Samples

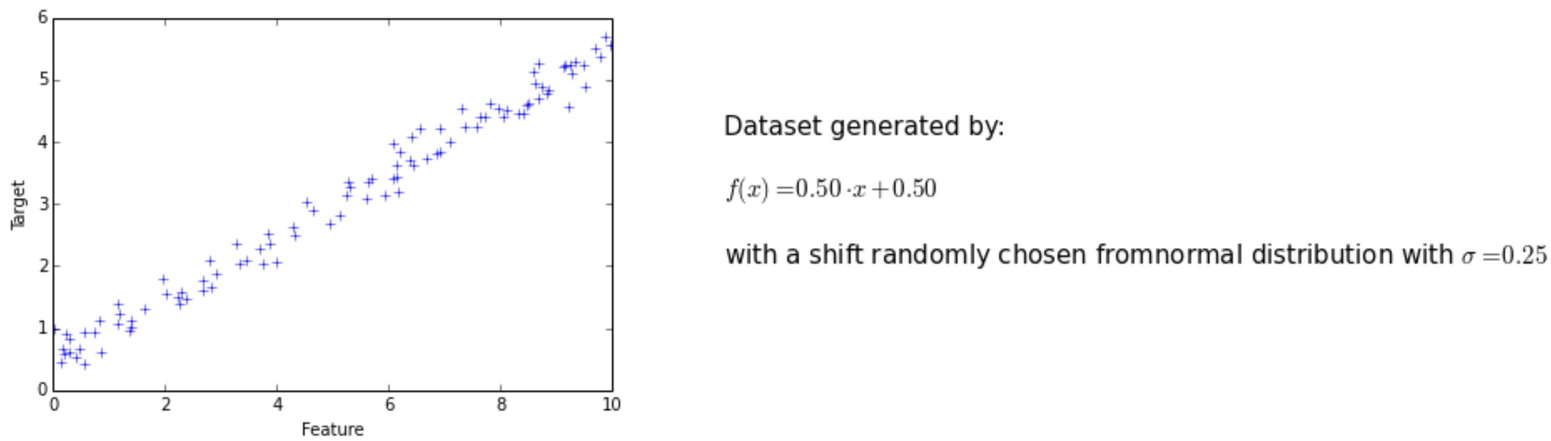
In [2]:

```
%matplotlib inline

### IMPORTS ###

import numpy
import matplotlib.pyplot as plt

rng = numpy.random # random number generator

### SETTINGS ###

N = 100 # number of samples

a = 0.50 # slope
b = 0.50 # y-intercept
s = 0.25 # sigma

### GENERATE SAMPLES ###

X = (10.0 * rng.sample(N))                                    # features
Y = [(a * X[i] + b) + rng.normal(0,s) for i in range(N)] # targets
```

Plot samples

In [3]:

```
### PLOT SAMPLES ###

plt.xlabel('Feature')
plt.ylabel('Target')
plt.plot(X, Y, '+')
plt.text(12, 2, 'Dataset generated by:\n\n'
                '$f(x) = %.2f \cdot x + %.2f$\n\n'
                'with a shift randomly chosen from'
                'normal distribution with $\sigma = %.2f$'
                % (a, b, s), fontsize = 15)
plt.show()
```



Theano

In [11]:

```
import theano
import theano.tensor as T

nTrainSteps = 1000 # number of training steps
alpha = 0.01 / N # learning rate

### SYMBOLIC VARIABLES ###

x = T.vector('x') # feature vector
y = T.vector('y') # target vector

w = theano.shared(rng.randn(), name = 'w') # random weights
b = theano.shared(rng.randn(), name = 'b') # bias term (w_0)

### EXPRESSION GRAPH ###

prediction = T.dot(x, w) + b          # hyphothesis
cost = T.sum(T.pow(prediction - y,2)) # cost function
gw, gb = T.grad(cost, [w,b])          # gradients

### COMPILE ###

# update weights to minimize cost
train = theano.function(inputs = [x,y],
                        outputs = cost,
                        updates = ((w, w - alpha * gw),
                                   (b, b - alpha * gb)))
```

Find weights

In [12]:
```
### TRAIN ###

costs = [] # value of cost function in each training step

for i in range(nTrainSteps): costs.append(train(X, Y))

print "f(x) = %.2f * x + %.2f" % (w.get_value(), b.get_value())
```

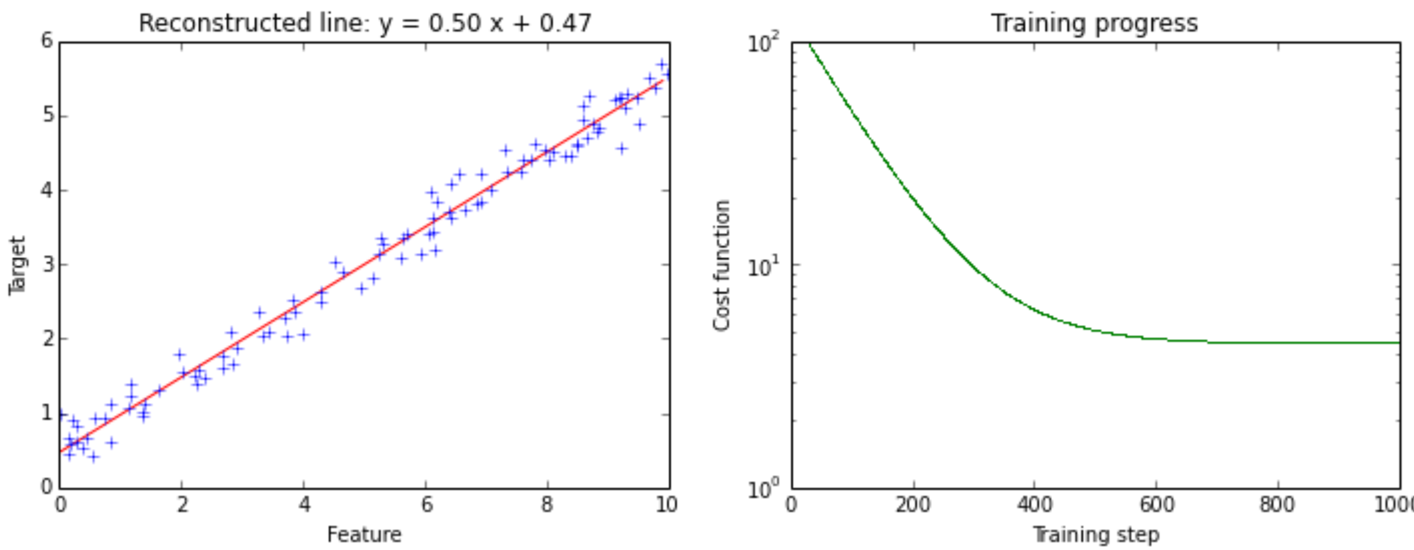
```
f(x) = 0.50 * x + 0.47
```

Plot results

In [13]:
```
def plotMe(): # just to fit on a slide
    fig, plots = plt.subplots(1,2) # create a 1x2 grid of plots
    # double horizontal size of figure
    fig.set_size_inches((2, 1) * fig.get_size_inches())

    # first plot
    plots[0].set_title('Reconstructed line: y = %.2f x + %.2f'
                       % (w.get_value(), b.get_value()))
    plots[0].set_xlabel('Feature')
    plots[0].set_ylabel('Target')

    u = numpy.arange(0, 10, 0.1)
    plots[0].plot(u, w.get_value() * u + b.get_value(), 'r', X, Y, '+')

    # second plot
    plots[1].set_title('Training progress')
    plots[1].set_xlabel('Training step')
    plots[1].set_ylabel('Cost function')

    plt.ylim([1, 100])
    plots[1].set_yscale('log')

    u = numpy.arange(0, nTrainSteps, 1)
    plots[1].plot(u, costs, 'g,')

    plt.show()
```

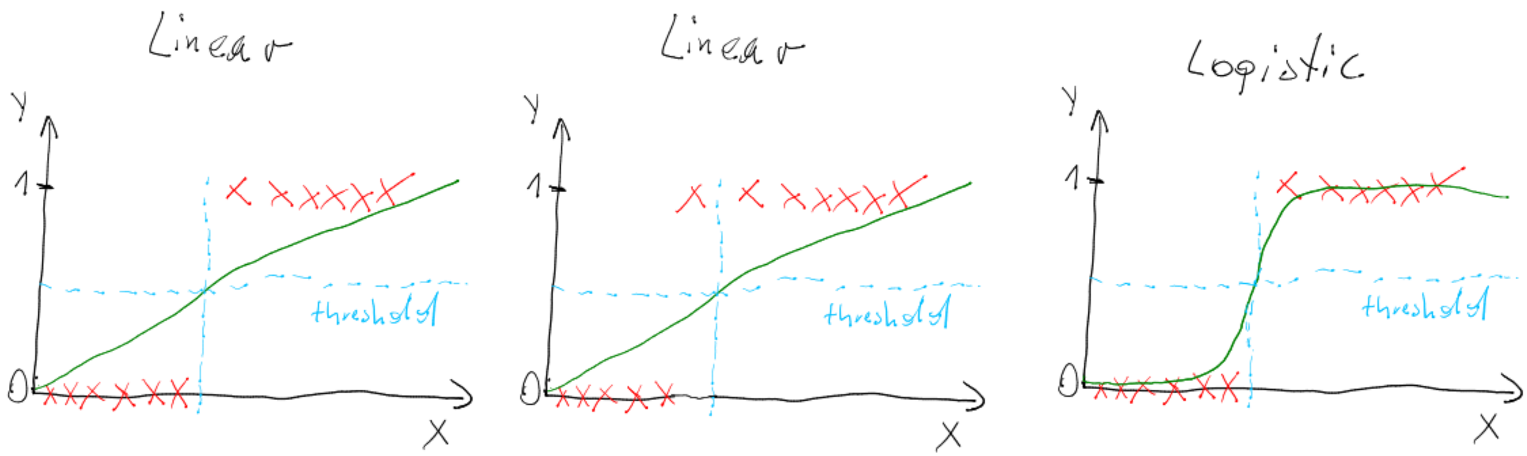
Plot results

In [14]:
```
plotMe()
```



Logistic Regression

Classification - linear vs logistic



Notation

- Logistic function:

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

- Hypothesis:

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

Notation

- Probability of 1:

$$P(y=1|x,w)=h(x)$$

- Probability of 0:

$$P(y=0|x,w)=1-h(x)$$

- Probability:

$$p(y|x,w)=(h(x))^{y}\cdot(1-h(x))^{1-y}$$

Likelihood

- Likelihood:

$$L(w) = p(y|X, w) = \prod_{i=1}^{N} p(y^{(i)}|x^{(i)}, w) = \prod_{i=1}^{N} (h(x^{(i)}))^{y^{(i)}} \cdot (1 - h(x^{(i)}))^{1-y^{(i)}}$$

- Log-likelihood:

$$l(w) = \log L(w) = \sum_{i=1}^{N} y^{(i)} \log h(x^{(i)}) + (1 - y^{(i)}) \log(1 - h(x^{(i)}))$$

Learning

- Gradient of logistic function:

$$\frac{dg(z)}{dz} = g(z)(1 - g(z))$$

- Gradient of log-likehood:

$$\frac{\partial l(w)}{\partial w_j} = (y - h(x)) \cdot x_j$$

Learning

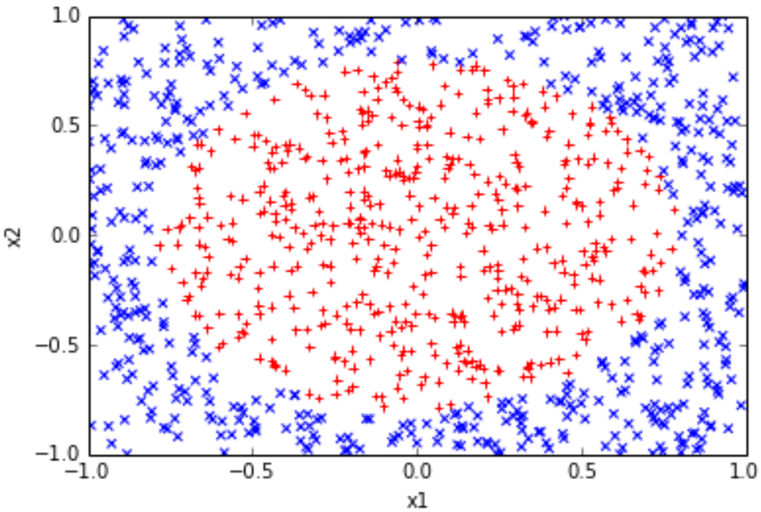
- Cost function:
$$f(w) = -l(w)$$
- Learning step (minimize $f(w)$):
$$w_j = w_j - \alpha \frac{\partial f(w)}{\partial w_j}$$
- Note, it is perceptron, when:
$$g(z) = 1 \text{ for } z \geq 1 \text{ or } 0 \text{ for } z < 1$$

Generate samples

In [15]:
```
%matplotlib inline

### IMPORTS ###

import numpy
import matplotlib.pyplot as plt

rng = numpy.random # random number generator

### SETTINGS ###

N = 1000 # number of samples
n = 4    # number of features

### FUNCTIONS ###

def isInCircle (point): # returns 1 (0) if point is (not) in circle
                        # (radius chosen so #inside ~ #outside)
  return int (point[0] * point[0] + point[1] * point[1] < 2 / 3.14)

### GENERATE SAMPLES ###

randomPoints = (2.0 * rng.sample ((N, 2)) - 1) # [-1,1]x[-1,1]

X = [[p[0], p[1], p[0]*p[0],p[1]*p[1]] for p in randomPoints]
Y = [isInCircle(x) for x in randomPoints] # 1/0 -> inside/outside circle
```

Plot samples

In [17]:
```
plotMe()
```



Dataset:

Points inside / outside circle$x^2 + y^2 = 2/3.14$

No. of points inside = 496

No. of points outside = 504

Theano

In [18]:

```
import theano
import theano.tensor as T

nTrainSteps = 1000 # number of training steps
alpha = 0.01 # learning rate

### SYMBOLIC VARIABLES ###

x = T.matrix('x') # feature vector
y = T.vector('y') # target vector

w = theano.shared(rng.randn(n), name = 'w') # n weights initialized randomly
b = theano.shared(rng.randn(), name = 'b')  # bias term (w_0)

### EXPRESSION GRAPH ###

h = 1 / (1 + T.exp(-T.dot(x, w) - b))          # hypothesis
prediction = h > 0.5                           # prediction threshold
xent = - y * T.log(h) - (1 - y) * T.log(1 - h) # cross-entropy loss function
cost = xent.sum()                              # cost function
gw, gb = T.grad(cost, [w,b])                   # gradients

### COMPILE ###

train = theano.function(inputs = [x,y],
                        outputs = cost,
                        updates = ((w, w - alpha * gw),
                                   (b, b - alpha * gb)))
```

Find weights

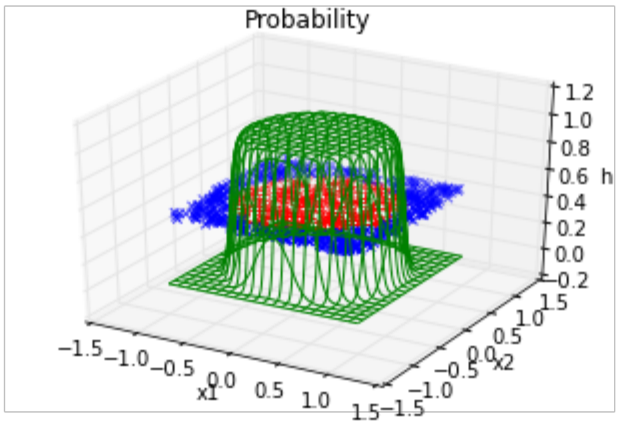
In [19]:

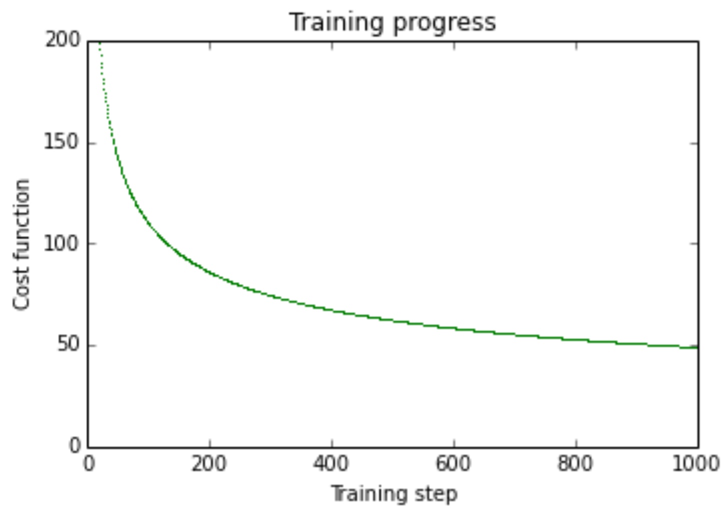
```
### TRAIN ###

costs = [] # value of cost function in each training step

for i in range(nTrainSteps): costs.append(train(X, Y))

print "w =", w.get_value()
print "b =", b.get_value()
```

```
w = [ -0.15778745   0.19991422 -25.09642442 -25.05667537]
b = 15.9232617172
```

Plot results

In [22]:
```
plotMe()
```





Test

In [25]:

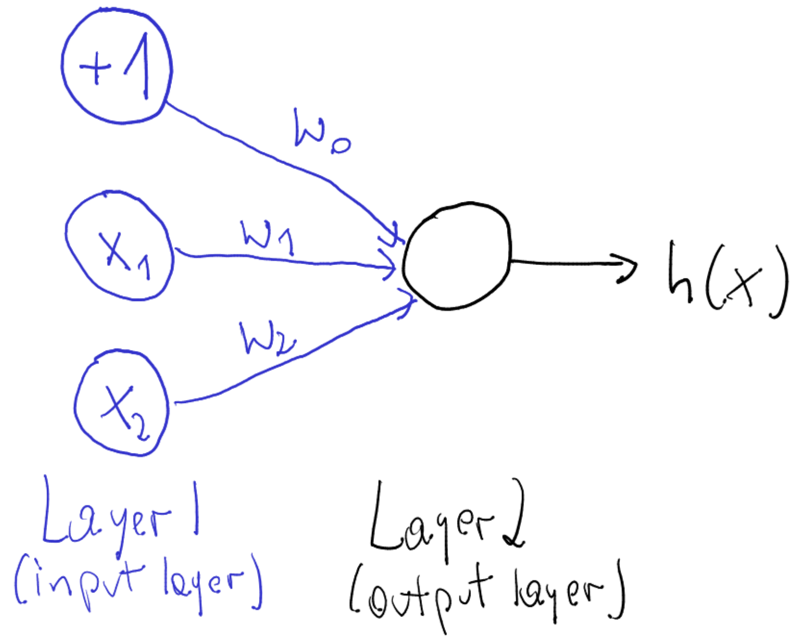
```
### COMPILE THEANO FUNTION ###

predict = theano.function(inputs = [x], outputs = prediction)

### SETTINGS ###

score = 0
nTest = 1000 # number of testing samples

### TESTING SAMPLES ###

randomPoints = (2.0 * rng.sample ((nTest, 2)) - 1) # random points [-1,1]x[-1,1]

testSample = [[p[0], p[1], p[0] * p[0], p[1] * p[1]] for p in randomPoints]

### PREDICT AND CALCULATE SCORE ###

result = predict (testSample) # predict inside / outside for testing sample

for i in range(nTest):
    if result[i] == isInCircle(randomPoints[i]): score += 1

print "Score = ", 1.0 * score / nTest
```

```
Score =  0.996
```

First neural network - AND gate

x	y	x AND y
0	0	0
1	0	0
0	1	0
1	1	1

Neural Network Scratch



- Hypothesis = logistic function: $h(x) = \frac{1}{1+e^{-w^T x}}$
- Intuition: $w_0 < 0, w_0 + w_1 < 0, w_0 + w_2 < 0, w_0 + w_1 + w_2 > 0$

Check intuition

In [34]:

```
from math import exp

### BY HAND WEIGHTS ###

w0 = -300
w1 = 200
w2 = 200

### LOGISTIC FUNCTION ###

def h(x1, x2):
    return 1 / (1 + exp(-w0 - w1 * x1 - w2 * x2))

### TRAINING SET ###

X = [[0,0], [0,1], [1,0], [1,1]] # input
Y = [0, 0, 0, 1]                 # expected output

### TEST ###

for x in X: print '%d AND %d = %f' % (x[0], x[1], h(x[0], x[1]))
```

```
0 AND 0 = 0.000000
0 AND 1 = 0.000000
1 AND 0 = 0.000000
1 AND 1 = 1.000000
```

Prepare Theano

In [35]:
```
import theano
import theano.tensor as T
import theano.tensor.nnet as nnet
import numpy

rng = numpy.random # random number generator

### SETTINGS ###

nTrainSteps = 1000 # number of training steps
alpha = 1.0 # learning rate

### SYMBOLIC VARIABLES ###

x = T.vector('x') # input
y = T.scalar('y') # expected value

w = theano.shared(rng.randn(2), name = 'w') # 2 weights initialized randomly
b = theano.shared(rng.randn(), name = 'b')  # bias term (w_0)

### EXPRESSION GRAPH ###

layer1 = nnet.sigmoid(T.dot(x,w) + b) # input layer
layer2 = T.sum(layer1)                # output layer
cost = (layer2 - y)**2                # cost function
gw, gb = T.grad(cost, [w,b])          # gradients
```

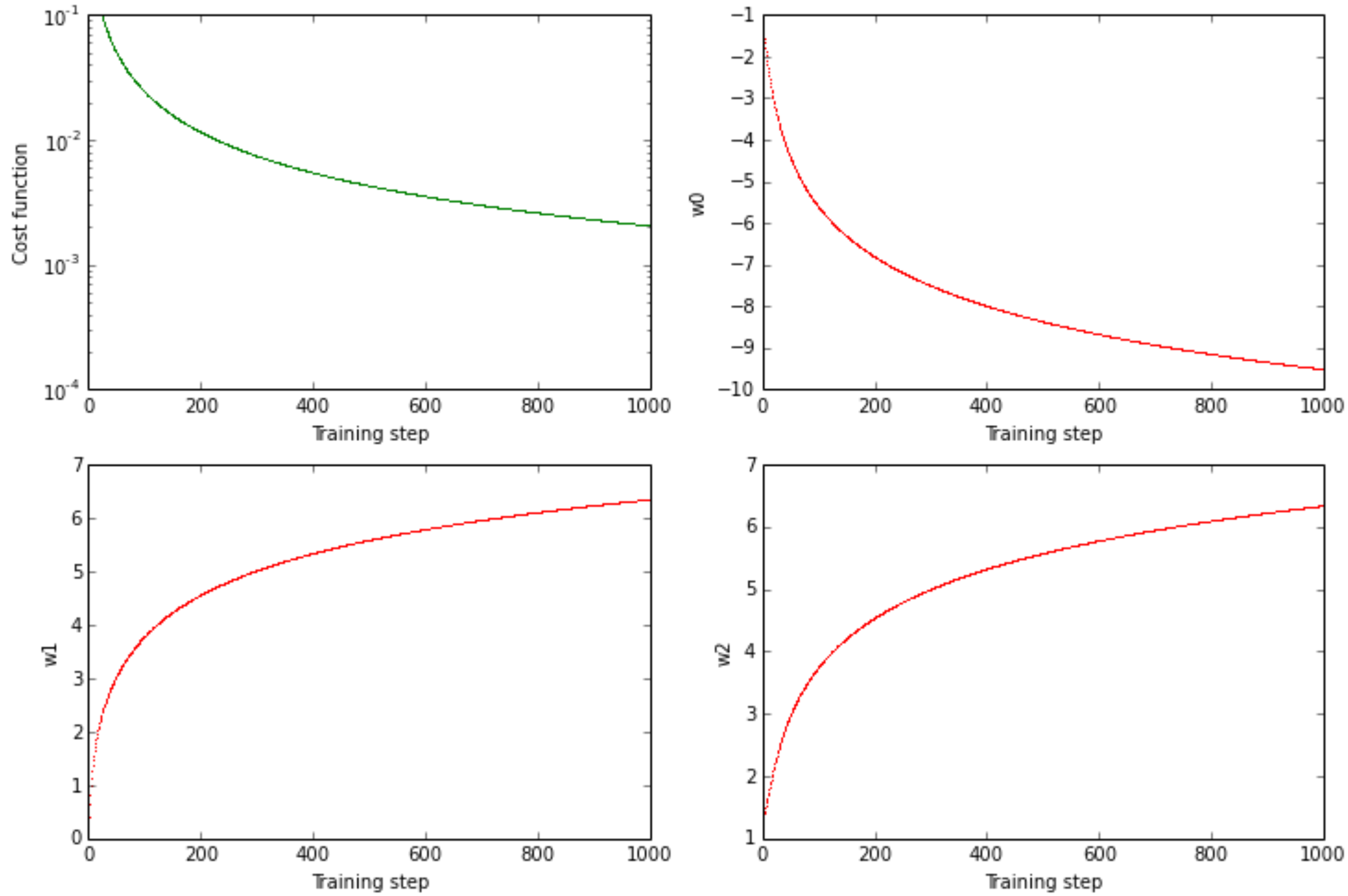
Learn

In [36]:

```
### TRAIN ###

train = theano.function(inputs = [x,y],
                        outputs = cost,
                        updates = ((w, w - alpha * gw),
                                   (b, b - alpha * gb)))

costs   = [] # value of cost function in each training step
weights = [] # value of weights in each training step
bias    = [] # value of bias term in each training step

for i in range (nTrainSteps):
    # train net using each element from X
    for j in range(4): c = train(X[j], Y[j])
    # save progress to plot them later
    costs.append(c)
    weights.append(w.get_value())
    bias.append(b.get_value())
```

Plot weights

In [38]:
```
plotMe()
```



Test

In [39]:

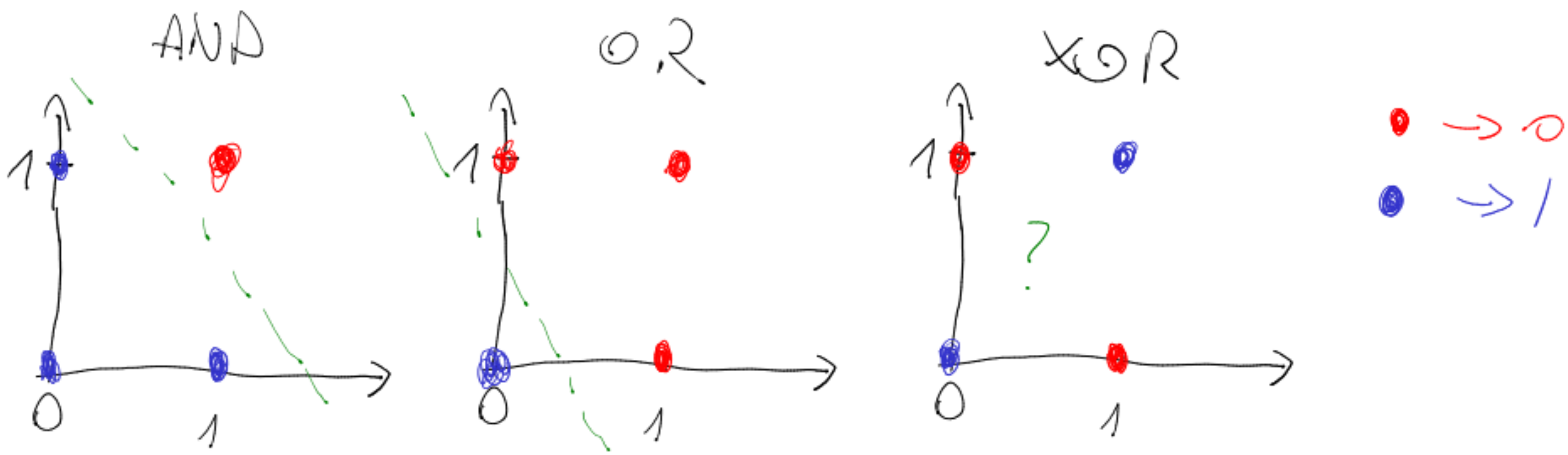
```
### TEST ###

predict = theano.function(inputs=[x], outputs=layer2)

for x in X: print '%d AND %d = %f' % (x[0], x[1], predict(x))
```

```
0 AND 0 = 0.000071
0 AND 1 = 0.037623
1 AND 0 = 0.037746
1 AND 1 = 0.955776
```

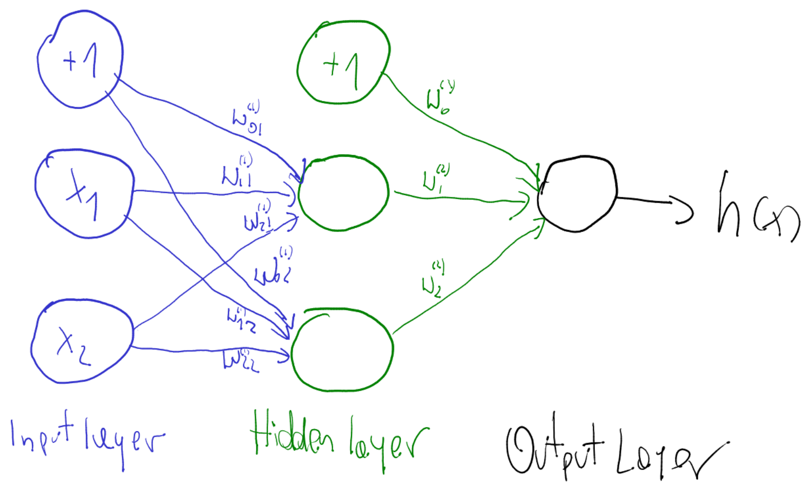
First non-linear NN - XOR gate

x	y	x XOR y
0	0	0
1	0	1
0	1	1
1	1	0

AND & OR vs XOR



Neural Network Scratch



- x XOR y = (x AND NOT y) OR (y AND NOT x)

Imports and stuff

In [3]:

```
### IMPORTS ###

import theano
import theano.tensor as T
import theano.tensor.nnet as nnet
import numpy

rng = numpy.random # random number generator

### SETTINGS ###

nTrainSteps = 10000 # number of training steps

alpha = 0.1 # learning rate

### TRAINING SET ###

X = [[0,0], [0,1], [1,0], [1,1]] # input
Y = [0, 1, 1, 0]                 # expected output
```

Variables and layers

In [4]:

```
### SYMBOLIC VARIABLES ###

x = T.vector('x') # input
y = T.scalar('y') # expected value

# first layer's weights (including bias)
w1 = theano.shared(rng.rand(3,2), name = 'w')
# second layer's weights (including bias)
w2 = theano.shared(rng.rand(3), name = 'b')

### EXPRESSION GRAPH ###

def layer (x, w):             # inputs, weights
    b = numpy.array([1])      # bias term
    xb = T.concatenate([x,b]) # input x with bias added
    return nnet.sigmoid(T.dot(w.T, xb))

hiddenLayer = layer (x, w1)                 # hidden layer
outputLayer = T.sum(layer(hiddenLayer, w2)) # output layer
cost = (outputLayer - y)**2                 # cost function

def gradient (c, w):                 # cost function, weights
    return w - alpha * T.grad (c, w) # update weights
```

Compile, train and run

In [5]:

```
### COMPILE ###

train = theano.function(inputs = [x,y],
                        outputs = cost,
                        updates = [(w1, gradient(cost, w1)),
                                   (w2, gradient(cost, w2))])

predict = theano.function(inputs=[x], outputs=outputLayer)

### TRAIN ###

for i in range (nTrainSteps):
    # train net using each element from X
    for j in range(4): c = train(X[j], Y[j])

### TEST ###

for x in X: print '%d XOR %d = %f' % (x[0], x[1], predict(x))
```

```
0 XOR 0 = 0.033570
0 XOR 1 = 0.970210
1 XOR 0 = 0.970186
1 XOR 1 = 0.031432
```

Something harder than logic gates

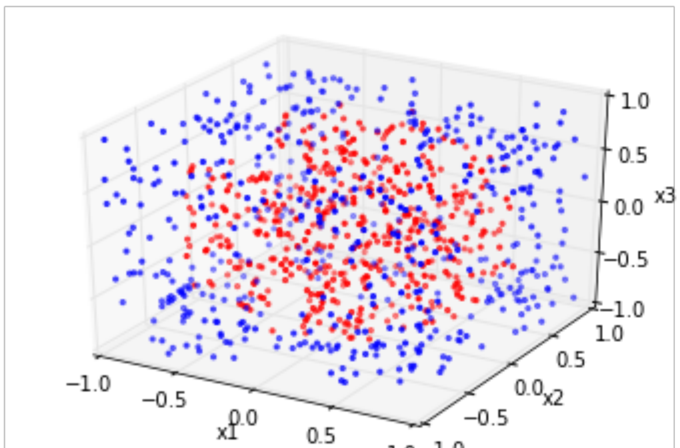
Points inside / outside ball

In [8]:

```
# IMPORTS AND STUFF HIDDEN IN SLIDESHOW

### SETTINGS ###

N = 1000            # number of samples
nTrainSteps = 10000 # number of training steps
alpha = 0.01        # learning rate
nTestSamples = 1000 # number of testing samples

### GENERATE SAMPLES ###

def isInBall (p):
  return int (p[0] * p[0] + p[1] * p[1] + p[2] * p[2] < 1)

X = (2.0 * rng.sample ((N, 3)) - 1) # random points [-1,1]x[-1,1]x[-1,1]
Y = [isInBall(x) for x in X]        # 1/0 for points inside/outside ball
```

Plot samples

In [12]:
```
plotMe()
```



Dataset:

Points inside / outside ball $x^2 + y^2 + z^2 = 1$

No. of points inside = 488

No. of points outside = 512

Prepare Theano

In [15]:
```
### SYMBOLIC VARIABLES ###

x = T.vector('x') # input
y = T.scalar('y') # expected value

w1 = theano.shared(rng.rand(4,7), name = 'w1') # hidden layer's weights (includi
ng bias)
w2 = theano.shared(rng.rand(8), name = 'w2')   # output layer's weights (includi
ng bias)

### EXPRESSION GRAPH ###

def layer (x, w):             # inputs, weights
    b = numpy.array([1])      # bias term
    xb = T.concatenate([x,b]) # input x with bias added
    return nnet.sigmoid(T.dot(w.T, xb))

hiddenLayer = layer (x, w1)                # hidden layer
outputLayer = T.sum(layer(hiddenLayer, w2)) # output layer
cost = (outputLayer - y)**2                # cost function
prediction = outputLayer > 0.5             # prediction threshold

def gradient (c, w):                   # cost function, weights
    return w - alpha * T.grad (c, w) # update weights
```

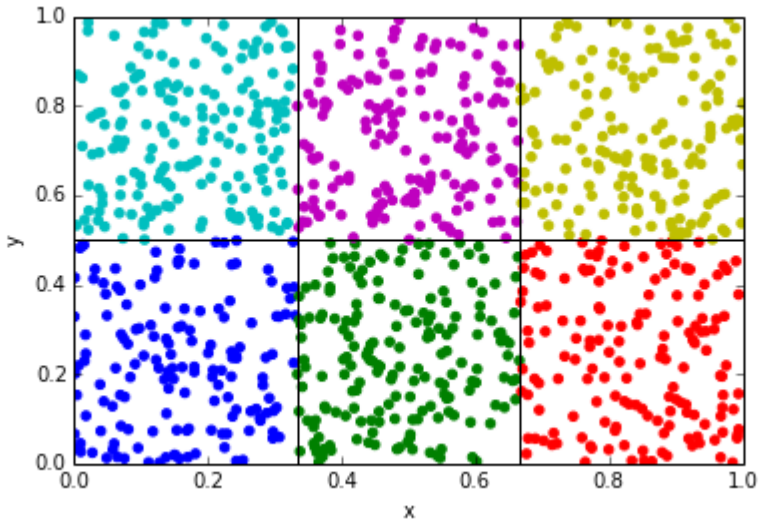
Compile, train and run

In [16]:
```
### COMPILE ###

train = theano.function(inputs = [x,y],
                        outputs = cost,
                        updates = [(w1, gradient(cost, w1)),
                                   (w2, gradient(cost, w2))])

predict = theano.function(inputs=[x], outputs=prediction)

### TRAIN ###

for i in range (nTrainSteps):
    for j in range(N):
        c = train(X[j], Y[j])

### TEST ###

score = 0

for i in range(nTestSamples):
    p = (2.0 * rng.sample (3) - 1)
    if predict(p) == isInBall(p): score += 1

print 'Score =', 1.0 * score / nTestSamples
```

```
Score = 0.963
```

Multiclass classification

In [21]:
```
plotMe()
```



Prepare Theano

In [23]:

```
###### SYMBOLIC VARIABLES ###

x = T.vector('x') # input (vector to extend by bias)
y = T.vector('y') # expected value

# first hidden layer's weights (including bias)
w1 = theano.shared(rng.rand(3,nClasses), name = 'w1')
# second hidden layer's weights (including bias)
w2 = theano.shared(rng.rand(nClasses + 1, nClasses), name = 'w2')
# output layer's weights (including bias)
w3 = theano.shared(rng.rand(nClasses + 1, nClasses), name = 'w3')

### EXPRESSION GRAPH ###

def layer (x, w):
    b = numpy.array([1])      # bias term
    xb = T.concatenate([x,b]) # input x with bias added
    return nnet.sigmoid(T.dot(w.T,xb))

hiddenLayer1 = layer(x, w1)            # hidden layer 1
hiddenLayer2 = layer(hiddenLayer1, w2) # hidden layer 2
outputLayer  = layer(hiddenLayer2, w3) # output layer

cost = T.sum(T.pow(outputLayer - y, 2)) # cost function

def gradient (c, w):                 # cost function, weights
    return w - alpha * T.grad (c, w) # update weights
```

Compile, train and run

In [29]:

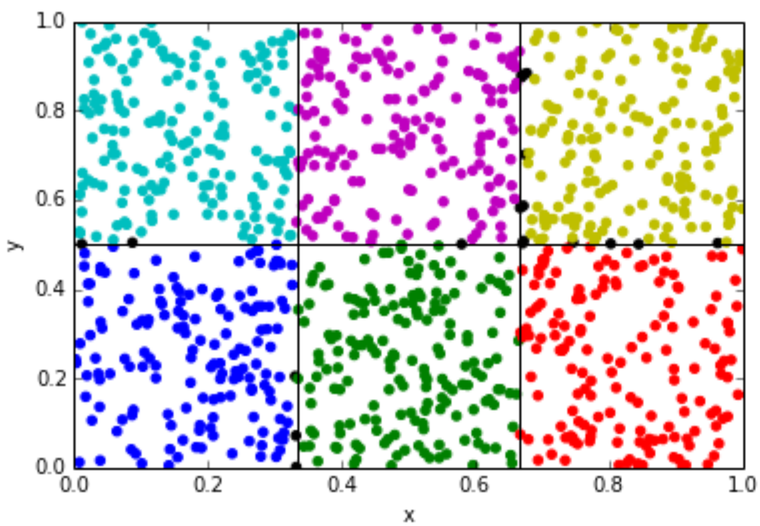
```
### COMPILE ###
train = theano.function(inputs = [x,y],
                        outputs = cost,
                        updates = [(w1, gradient(cost, w1)),
                                   (w2, gradient(cost, w2)),
                                   (w3, gradient(cost, w3))])

predict = theano.function(inputs=[x], outputs=outputLayer)
### TRAIN ###
for i in range (nTrainSteps):
    for j in range(nTrainSamples): c = train(X[j], Y[j])
### TEST ###
score = 0 # final score
good = [] # correctly reconstructed points
bad = []  # not correctly ...

for i in range(nTestSamples):          # generate nTestSamples testing points
    t = rng.sample(2)                  # random point
    p = predict(t).tolist()            # prediction [prob c1, prob c2, ...]
    if p.index(max(p)) == getClass (t): # prediction = max prob
        score += 1                     # good prediction
        good.append(t)
    else: bad.append(t)                # bad prediction

print '\nScore =', 1.0 * score / nTestSamples
```

```
Score = 0.982
```

Plot results

In [31]:

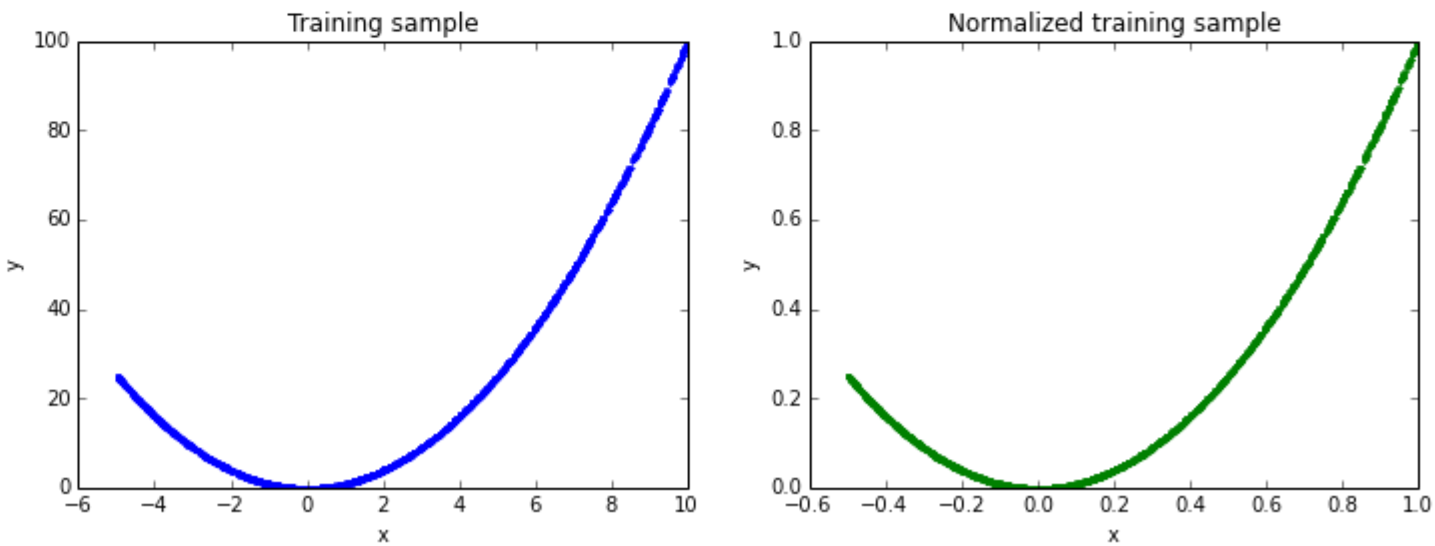
```
plotMe()
```



Black points are misclassified

Regression

In [54]:
```
plotMe()
```



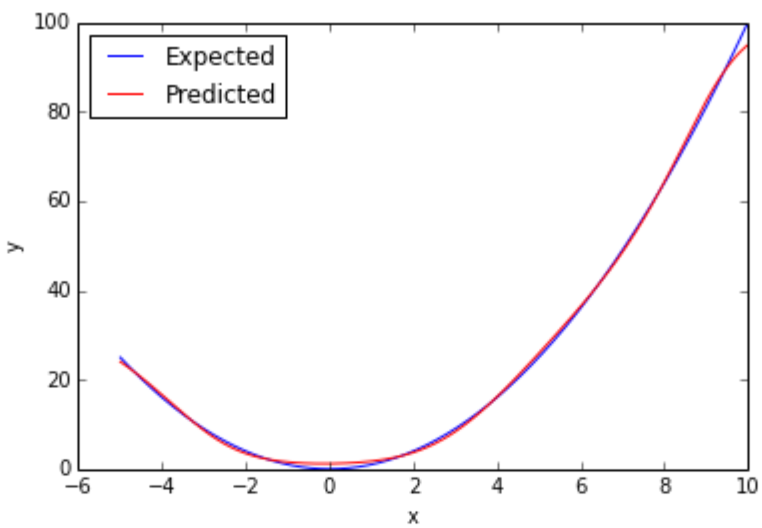
Theano settings

In [55]:

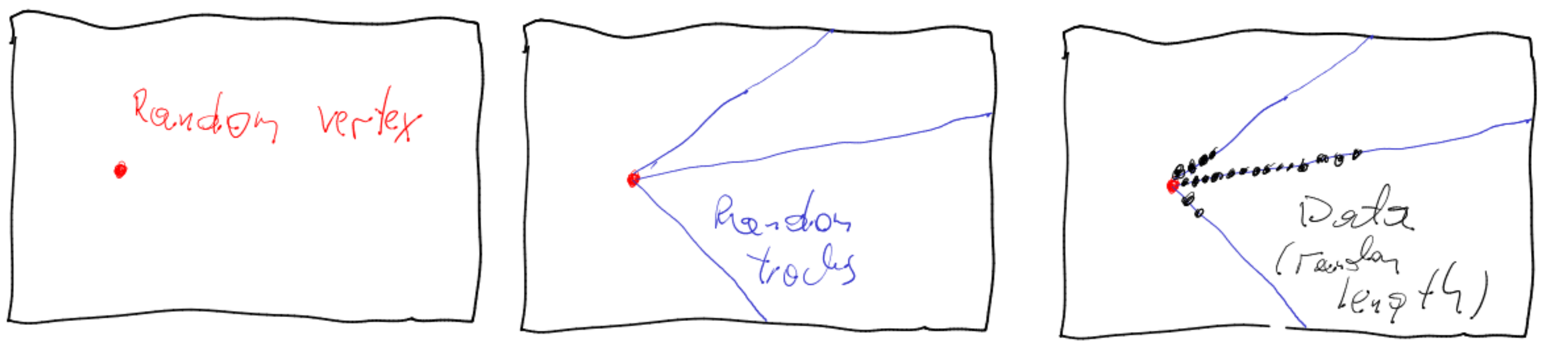
```
###### SYMBOLIC VARIABLES ###

x = T.vector('x') # input
y = T.scalar('y') # expected value

w1 = theano.shared(rng.rand(2,2), name = 'w1') # first hidden layer's weights (i
ncluding bias)
w2 = theano.shared(rng.rand(3,2), name = 'w2') # second hidden layer's weights (
including bias)
w3 = theano.shared(rng.rand(3), name = 'w3')   # output layer's weights (includi
ng bias)
```

Results

In [57]:
```
test = (randomX(nTestSamples)) # random x values
test.sort()                    # sorted x values

expected  = [f(t) for t in test]                              # expexted values
for test points
predicted = [scaleBack(predict([t / xFactor])) for t in test] # predicited value
s for test points

plt.xlabel('x'); plt.ylabel('y')
plt.plot(test, expected, 'b', label = 'Expected')
plt.plot(test, predicted, 'r', label = 'Predicted')

plt.legend(loc=2) # legend in upper left corner

plt.show()
```



Toy Detector



Settings

In [33]:
```
### IMPORTS ###

%matplotlib inline

import numpy
import matplotlib.pyplot as plt
import math

rng = numpy.random # random number generator

### DETECTOR SETTINGS ###

dim = [100,50]      # dimension [width, height]
N = dim[0] * dim[1] # number of blocks in the detector

### EVENT SETTINGS ###

nTracks = 5 # maximum no. of tracks
```

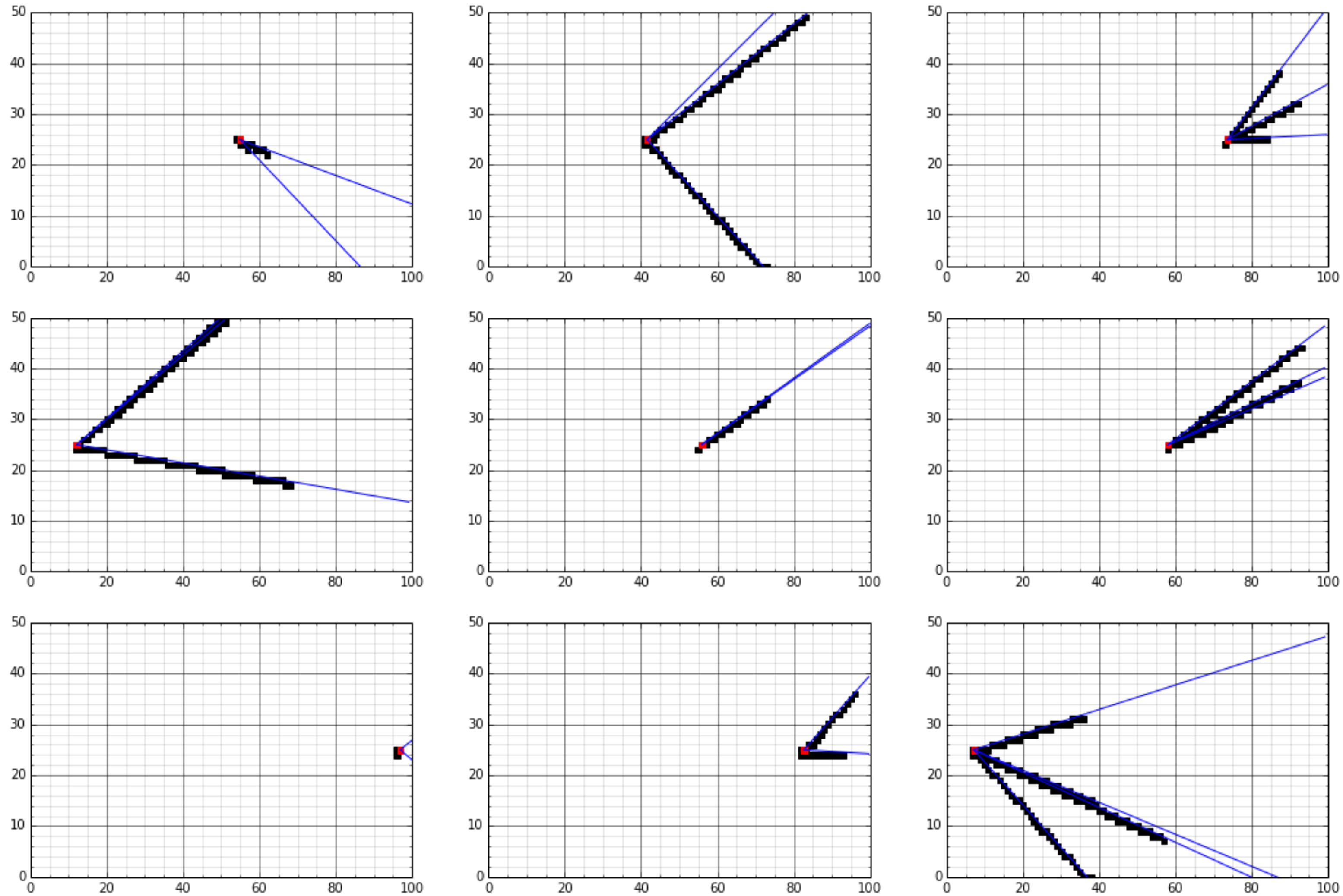
Event

In [35]:

```
class Event:
    # initialize with random vertex and generate tracks
    def __init__ (self):
        # vertex = [x,y]
        self.vertex =  dim * rng.sample(2)
        self.vertex[1] = dim[1] / 2 # in this version all vertices at y-center
        # each event has random number of tracks
        self.tracks = [self.genTrack() for i in range(rng.randint(2,nTracks))]
        # data represents as array of 0 (no track) and 1 (track) points in the d
etector
        self.data = [0] * N
        # fill data with tracks
        for t in self.tracks: self.genData(t[0], t[1])
    # generate track (random line coming from vertex)
    def genTrack (self):
        # y = ax + b
        a = 2.0 * rng.sample() - 1.0
        b = self.vertex[1] - a * self.vertex[0]
        return [a,b]
    # generate data from tracks
    def genData (self, a, b):
        # track starts in the vertex
        start = int(self.vertex[0])
        # track length is random
        end = start + int((dim[0] - start) * rng.sample())
        # "convert" track to detector points
        for x in range(start, end):
            y = int(a * x + b)  # round y
            if y < 0 or y >= dim[1]: break
            self.data[x + y * dim[0]] = 1
```

Event display

In [39]:
```
EventDisplay(3,3)
```



Out[39]:
```
<__main__.EventDisplay instance at 0x7f2dbda02c68>
```

Settings

In [40]:

```
### NET SETTINGS ###

nTrainSamples = 100000 # number of learning samples
nTrainSteps = 400      # number of training steps
alpha = 0.1            # learning rate
nTestSamples = 1000    # number of testing samples

h1 = 15 # no. of neurons in first hidden layer
h2 = 10 # no. of neurons in second hidden layer

###### SYMBOLIC VARIABLES ###

x = T.vector('x') # input
y = T.scalar('y') # expected value

w1 = theano.shared(rng.rand(N + 1, h1), name = 'w1')
w2 = theano.shared(rng.rand(h1 + 1, h2), name = 'w2')
w3 = theano.shared(rng.rand(h2 + 1), name = 'w3')
```

Training samples

In [42]:

```
### GENERATE TRAINING SAMPLES ###

X = []
Y = []

for i in range(nTrainSamples):
    e = Event()
    X.append(e.data)
    # normalize Y to [0,1]
    Y.append(e.vertex[0] / dim[0])

# important for running on GPU
Y = numpy.array(Y, dtype='float32')
```

Results

- Full code: https://github.com/TomaszGolan/mlScratchpad/blob/master/10_theano_toy_detector.py (https://github.com/TomaszGolan/mlScratchpad/blob/master/10_theano_toy_detector.py)

In [ ]:

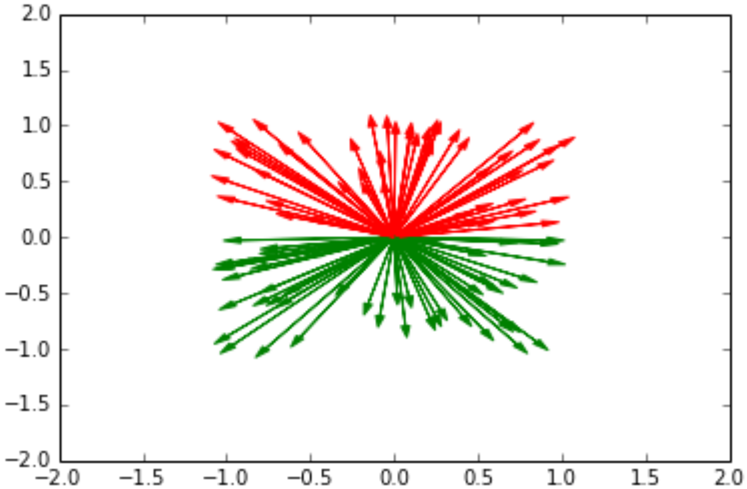
```
Epoch: 391, cost = 0.000011
Epoch: 392, cost = 0.000010
Epoch: 393, cost = 0.000009
Epoch: 394, cost = 0.000008
Epoch: 395, cost = 0.000007
Epoch: 396, cost = 0.000006
Epoch: 397, cost = 0.000006
Epoch: 398, cost = 0.000005
Epoch: 399, cost = 0.000004
Epoch: 400, cost = 0.000004

Reconstruted events: 95.50%
         within 1 planes: 49.200000%
         within 2 planes: 23.000000%
         within 3 planes: 12.600000%
         within 4 planes: 7.300000%
         within 5 planes: 3.400000%
```

Some thoughts on NN

Vectors up and down

In [3]:

```
X = (2.0 * rng.sample((N,2)) - 1) # [-1,1]x[-1,2]
Y = [int(x[1] > 0) for x in X]    # 1/0 for up/down

# Plot training samples

plt.xlim([-2,2])
plt.ylim([-2,2])

for i in range(100):
    if Y[i]: c = 'r'
    else: c = 'g'
    plt.arrow(0, 0, X[i][0], X[i][1], head_width=0.05, head_length=0.1, fc=c, ec
=c)

plt.show()
```



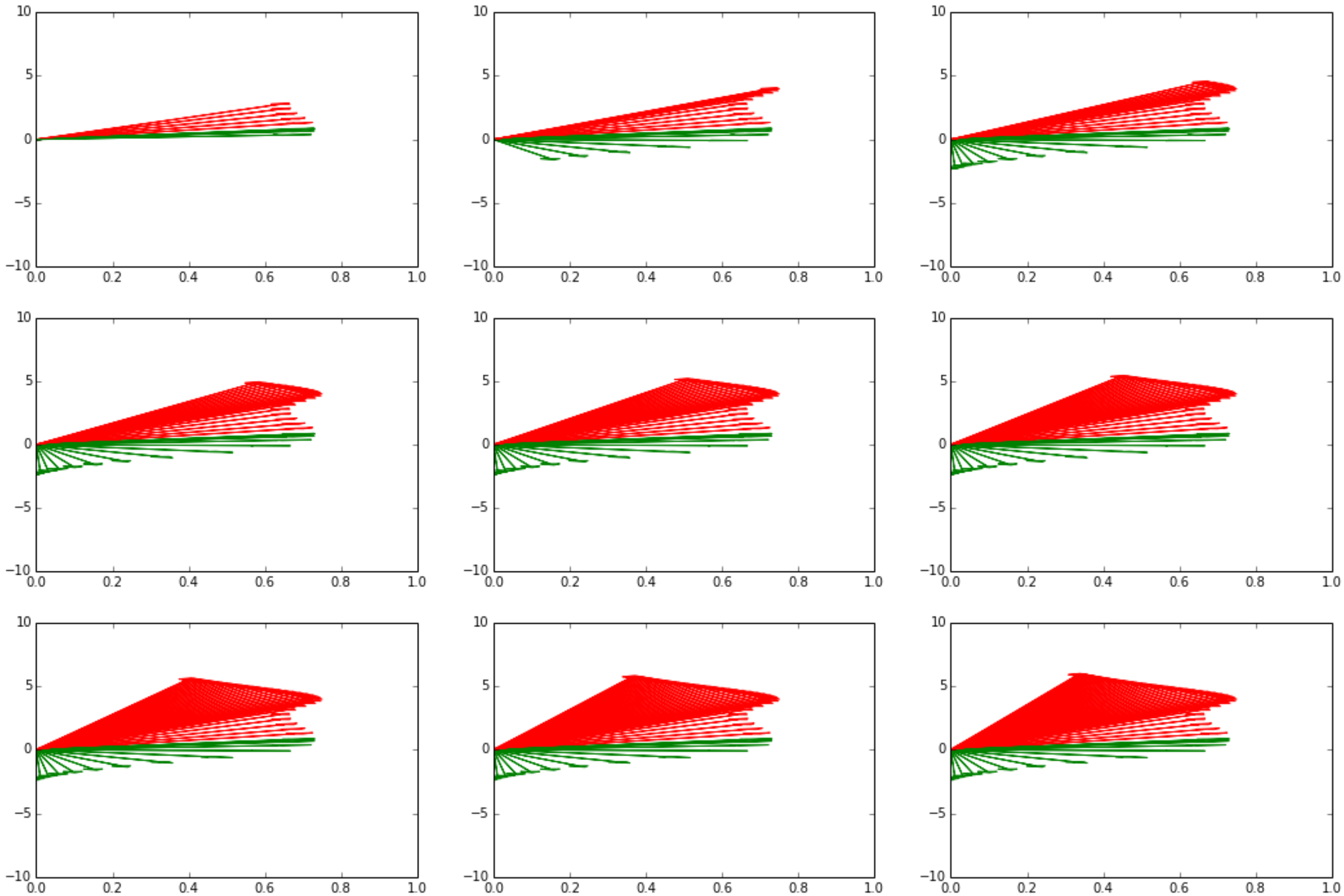
Train with two neurons

In [13]:

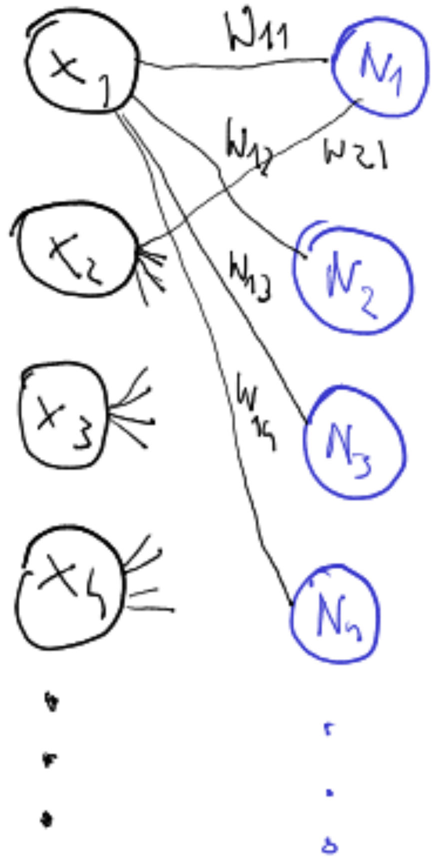
```
### TRAIN ###

neuron1 = []
neuron2 = []

for i in range (nEpochs):
    for j in range(N): c = train(X[j], Y[j])
    neuron1.append([w1.get_value()[0][0], w1.get_value()[1][0]])
    neuron2.append([w1.get_value()[0][1], w1.get_value()[1][1]])
```

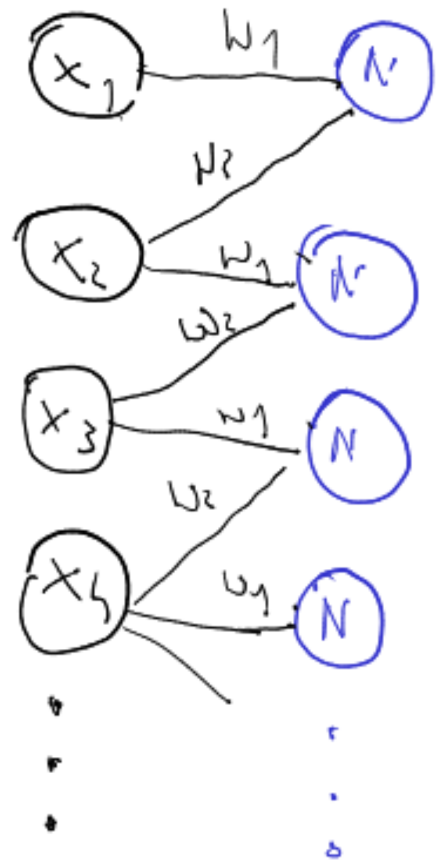
Weights as vectors

In [15]:
```
plotMe()
```

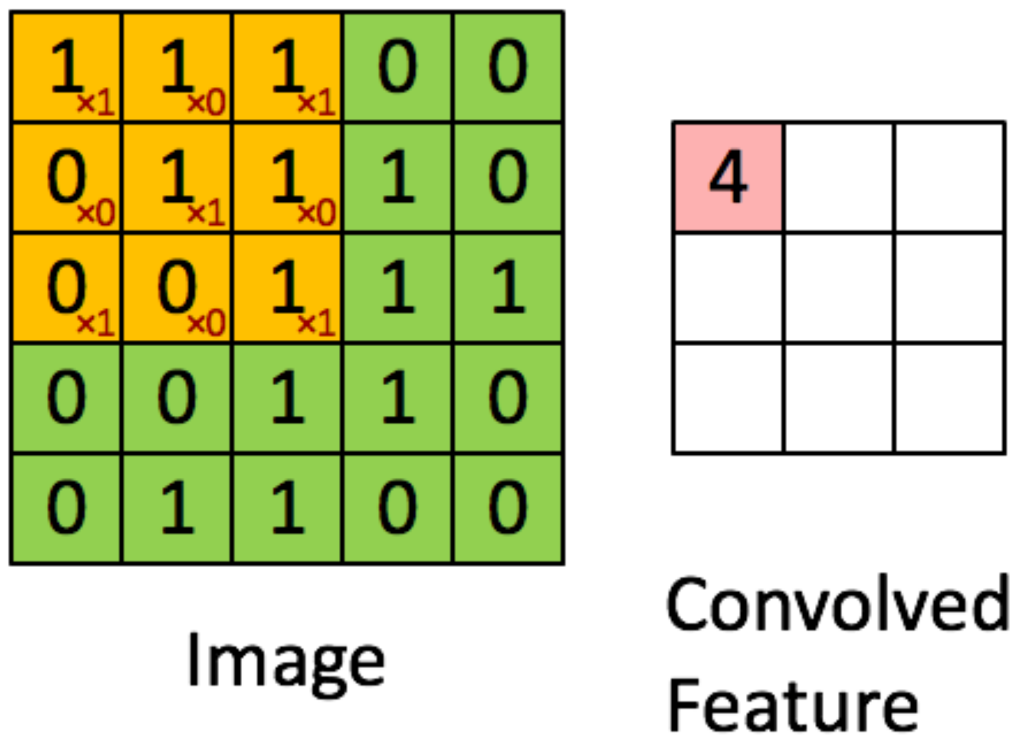


Convolutional Neural Network





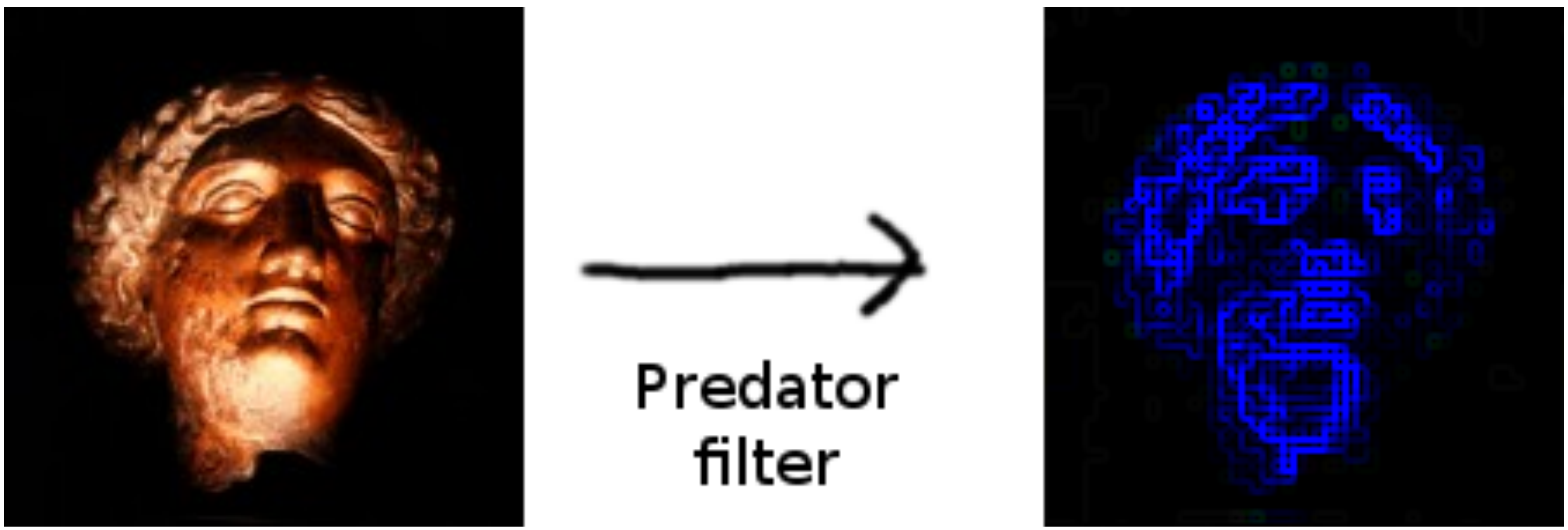
Convolution



(src: http://deeplearning.stanford.edu/wiki/images/6/6c/Convolution_schematic.gif (http://deeplearning.stanford.edu/wiki/images/6/6c/Convolution_schematic.gif))

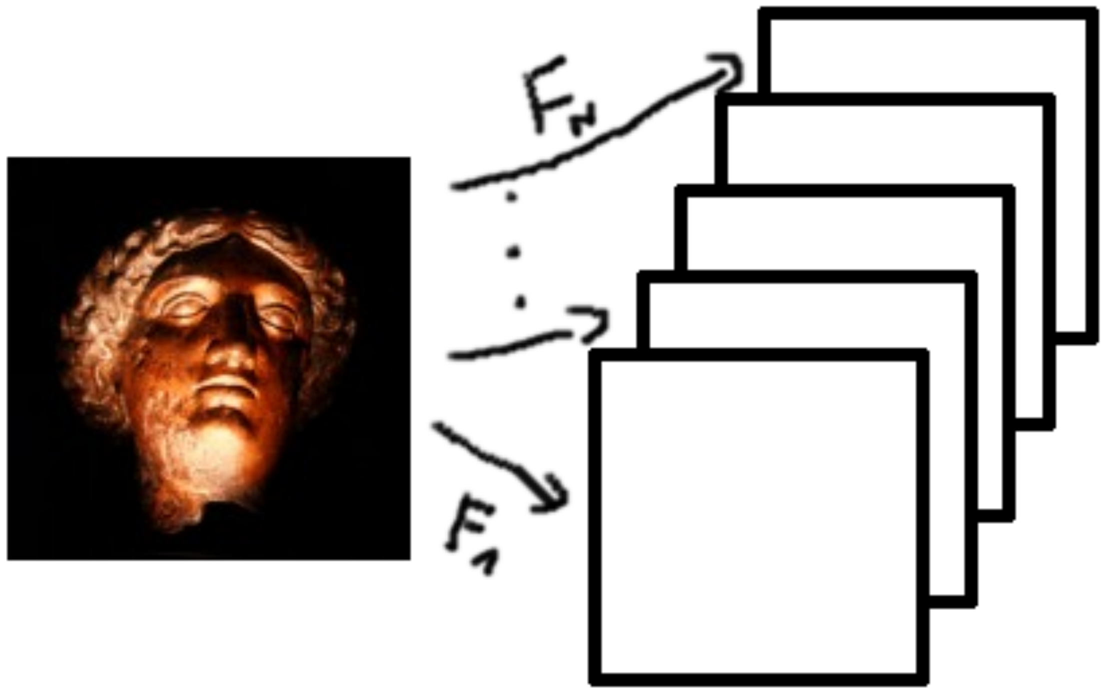
Convolution

”Clones” of a neuron looking at different part of an image

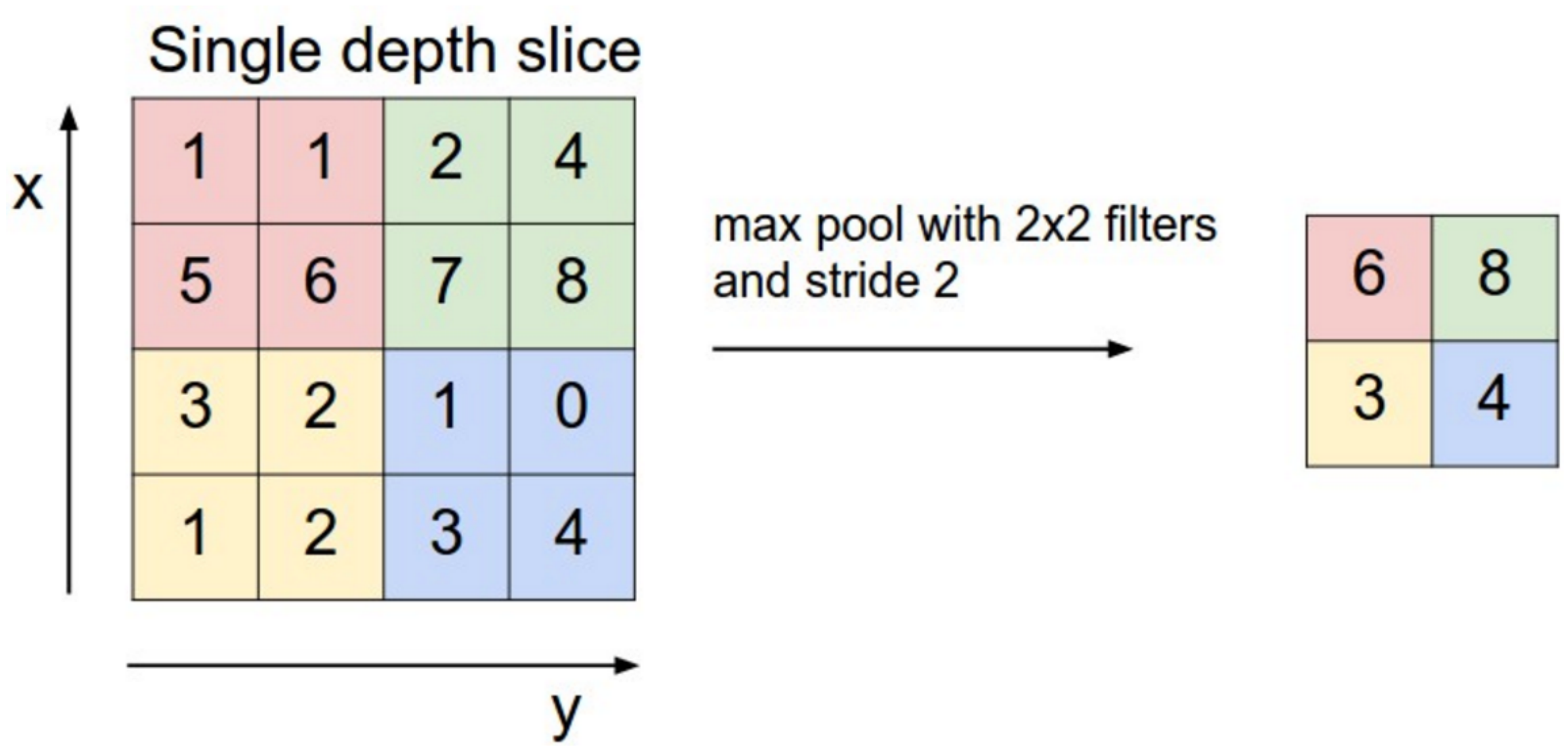


Convolution Layer

No. of convolved feature vectors / matrices = No. of filters

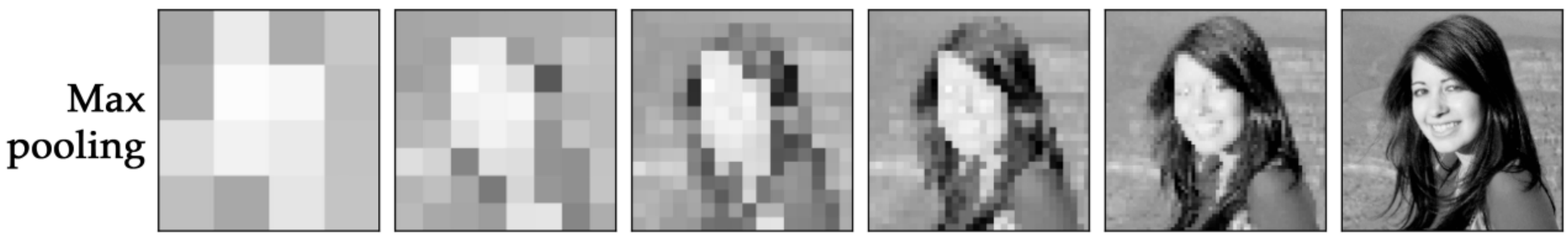


Pooling



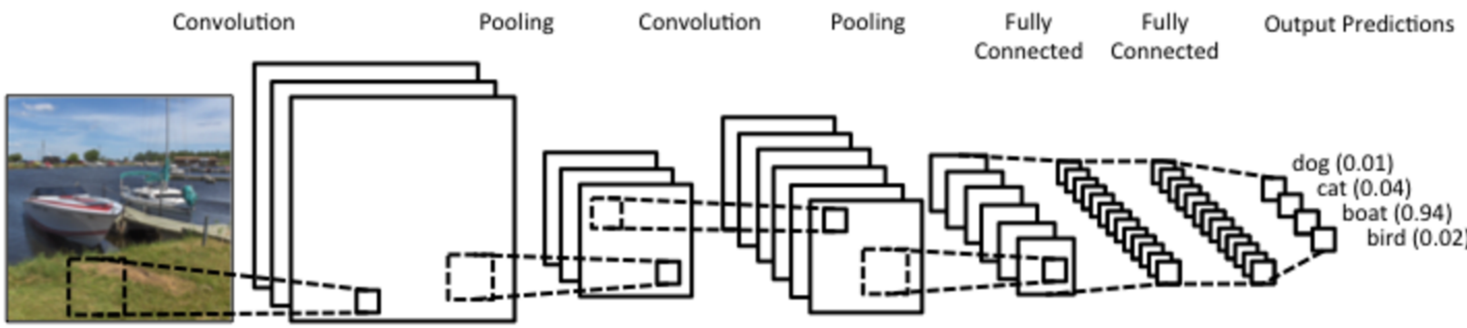
(src: http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/ (http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/))

Pooling



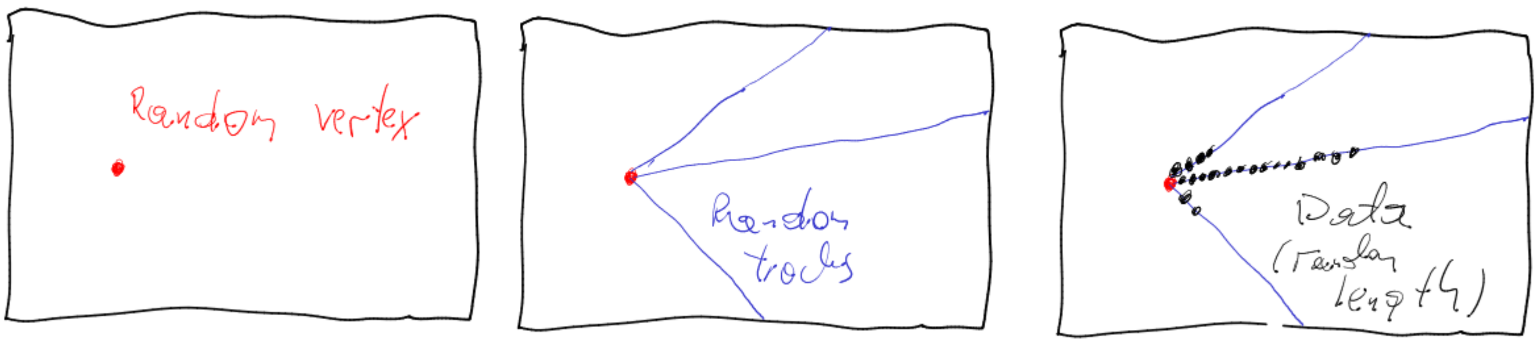
src: http://arxiv.org/abs/1506.03767 (http://arxiv.org/abs/1506.03767)

CNN Example



src: http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/ (http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/)

Toy Detector with CNN



Previously

- 15 neurons in first hidden layer
- 10 neurons in second layer
- many attempts to find 15 and 10...
- 100k training samples
- 400 epochs to get accuracy about 95%

Settings for CNN

- net taken from Lasagne tutorial
- adjusted to regression for toy detector
- 10k training samples
- 10-20 epochs to get accuracy about 95%
- code: https://github.com/TomaszGolan/mlScratchpad/blob/master/12_lasagne_toy_detector.py (https://github.com/TomaszGolan/mlScratchpad/blob/master/12_lasagne_toy_detector.py)

Results

In [38]:
```
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt

with open("toyDetectorConv_10k.dat", "r") as infile:
    data = [map(float, line.split()) for line in infile]

plt.xlabel('Epoch')
plt.ylabel('Accuracy')

plt.plot(*zip(*data))

plt.show()
```