Simple Neural Network

Day 1 - Solving XOR

In this section, the basic of the neural network will be reviewed. This basics includes the implementation of the forward and backward path of a neural network and try to solve XOR problem. The reason XOR is chosen is that, it is not linear and thus can not be solved linear discriminators as well as one simple perceptron. We firts solve the XOR problem and then, after the network proofed, we jump to harder problems.

Below is the implementation of this function for two inputs as well as all four possible combination of the inputs.

In [1]:

```
import os
import numpy as np
from random import randint
import matplotlib.pyplot as plt

def xor(x, y):
    return 1 if x+y != 1 else 0

x_train = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1],
])

y_train = np.array([xor(i[0], i[1]) for i in x_train])

for i in range(4):
    print x_train[i], '~>', y_train[i]
```

```
[0 0] ~> 1
[0 1] ~> 0
[1 0] ~> 0
[1 1] ~> 1
```

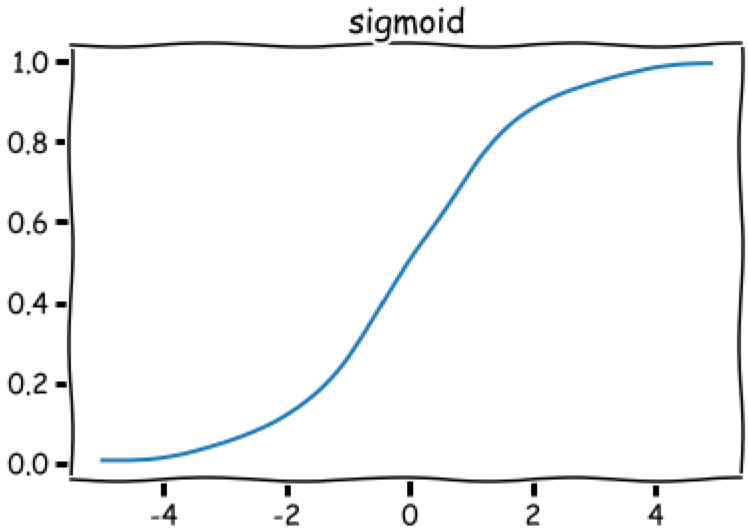
Next, let's define an activation function. This will add the non-linearity to the neuron model. The function we want use is `sigmoid` function. Sigmoid is one of the most popular activations in the Neural Networks. It is defined as below:

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

This means, whenever the input is low (let's say below -4) the activation function returns 0 and on the other hand, when the input is high (similar to the previous, like +4 and above) the activation function return active or 1. This function is plotted in the below section.

In [2]:

```
def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

with plt.xkcd():
    x = np.array(list(range(-50, +50))) / 10.0
    plt.plot(x, sigmoid(x))
    plt.title('sigmoid')
    plt.show()
```



To solve this problem we have used a two-layered structure as depicted below. We know this structure is capable of solving this problem. We can test it by set the weights manually. Note that in this structure the a1 plays role of AND function and a2 playes the role of AND-of-Negative-Inputs. and then o has functionality of OR function and has the output we desire.
Neural Networkk Design

In [3]:

```
W1 = np.array([[20, 20], [-20, -20]])
B1 = np.array([[-30], [10]])

W2 = np.array([[20, 20]])
B2 = np.array([[-10]])
```

The calculation of each layer of the network is as below. In which the weight of each connection is multiplied by the value of the connection end. Then these values sum up together to form the observation of the neuron, z. Then the non-linearity function, also known as activation function, is applied. This calculation for the layer L of the network is drived as:

$$z_l = w^l z_{l-1} + b^l$$
$$a_l = g(z_l)$$

In this equations, the g is the activation function. The W is the weight matrix, and consists of the weight of the whole layer L. We have to note that, z_0 is the input values or $x^{(i)}$. Below is the implementation of these equation:

In [4]:

```
x0 = np.reshape(x_train[0], (2, 1))

# Layer #1
z1 = W1.dot(x0) + B1
a1 = sigmoid(z1)

# Layer #2
z2 = W2.dot(a1) + B2
a2 = sigmoid(z2)

output = 1 if z2 > 0.5 else 0
print x_train[0], '~~>', output
```

```
[0 0] ~~> 1
```

If we tie together these operations, we would have `forward` function. Then we can run it for the entier train set:

In [5]:

```
def forward(a0):
    # Layer #1
    z1 = W1.dot(a0) + B1
    a1 = sigmoid(z1)

    # L\ayer #2
    z2 = W2.dot(a1) + B2
    a2 = sigmoid(z2)
    return a2

for x_i, y_i in zip(x_train, y_train):
    print x_i, '~~>', '%.2f' % forward(np.reshape(x_i, (2,1))), '[%d]' % y_i
```

```
[0 0] ~~> 1.00 [1]
[0 1] ~~> 0.00 [0]
[1 0] ~~> 0.00 [0]
[1 1] ~~> 1.00 [1]
```

Backpropagation

Up to here, we have imagined we have the weights for the network. But how one should come up with these weights? In this section we will walk through the process of calculating these weights. First, let's say we have random weights. Then, we need a way to tell how well these random weight will perform on the input set. Thus, we define a loss function as:

$$J(W) = \frac{1}{m} \sum_{i=1}^{m} (\hat{Y}^{(i)} - y^{(i)})^2$$

In [6]:

```
# randomly initialize the weights
W1 = np.random.uniform(low=0, high=+1, size=(2,2))
B1 = np.random.uniform(low=0, high=+1, size=(2,1))
W2 = np.random.uniform(low=0, high=+1, size=(1,2))
B2 = np.random.uniform(low=0, high=+1, size=(1,1))
```

In [7]:

```
# calculating loss function - logistic error
loss = 0
for x_i, y_i in zip(x_train, y_train):
    h_t = forward(np.reshape(x_i, (2,1)))
    loss += (h_t - y_i) ** 2
loss /= x_train.shape[0]

print 'loss = %.2f' % loss
```

```
loss = 0.37
```

Before the next step, we have to augment the inputs in order to achieve a better performance (Since, the oprations are in form of matrix multiplication, it's faster to do it over augmented results rather than run it several times).

In [8]:

```
x_train_aug = []
y_train_aug = []
for _ in range(1024):
    a = randint(0, 1)
    b = randint(0, 1)
    x_train_aug.append([a, b])
    y_train_aug.append([xor(a, b)])

x_train_aug = np.array(x_train_aug)
y_train_aug = np.array(y_train_aug)
```

In order to computer the weights, we have to optimize them gradually toward the optimal state, by minimizing the `loss` value. To do so, we simply have to calculate the derivation of the `loss` value with respect of each weight and then subtract a portion of it from that weight on each iteration. These derivations are calculated as below:

$$\frac{\partial J}{\partial W_j} = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial J^{(i)}}{\partial y^{(i)}}\frac{\partial y^{(i)}}{\partial w_j}$$

So we have:

$$= \frac{1}{m}\sum_{i=1}^{m}\frac{\partial(\hat{y}^{(i)} - y^{(i)})^2}{\partial y^{(i)}}\frac{\partial y^{(i)}}{\partial w_j}$$

And after calculating the derivation we would have:

$$\nabla_w L = -\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})\frac{\partial y^{(i)}}{\partial w_j}$$

Finally, with having the gradient we can update the weights:

$$W^{t+1} = W^t + \eta\nabla_w L$$

In which the η is a hyperparameter of the system and is called learning-rate.

In [9]:

```
def derivative_sigmoid(x):
    return sigmoid(x) * (1 - sigmoid(x))
```

In [10]:

```
# first run the forward path in other to have the output layers
x_t = x_train[0]
y_t = y_train[0]

a0 = np.reshape(x_t, (2,1))

# Layer #1
z1 = W1.dot(a0) + B1
a1 = sigmoid(z1)

# Layer #2
z2 = W2.dot(a1) + B2
a2 = sigmoid(z2)
```

In [11]:

```
y_hat = a2

e2 = y_t - y_hat
delta_2 = e2 * derivative_sigmoid(z2)
print 'δ2:', delta_2

delta_1 = W2.T.dot(delta_2) * derivative_sigmoid(z1)
print 'δ1:', delta_1
```

```
δ2: [[0.02599883]]
δ1: [[0.00637424]
 [0.00175163]]
```

So, we calculate the Δ value for each weight as below:

In [12]:

```
# now do the above for all trainset
def backward(x0, y0):
    # Forward
    z1 = W1.dot(x0) + B1
    a1 = sigmoid(z1)
    z2 = W2.dot(a1) + B2
    a2 = sigmoid(z2)

    # Backward
    delta_2 = (y0-a2)*(derivative_sigmoid(z2))
    delta_1 = W2.T.dot(delta_2)*(derivative_sigmoid(z1))

    m = x0.shape[1]

    d1w = delta_1.dot(x0.T) / float(m)
    d1b = delta_1.dot(np.ones((m,1))) / float(m)

    d2w = delta_2.dot(a1.T) / float(m)
    d2b = delta_2.dot(np.ones((m,1))) / float(m)
    return d1w, d1b, d2w, d2b


Delta_1_w = np.zeros_like(W1)
Delta_1_b = np.zeros_like(B1)
Delta_2_w = np.zeros_like(W2)
Delta_2_b = np.zeros_like(B2)

for x_t, y_t in zip(x_train, y_train):
    d1w, d1b, d2w, d2b = backward(np.reshape(x_t, (2,1)), y_t)
    Delta_1_w += d1w
    Delta_1_b += d1b
    Delta_2_w += d2w
    Delta_2_b += d2b

print 'Δ1w:\n', Delta_1_w
print 'Δ1b:\n', Delta_1_b
print ''
print 'Δ2w:\n', Delta_2_w
print 'Δ2b:\n', Delta_2_b
```

```
Δ1w:
[[-0.02435853 -0.01896964]
 [-0.00614098 -0.00507039]]
Δ1b:
[[-0.04008598]
 [-0.01027731]]

Δ2w:
[[-0.11830779 -0.14295591]]
Δ2b:
[[-0.18080588]]
```

Finally, we run the eintire process in a loop for `epoch` iterations:

In [13]:

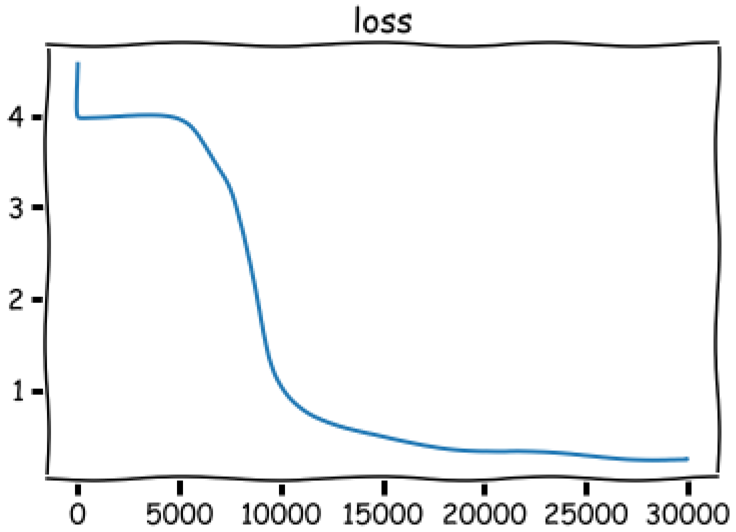
```
W1 = np.random.uniform(low=0, high=+1, size=(2,2))
B1 = np.random.uniform(low=0, high=+1, size=(2,1))
W2 = np.random.uniform(low=0, high=+1, size=(1,2))
B2 = np.random.uniform(low=0, high=+1, size=(1,1))

# used in plotting the progress
loss_history = []
hist_W1 = []
hist_B1 = []
hist_W2 = []
hist_B2 = []
```

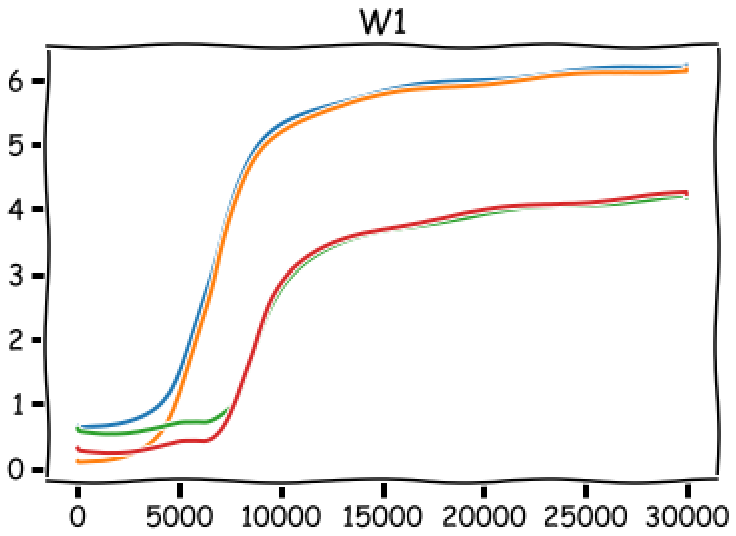
In [14]:

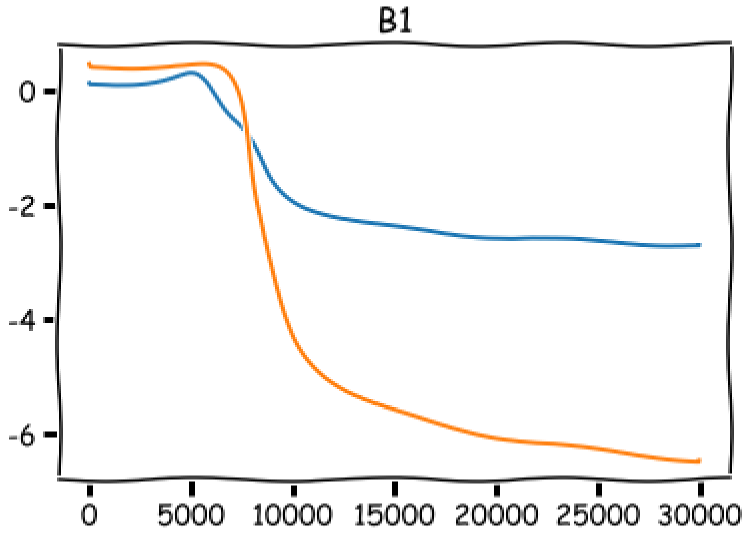
```
epochs = 30000
batch_size = 64
lr = .02

for e in range(epochs):
    i=0
    batch_loss = []
    while(i<x_train_aug.shape[0]):
        x_batch = x_train_aug.T[:, i:i+batch_size]
        y_batch = y_train_aug.T[:, i:i+batch_size]
        i += batch_size

        d1w, d1b, d2w, d2b = backward(x_batch, y_batch)

        W1 += lr * d1w
        B1 += lr * d1b

        W2 += lr * d2w
        B2 += lr * d2b

        a2 = forward(x_batch)
        batch_loss.append(np.linalg.norm(a2 - y_batch))

    loss_history.append(np.mean(batch_loss))
    hist_W1.append(W1.flatten())
    hist_B1.append(B1.flatten())
    hist_W2.append(W2.flatten())
    hist_B2.append(B2.flatten())
```

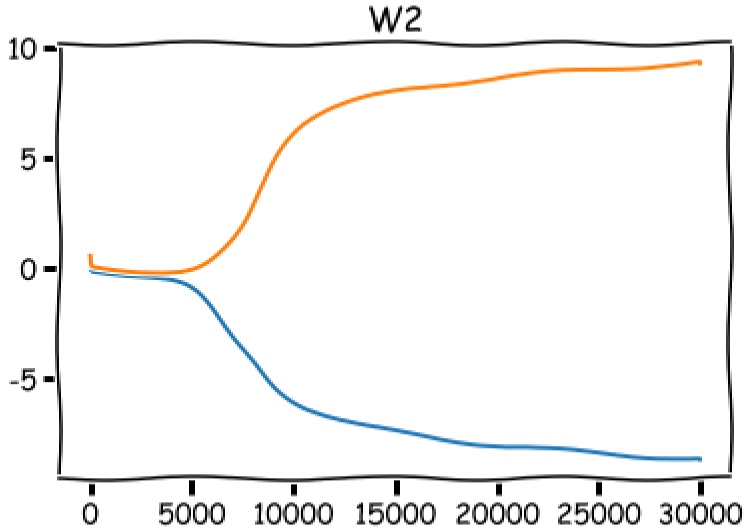
In [15]:

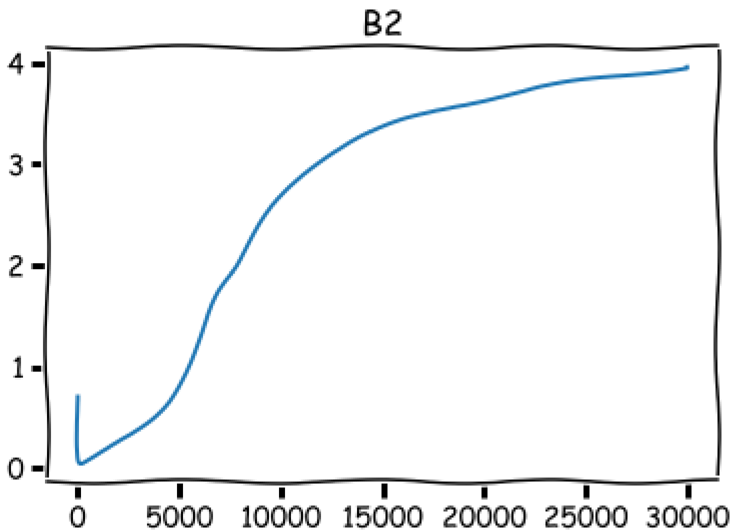
```
with plt.xkcd():
    plt.plot(loss_history)
    plt.title('loss')
    plt.show()

    plt.plot(hist_W1)
    plt.title('W1')
    plt.show()

    plt.plot(hist_B1)
    plt.title('B1')
    plt.show()

    plt.plot(hist_W2)
    plt.title('W2')
    plt.show()

    plt.plot(hist_B2)
    plt.title('B2')
    plt.show()
```


loss
4
3
2
1
0
5000
10000
15000
20000
25000
30000


W1
6
5
4
3
2
1
0
0
5000
10000
15000
20000
25000
30000

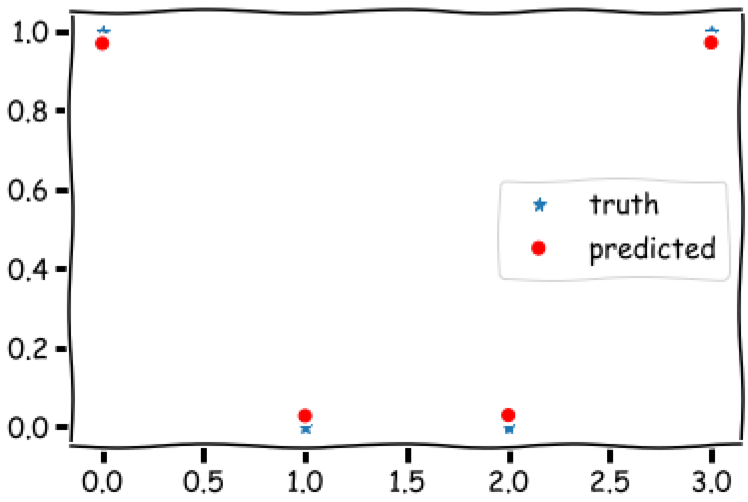

B1
0
-2
-4
-6
0
5000
10000
15000
20000
25000
30000


W2
10
5
0
-5
0
5000
10000
15000
20000
25000
30000


B2
4
3
2
1
0
0
5000
10000
15000
20000
25000
30000

In [16]:

```
M = x_train_aug.shape[0]
xx = x_train_aug[:, 0] + x_train_aug[:, 1] * 2
xx = np.reshape(xx, (M, 1)).transpose()

y_hat = forward(x_train_aug.T)

with plt.xkcd():
    plt.plot(xx.T, y_train_aug, '*', label='truth')
    plt.plot(xx.T, y_hat.T, 'ro', label='predicted')
    plt.legend()
    plt.show()

for x_i, y_i in zip(x_train, y_train):
    a = forward(np.reshape(x_i, (2, 1)))
    r = 1 * (a > 0.5)
    print x_i, '~~>', '%d (=%.2f)' % (r, a), '[%d]' % y_i
```




```
[0 0] ~~> 1 (=0.97) [1]
[0 1] ~~> 0 (=0.03) [0]
[1 0] ~~> 0 (=0.03) [0]
[1 1] ~~> 1 (=0.97) [1]
```

In the next section we will optimize this code to have more felixible neural network and try to run it on more advanced datasets. Please follow up with this [link (https://github.com/ArefMq/simple-nn/blob/master/Day-2.ipynb)](https://github.com/ArefMq/simple-nn/blob/master/Day-2.ipynb).

Day 2 - Clean Network

In this section, we haved used codes from privous section and cleaned it up to get a dynamic code that could handle more vriaity of architechtures.

In [1]:

```
import os
import numpy as np
from random import randint
import matplotlib.pyplot as plt
```

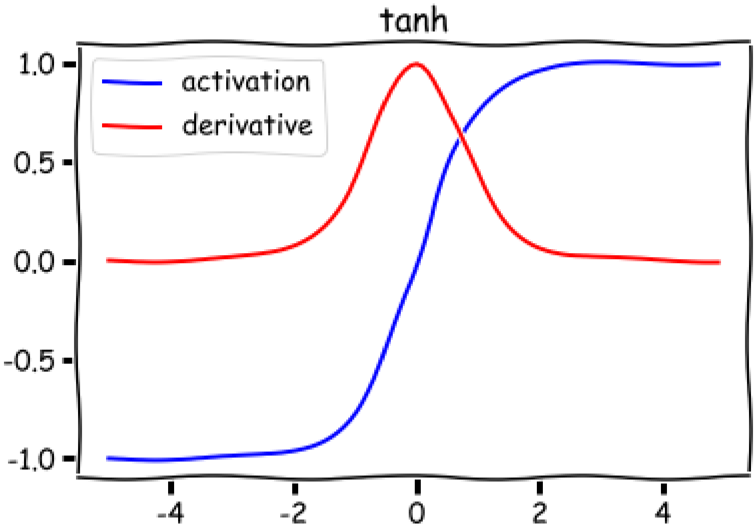
Afterward, we required a better approch toward the activation function and their derivative. So, here we define an abstract class to handle the complexity of this functions.

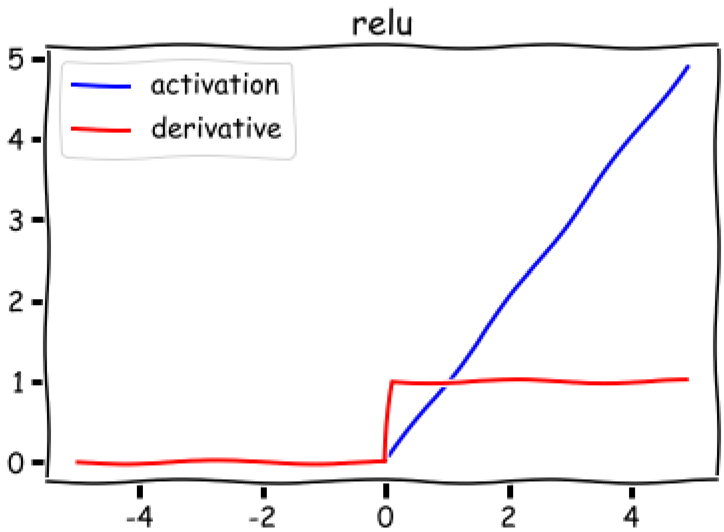
In [2]:

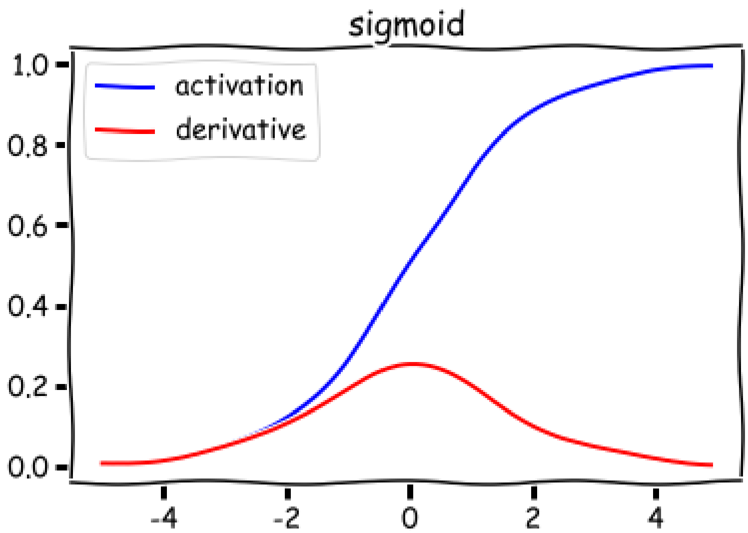
```
from abc import abstractmethod

# TODO: fix abstraction
class ActivationFunction:
    @abstractmethod
    def activate(self, x):
        pass

    @abstractmethod
    def derivative(self, x):
        pass

    def __call__(self, x):
        return self.activate(x)


# And here some sample activation functions
class Sigmoid(ActivationFunction):
    def activate(self, x):
        return 1.0 / (1.0 + np.exp(-x))

    def derivative(self, x):
        return self.activate(x) * (1.0 - self.activate(x))

class ReLU(ActivationFunction):
    def activate(self, x):
        return x * (x > 0)

    def derivative(self, x):
        return 1.0 * (x > 0)


class tanh(ActivationFunction):
    def activate(self, x):
        return np.tanh(x)

    def derivative(self, x):
        return 1.0 / np.cosh(x) ** 2


ACTIVATION_FUNCTIONS = {
    'sigmoid': Sigmoid(),
    'relu': ReLU(),
    'tanh': tanh(),
}

def get_activation_function(actv_func):
    if isinstance(actv_func, str):
        if actv_func not in ACTIVATION_FUNCTIONS:
            raise Exception('activation "%s" not found' % actv_func)
        actv_func = ACTIVATION_FUNCTIONS[actv_func]
    return actv_func
```

In [3]:

```
%matplotlib inline

# List Activation Functions
with plt.xkcd():
    x = np.array(list(range(-50, +50))) / 10.0
    for name, func in ACTIVATION_FUNCTIONS.items():
        plt.plot(x, func(x), 'b', label='activation')
        plt.plot(x, func.derivative(x), 'r', label='derivative')
        plt.title(name)
        plt.legend()
        plt.show()
```







We have defined a `Layer` class which represent each layers of the network. Each layer is defined by a `n` representing its number of neurons and a `actv_func` (short for activation-function). Then each layer has three functionality, either they can perform a forward propagation, a backward propagation, or they can optimize theyr weights.

In [4]:

```
class Layer:
    def __init__(self, n, prev_n, actv_func):
        self.actv_func = get_activation_function(actv_func)
        self.n = n
        self.prev_n = prev_n
        self.initialize()

    def initialize(self):
        self.w = np.random.uniform(low=0, high=+1, size=(self.n, self.prev_n))
        self.b = np.random.uniform(low=0, high=+1, size=(self.n, 1))

        # These parameters will be used in backprop
        self.x0 = 0
        self.z0 = 0
        self.dw = 0
        self.db = 0

        # Debug plot
        self.hist_w = []
        self.hist_b = []

    def set_params(self, new_w, new_b, new_func=None):
        if new_w.shape != self.w.shape:
            raise Exception('weight size mismatch. Expecting %s but got %s' % (s
elf.w.shape, new_w.shape))
        if new_b.shape != self.b.shape:
            raise Exception('bias size mismatch. Expecting %s but got %s' % (sel
f.b.shape, new_b.shape))

        self.w = new_w
        self.b = new_b
        if new_func is not None:
            self.actv_func = get_activation_function(new_func)

    def forward(self, x):
        z = self.w.dot(x) + self.b
        a = self.actv_func(z)

        self.z0 = z
        self.x0 = x

        return a

    def backward(self, error, m):
        delta = error * self.actv_func.derivative(self.z0)
        self.dw = delta.dot(self.x0.T) / float(m)
        self.db = delta.dot(np.ones((m,1))) / float(m)
        return self.w.T.dot(delta)

    def optimize_weights(self, eta):
        self.w += eta * self.dw
        self.b += eta * self.db

        self.hist_w.append(self.w.flatten())
        self.hist_b.append(self.b.flatten())
```

Afterward, we have defended a class called `network`. This class is responsible for connecting the layers together both in forward and backward propagations.

In [5]:

```
class Network:
    def __init__(self, input_size):
        self.layers = []
        self.last_layer_size = input_size
        self.lr = 0.01
        self.initialize()

    def add_layer(self, n, activation='sigmoid'):
        self.layers.append(Layer(
            n,
            self.last_layer_size,
            activation
        ))
        self.last_layer_size = n

    def predict(self, x0):
        z = x0
        for l in self.layers:
            z = l.forward(z)
        return z

    def backpropagate(self, x0, y0):
        m = x0.shape[1]
        y_hat = self.predict(x0)
        error = y0-y_hat

        for i in reversed(range(len(self.layers))):
            error = self.layers[i].backward(error, m)

        for i in range(len(self.layers)):
            self.layers[i].optimize_weights(self.lr)

    def initialize(self):
        self.loss_history = []
        for l in self.layers:
            l.initialize()

    def train(self, x, y, batch_size, epochs, lr=None, initialize=False):
        if initialize:
            self.initialize()

        if lr is not None:
            self.lr = lr

        for e in range(epochs):
            i=0
            batch_loss = []
            while(i<x.shape[1]):
                x_batch = x[:, i:i+batch_size]
                y_batch = y[:, i:i+batch_size]
                i += batch_size

                self.backpropagate(x_batch, y_batch)
                batch_loss.append(np.linalg.norm(self.predict(x_batch) - y_batch
))

            self.loss_history.append(np.mean(batch_loss))

    def plot_loss(self, weight_history=False):
```

```
        with plt.xkcd():
            plt.plot(nn.loss_history)
            plt.title('loss')
            plt.show()

            for i, l in enumerate(self.layers):
                plt.plot(l.hist_w)
                plt.title('W%d' % (i+1))
                plt.show()

                plt.plot(l.hist_b)
                plt.title('B%d' % (i+1))
                plt.show()
```

Before attempting to creating and training the network, we first needs some data.

In [6]:

```
# creating the dataset
def xor(x, y):
    return 1 if x+y != 1 else 0

x_train_aug = []
y_train_aug = []

for _ in range(1024):
    a = randint(0, 1)
    b = randint(0, 1)
    x_train_aug.append([a, b])
    y_train_aug.append([xor(a, b)])

x_train_aug = np.array(x_train_aug)
y_train_aug = np.array(y_train_aug)

x_train = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1],
])

y_train = np.array([xor(i[0], i[1]) for i in x_train])
```

In [7]:

```
def plot_prediction():
    M = x_train_aug.shape[0]
    xx = x_train_aug[:, 0] + x_train_aug[:, 1] * 2
    xx = np.reshape(xx, (M, 1)).transpose()

    y_hat = nn.predict(x_train_aug.T)

    with plt.xkcd():
        plt.plot(xx.T, y_train_aug, '*', label='truth')
        plt.plot(xx.T, y_hat.T, 'ro', label='predicted')
        plt.legend()
        plt.show()

    for x_i, y_i in zip(x_train, y_train):
        a = nn.predict(np.reshape(x_i, (2, 1)))
        r = 1 * (a > 0.5)
        print x_i, '~~>', '%d (=%.2f)' % (r, a), '[%d]' % y_i
```

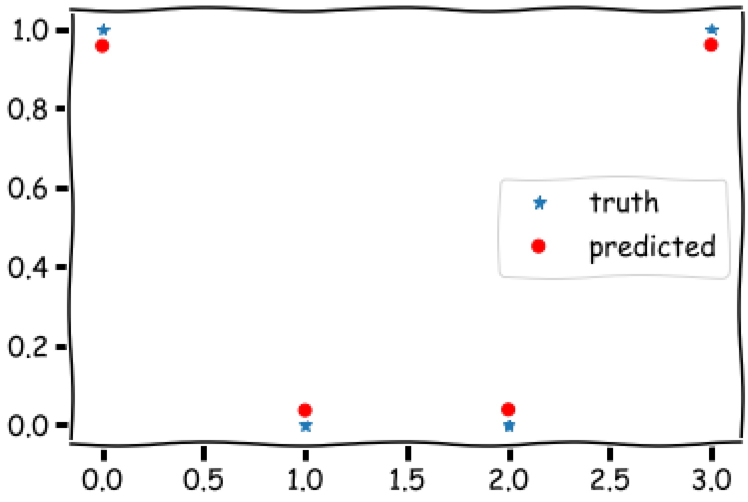
Here is definition of a simple neural network that used in the previous section followed by training it. This network has two layer ,hidden and output, with neuron size of 2 and 1 respectivly.

In [8]:

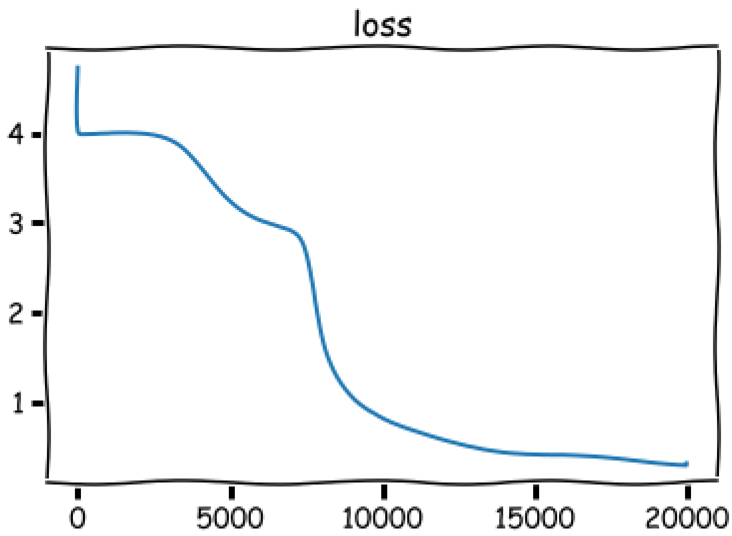
```
nn = Network(input_size=2)
nn.add_layer(2)
nn.add_layer(1)
```

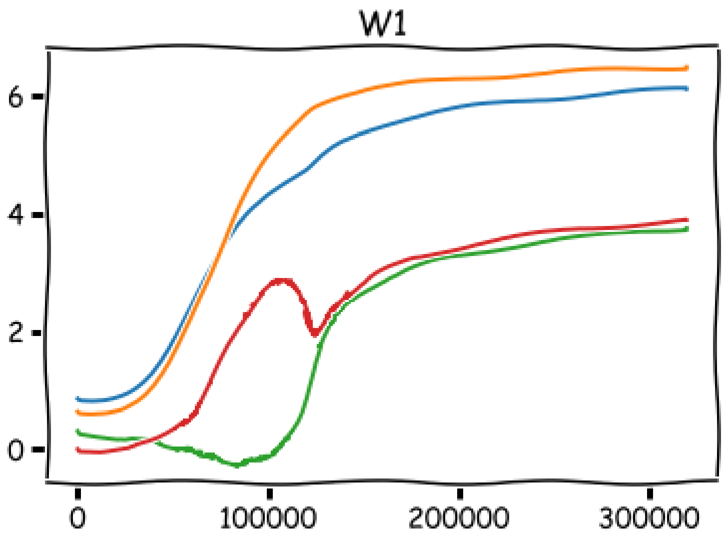
In [9]:

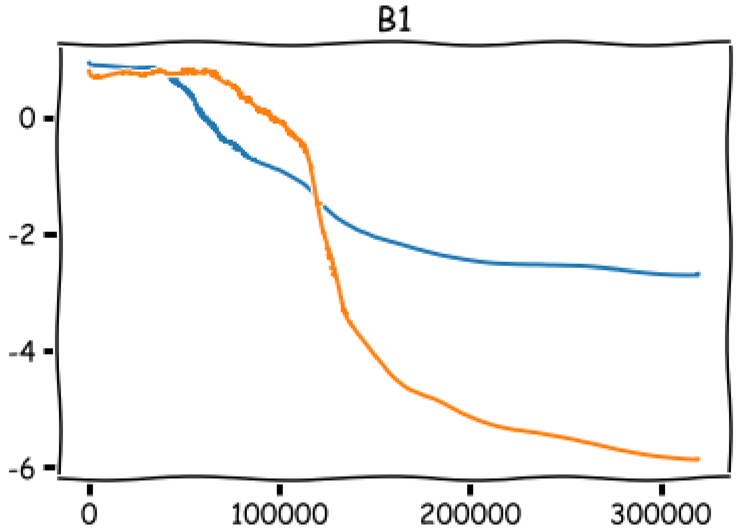
```
nn.train(x_train_aug.T, y_train_aug.T, epochs=20000, batch_size=64, lr=.02)
plot_prediction()
nn.plot_loss(weight_history=True)
```

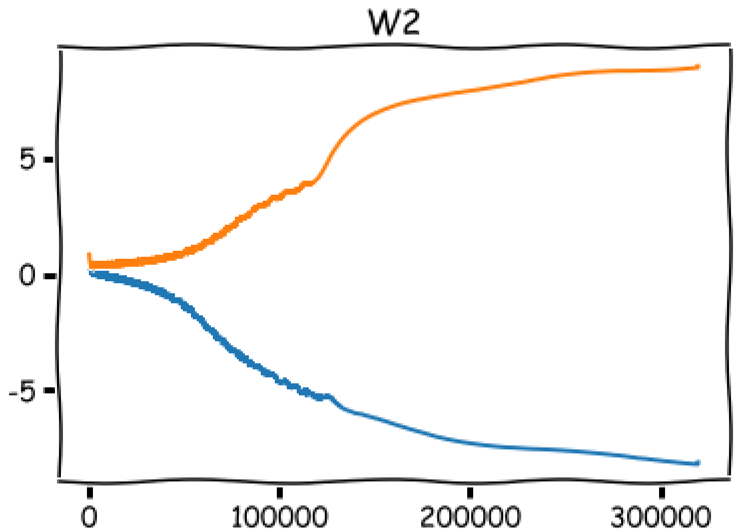


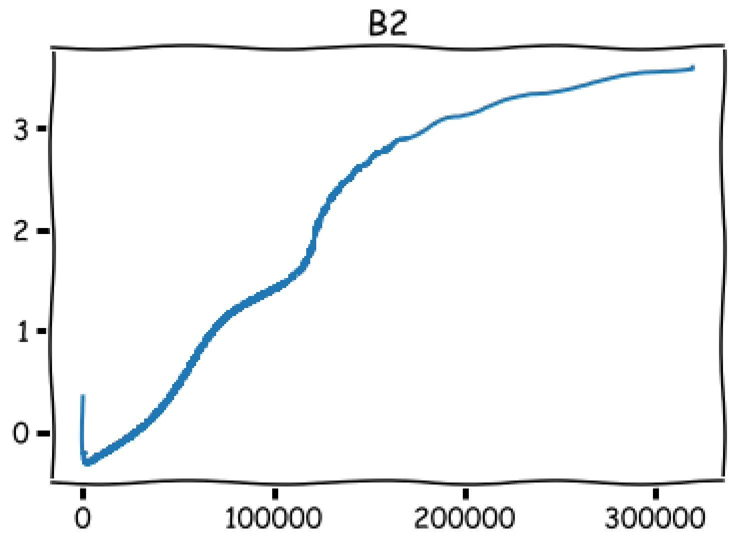

```
[0 0] ~~> 1 (=0.96) [1]
[0 1] ~~> 0 (=0.04) [0]
[1 0] ~~> 0 (=0.04) [0]
[1 1] ~~> 1 (=0.96) [1]
```






B1
0
-2
-4
-6
0
100000
200000
300000


W2
5
0
-5
0
100000
200000
300000


B2
3
2
1
0
0
100000
200000
300000

In the next section we will run this code on more advance datasets and try to classify images. Please follow up with this [link (https://github.com/ArefMq/simple-nn/blob/master/Day-3.ipynb)](https://github.com/ArefMq/simple-nn/blob/master/Day-3.ipynb).

In [84]:

```
import os
import numpy as np
from random import randint
import matplotlib.pyplot as plt

from simple_nn import Network

np.seterr(all='raise')

try:
    from notify import notify
except ImportError:
    def notify(*msgs, **kwargs):
        print ' '.join([str(m) for m in msgs])

EPSILLON = 1e-10
```

In [85]:

```
def flatten(input_value):
    return input_value.reshape(input_value.shape[0], -1)

def normalize(input_value):
    res = input_value - np.min(input_value)
    return res / np.max(res)

def image_preprocess(image_data):
    return normalize(flatten(image_data))
```

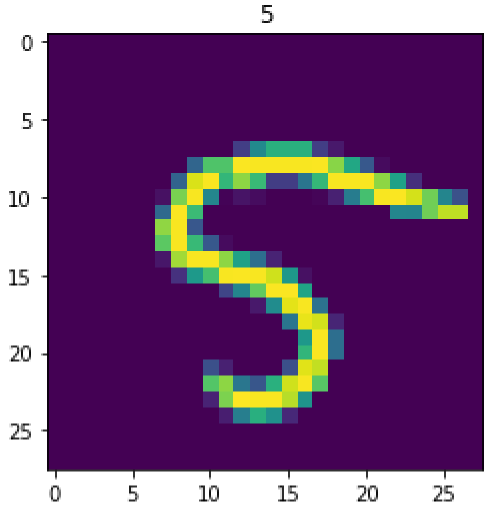
In [86]:

```
# Loading dataset
import torchvision
root = os.path.expanduser("./datasets/mnist")
train_dataset = torchvision.datasets.MNIST(root, train=True, transform=None, tar
get_transform=None, download=True)
test_dataset = torchvision.datasets.MNIST(root, train=False, transform=None, tar
get_transform=None, download=True)

x_train = np.array(train_dataset.train_data, dtype=np.float)
x_test = np.array(test_dataset.test_data, dtype=np.float)
y_train = np.array(train_dataset.train_labels, dtype=np.int)
y_test  = np.array(test_dataset.test_labels, dtype=np.int)
```

In [87]:

```
%matplotlib inline

# Show details about the dataset
print 'x_train shape:', x_train.shape
print 'x_train shape:', x_test.shape

i = randint(0, x_train.shape[0]-1)
plt.imshow(x_train[i, ...])
plt.title(y_train[i])
plt.show()
```

```
x_train shape: (60000, 28, 28)
x_train shape: (10000, 28, 28)
```



In [88]:

```
def to_categorical(y_data):
    l = np.max(y_data) - np.min(y_data) + 1
    res = np.zeros((y_data.shape[0], l))
    for i in range(y_data.shape[0]):
        res[i, y_data[i]] = 1
    return res
```

In [89]:

```
x_train_processed = image_preprocess(x_train)#[:100, :]
x_test_processed = image_preprocess(x_test)#[:100, :]

y_train_processed = to_categorical(y_train)#[:100, :]
y_test_processed = to_categorical(y_test)#[:100, :]

print 'x_train', x_train.shape
print 'x_train_processed', x_train_processed.shape
print ''
print 'y_train', y_train.shape
print 'y_train_processed', y_train_processed.shape
```

```
x_train (60000, 28, 28)
x_train_processed (60000, 784)

y_train (60000,)
y_train_processed (60000, 10)
```

In [90]:

```
def check_acc(dataset_x, dataset_y):
    m = dataset_y.shape[0]

    y_hat_test = nn.predict(dataset_x.T)

    error = np.zeros(m)
    for i in range(m):
        error[i] = np.argmax(y_hat_test[:, i]) == np.argmax(dataset_y.T[:, i])
    return np.sum(error) / len(error)
```

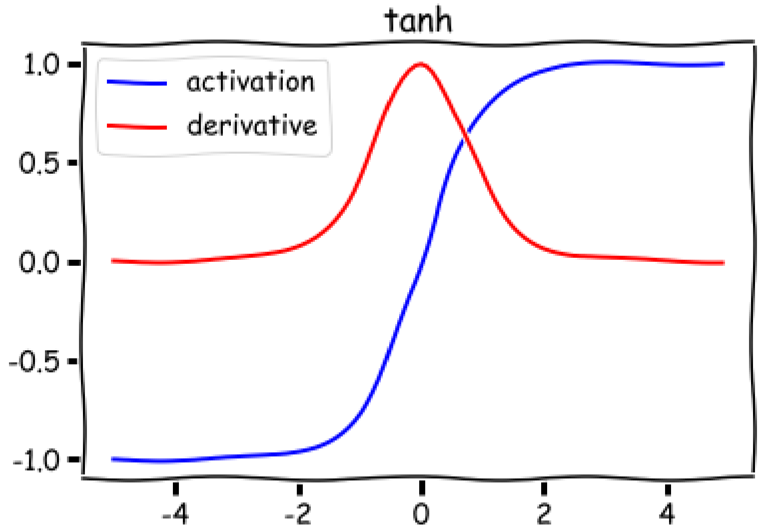
In [91]:

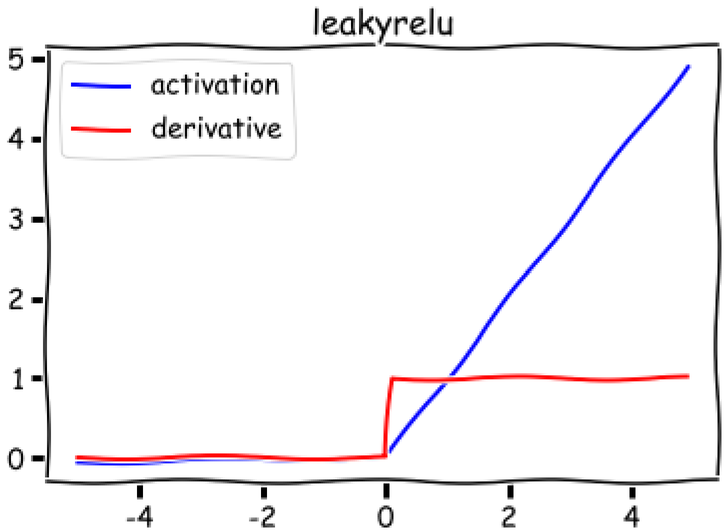
```
from abc import abstractmethod


def flatten(input_value):
    return input_value.reshape(input_value.shape[0], -1)

def normalize(input_value):
    res = input_value - np.min(input_value)
    return res / np.max(res)

def image_preprocess(image_data):
    return normalize(flatten(image_data))

# TODO: fix abstraction
class ActivationFunction:
    @abstractmethod
    def activate(self, x):
        pass

    @abstractmethod
    def derivative(self, x):
        pass

    def __call__(self, x):
        return self.activate(x)


# And here some sample activation functions
class Sigmoid(ActivationFunction):
    def activate(self, x):
        '''
        Numerically stable sigmoid function. instead of:
            return 1.0 / (1.0 + np.exp(-x))"
        '''
        try:
            x = np.clip(x, -30, +30)
            return np.where(x >= 0,
                    1. / (1. + np.exp(-x)),
                    np.exp(x) / (1. + np.exp(x)))
        except:
            print 'mean', np.mean(x)
            print 'mmx ', np.min(x), '~>', np.max(x)
            raise

    def derivative(self, x):
        return self.activate(x) * (1. - self.activate(x))


class ReLU(ActivationFunction):
    def activate(self, x):
        return x * (x > 0)

    def derivative(self, x):
        return 1.0 * (x > 0)


class LeakyReLU(ActivationFunction):
    def activate(self, x):
        return np.where(x > 0, x, 0.01*x)
```

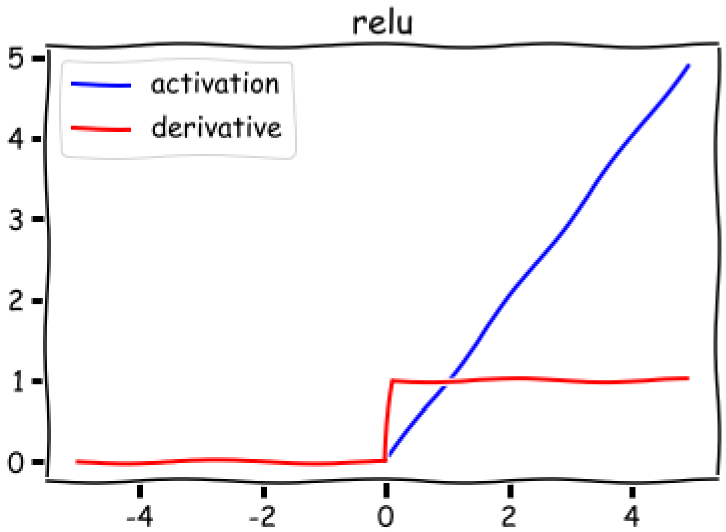
```python
    def derivative(self, x):
        return np.where(x > 0, 1, 0.01)


class tanh(ActivationFunction):
    def activate(self, x):
        return np.tanh(x)

    def derivative(self, x):
        return 1.0 / np.cosh(x) ** 2


ACTIVATION_FUNCTIONS = {
    'sigmoid': Sigmoid(),
    'relu': ReLU(),
    'tanh': tanh(),
    'leakyrelu': LeakyReLU(),
}

def get_activation_function(actv_func):
    if isinstance(actv_func, str):
        if actv_func not in ACTIVATION_FUNCTIONS:
            raise Exception('activation "%s" not found' % actv_func)
        actv_func = ACTIVATION_FUNCTIONS[actv_func]
    return actv_func
```

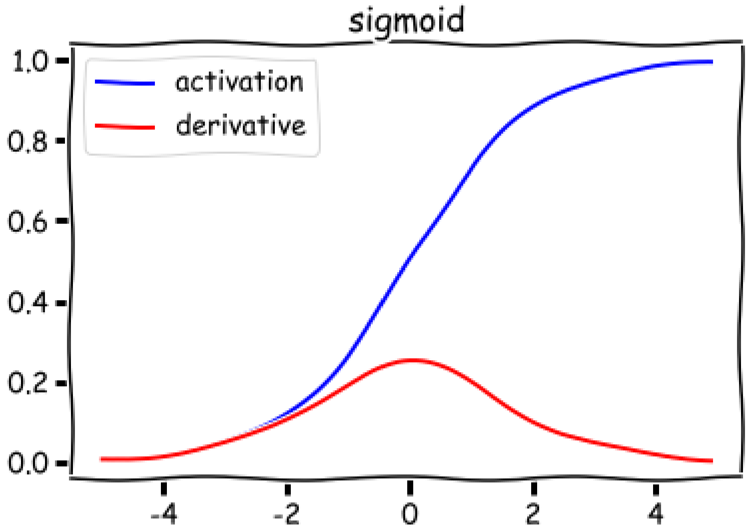
In [92]:

```
%matplotlib inline

# List Activation Functions
with plt.xkcd():
    x = np.array(list(range(-50, +50))) / 10.0
    for name, func in ACTIVATION_FUNCTIONS.items():
        plt.plot(x, func(x), 'b', label='activation')
        plt.plot(x, func.derivative(x), 'r', label='derivative')
        plt.title(name)
        plt.legend()
        plt.show()
```


tanh
activation
derivative
1.0
0.5
0.0
-0.5
-1.0
-4
-2
0
2
4


leakyrelu
activation
derivative
5
4
3
2
1
0
-4
-2
0
2
4


relu
activation
derivative
5
4
3
2
1
0
-4
-2
0
2
4


sigmoid
activation
derivative
1.0
0.8
0.6
0.4
0.2
0.0
-4
-2
0
2
4

In [93]:

```
NEURON_CLIP = 900.

class Layer:
    def __init__(self, n, prev_n, **kwargs):
        self.actv_func_name = kwargs.get('activation', 'sigmoid')
        self.actv_func = get_activation_function(self.actv_func_name)

        self.normalized = kwargs.get('normalized', False)
        self.keep_prob = kwargs.get('keep_prob', None)

        self.n = n
        self.prev_n = prev_n
        self.initialize()

    def initialize(self):
        # FIXME: fix here
        if self.actv_func_name in ['relu', 'leakyrelu']:
            self.w = np.random.normal(loc=0.5, scale=0.1, size=(self.n, self.pre
v_n))
            self.b = np.random.normal(loc=0.5, scale=0.1, size=(self.n, 1))
        else:
            self.w = np.random.normal(loc=0, scale=0.1, size=(self.n, self.prev_
n))
            self.b = np.random.normal(loc=0, scale=0.1, size=(self.n, 1))

        # These parameters will be used in backprop
        self.x0 = 0
        self.z0 = 0
        self.dw = 0
        self.db = 0
        self.moment_dw = 0
        self.moment_db = 0

        # normalization
        self.mu = 0
        self.var = 1
        self.beta = 0
        self.gamma = 1

        # Debug plot
        self.random_weight_selector = np.random.random(self.w.shape) < (10./(sel
f.w.shape[0]*self.w.shape[1]))
        self.random_bias_selector = np.random.random(self.b.shape) < (10./self.b
.shape[0])
        self.hist_w = []
        self.hist_b = []

    def set_params(self, new_w, new_b, new_func=None):
        if new_w.shape != self.w.shape:
            raise Exception('weight size mismatch. Expecting %s but got %s' % (s
elf.w.shape, new_w.shape))
        if new_b.shape != self.b.shape:
            raise Exception('bias size mismatch. Expecting %s but got %s' % (sel
f.b.shape, new_b.shape))

        self.w = new_w
        self.b = new_b
        if new_func is not None:
            self.actv_func = get_activation_function(new_func)
```

```
    def normalize(self, vec, set_value=False):
        if set_value:
            EXP_AVG_COEFF = 0.7
            self.mu = self.mu * EXP_AVG_COEFF + np.mean(vec, axis=1) * (1-EXP_AV
G_COEFF)
            self.var = self.var * EXP_AVG_COEFF + np.var(vec, axis=1) * (1-EXP_A
VG_COEFF)

        norm = (vec.T - self.mu) / np.sqrt(self.var + EPSILLON)
        return norm.T

    def batchnorm_backward(self, error):
        X, X_norm, mu, var, gamma, beta = cache

        N, D = X.shape

        X_mu = X - self.mu
        std_inv = 1. / np.sqrt(self.var + EPSILLON)

        dX_norm = dout * self.gamma
        dvar = np.sum(dX_norm * X_mu, axis=0) * -.5 * std_inv**3
        dmu = np.sum(dX_norm * -std_inv, axis=0) + dvar * np.mean(-2. * X_mu, ax
is=0)

        dX = (dX_norm * std_inv) + (dvar * 2 * X_mu / N) + (dmu / N)
        dgamma = np.sum(error * X_norm, axis=0)
        dbeta = np.sum(error, axis=0)

        return dX, dgamma, dbeta

    def forward(self, x, is_in_backprop):
        z = self.w.dot(x) + self.b
        z = np.clip(z, -NEURON_CLIP, +NEURON_CLIP)
        if self.normalized:
            z = self.normalize(z, is_in_backprop) * self.gamma + self.beta

        a = self.actv_func(z)

        self.z0 = z
        self.x0 = x

        return a

    def backward(self, error, reg_lambda, m):
        delta = error * self.actv_func.derivative(self.z0)
        self.dw = delta.dot(self.x0.T) / float(m) - reg_lambda * self.w / float(
m)
        self.db = delta.dot(np.ones((m,1))) / float(m) - reg_lambda * self.b / f
loat(m)
        return self.w.T.dot(delta)

    def optimize_weights(self, eta, momentom=0.8):
        self.moment_dw = self.moment_dw * momentom + self.dw * (1-momentom)
        self.moment_db = self.moment_db * momentom + self.db * (1-momentom)

        self.w += eta * self.moment_dw
        self.b += eta * self.moment_db

        self.hist_w.append(self.w[self.random_weight_selector].flatten())
```

```
        self.hist_b.append(self.b[self.random_bias_selector].flatten())
```

In [94]:

```
class Network:
    def __init__(self, input_size):
        self.layers = []
        self.last_layer_size = input_size
        self.lr = 0.01
        self.regularization_lambda = 0.01
        self.initialize()

    def add_layer(self, n, activation='sigmoid', **kwargs):
        self.layers.append(Layer(
            n,
            self.last_layer_size,
            activation=activation,
            **kwargs
        ))
        self.last_layer_size = n

    def predict(self, x0, is_in_backprop=False):
        z = x0
        for l in self.layers:
            z = l.forward(z, is_in_backprop)
        return z

    def backpropagate(self, x0, y0):
        m = x0.shape[1]
        y_hat = self.predict(x0, True)
        error = self.loss_function(y0, y_hat)

        for i in reversed(range(len(self.layers))):
            error = self.layers[i].backward(error, self.regularization_lambda, m
)

        for i in range(len(self.layers)):
            self.layers[i].optimize_weights(self.lr)

    def initialize(self):
        self.train_loss_history = []
        self.test_loss_history = []
        for l in self.layers:
            l.initialize()

    def loss_function(self, y0, y_hat):
        regularization_term = self.regularization_lambda * sum([
#               np.sum(np.square(l.w)) + np.sum(np.square(l.b)) for l in self.laye
rs
            np.sum(np.square(l.w)) for l in self.layers
        ]) / (2. * y0.shape[1])

        if y0.shape[0] == 1:
            return (y0-y_hat) + regularization_term
        else:
            return y0*np.log(y_hat+EPSILLON) + (1-y0)*np.log(1-y_hat+EPSILLON) +
regularization_term

    def train(self, x, y, epochs, **kwargs):
        if kwargs.get('initialize', False):
            self.initialize()

        if 'test_data' in kwargs:
```

```
            x_test = kwargs['test_data'][0]
            y_test = kwargs['test_data'][1]
            test_data = True
        else:
            test_data = False

        self.lr = kwargs.get('lr', self.lr)
        batch_size = kwargs.get('batch_size', 32)

        for e in range(epochs):
            i = 0; print_counter = 0
            train_batch_loss = []
            while(i<x.shape[1]):
                x_batch = x[:, i:i+batch_size]
                y_batch = y[:, i:i+batch_size]
                i += batch_size
                print_counter += 1

                self.backpropagate(x_batch, y_batch)
                train_batch_loss.append(np.linalg.norm(self.loss_function(self.p
redict(x_batch), y_batch)))

            loss =  np.mean(train_batch_loss)
            self.train_loss_history.append(loss)

            # calculate the test acc
            if test_data:
                i=0
                test_batch_loss = []
                while(i<x_test.shape[1]):
                    x_batch = x_test[:, i:i+batch_size]
                    y_batch = y_test[:, i:i+batch_size]
                    i += batch_size
                    test_batch_loss.append(np.linalg.norm(self.loss_function(sel
f.predict(x_batch), y_batch)))
                self.test_loss_history.append(np.mean(test_batch_loss))


            if np.isnan(loss):
                raise Exception('loss is NaN')


            if kwargs.get('verbose', True) and (epochs <= 10 or e % int(epochs/1
0) == 0):
                if kwargs.get('visualize_progress', False):
                    self.plot_loss()
                print 'epoch: %d/%d - loss=%.2f' % (e+1, epochs, loss)

                # TODO: remove this
                for i in range(len(self.layers)):
                    c = self.layers[i].dw.shape
#                     print '     dw%d:' % i, np.sum(self.layers[i].dw) / (c[0]
 * c[1])
#                     print '      w%d:' % i, np.sum(self.layers[i].w) / (c[0] *
c[1])
#                     print ''


    def plot_loss(self):
        with plt.xkcd():
            plt.plot(self.train_loss_history, label='train')
```

```python
            if len(self.test_loss_history) > 0:
                plt.plot(self.test_loss_history, 'r--', label='test')
                plt.legend()
            plt.title('loss')
            plt.show()

            for i, l in enumerate(self.layers):
                plt.plot(l.hist_w)
                plt.title('W%d' % (i+1))
                plt.show()

                plt.plot(l.hist_b)
                plt.title('B%d' % (i+1))
                plt.show()
```

In [95]:

```python
# creating the dataset
def xor(x, y):
    return 1 if x+y != 1 else 0

xor_x_train_aug = []
xor_y_train_aug = []

for _ in range(1024):
    a = randint(0, 1)
    b = randint(0, 1)
    xor_x_train_aug.append([a, b])
    xor_y_train_aug.append([xor(a, b)])

xor_x_train_aug = np.array(xor_x_train_aug)
xor_y_train_aug = np.array(xor_y_train_aug)

xor_x_train_test = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1],
])

xor_y_train_test = np.array([xor(i[0], i[1]) for i in xor_x_train_test])
```

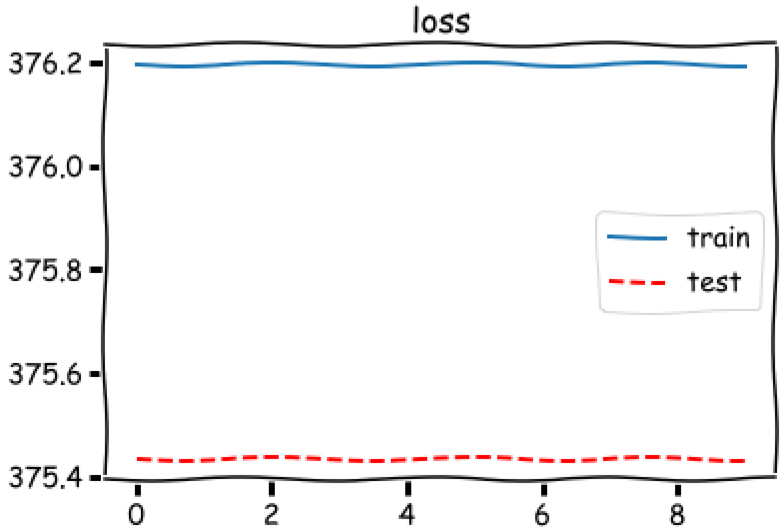
In [96]:

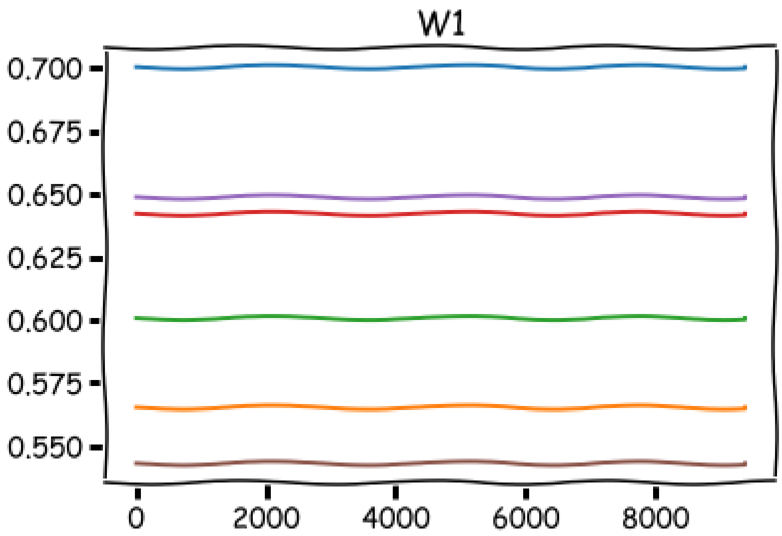
```
def plot_prediction():
    M = xor_x_train_aug.shape[0]
    xx = xor_x_train_aug[:, 0] + xor_x_train_aug[:, 1] * 2
    xx = np.reshape(xx, (M, 1)).transpose()

    y_hat = nn.predict(xor_x_train_aug.T)

    with plt.xkcd():
        plt.plot(xx.T, xor_y_train_aug, '*', label='truth')
        plt.plot(xx.T, y_hat.T, 'ro', label='predicted')
        plt.legend()
        plt.show()

    for x_i, y_i in zip(xor_x_train_test, xor_y_train_test):
        a = nn.predict(np.reshape(x_i, (2, 1)))
        r = 1 * (a > 0.5)
        print x_i, '~~>', '%d (=%.2f)' % (r, a), '[%d]' % y_i
```

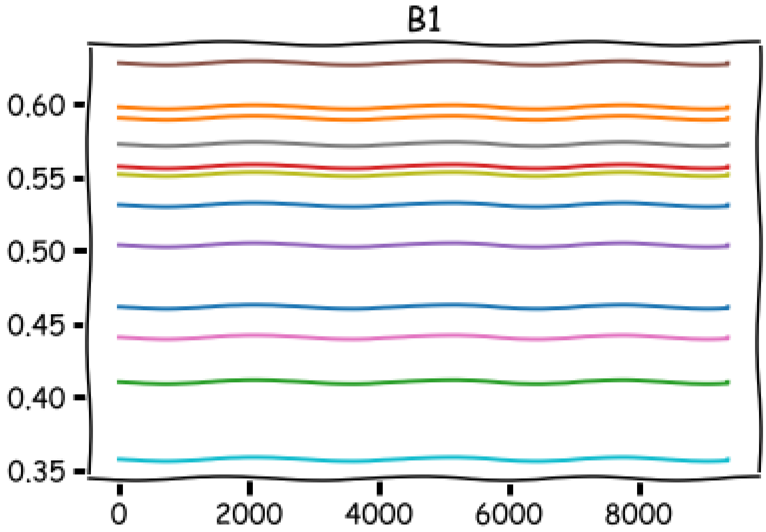
In [99]:

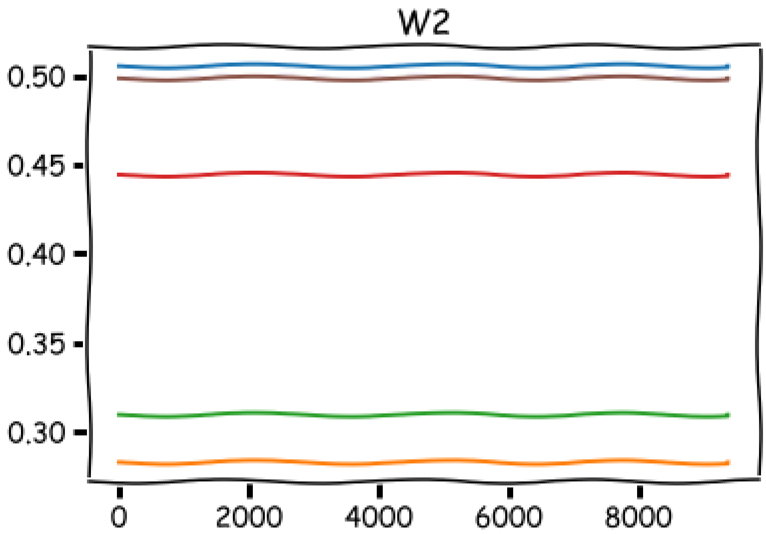
```
nn = Network(input_size=x_train_processed.shape[1])
nn.add_layer(1024, 'relu')
nn.add_layer(1024, 'relu')
nn.add_layer(10)

nn.regularization_lambda = 0

nn.train(
    x_train_processed.T,
    y_train_processed.T,
    epochs=10,
    batch_size=64,
#     visualize_progress=True,
    test_data=(x_test_processed.T, y_test_processed.T),
    lr=.01
)
nn.plot_loss()
print 'test acc: %.2f%%' % (check_acc(x_test_processed, y_test_processed) * 100)
print 'train acc: %.2f%%' % (check_acc(x_train_processed, y_train_processed) * 1
00)
```

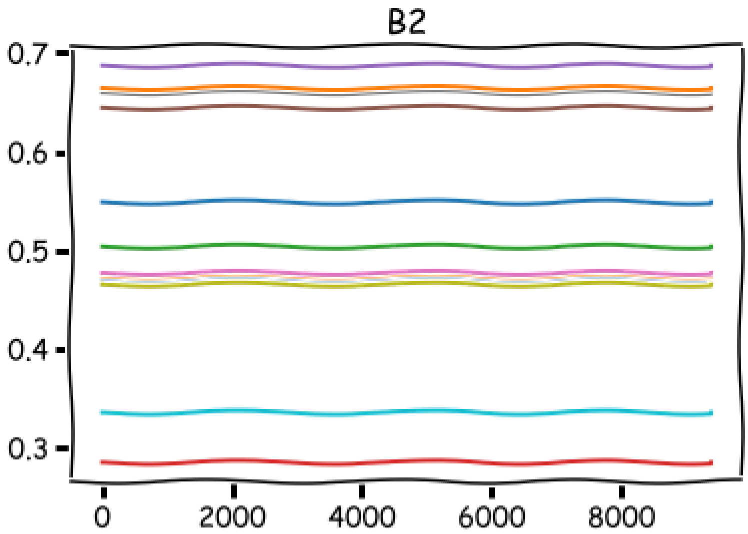
```
epoch: 1/10 - loss=376.20
epoch: 2/10 - loss=376.20
epoch: 3/10 - loss=376.20
epoch: 4/10 - loss=376.20
epoch: 5/10 - loss=376.20
epoch: 6/10 - loss=376.20
epoch: 7/10 - loss=376.20
epoch: 8/10 - loss=376.20
epoch: 9/10 - loss=376.20
epoch: 10/10 - loss=376.20
```

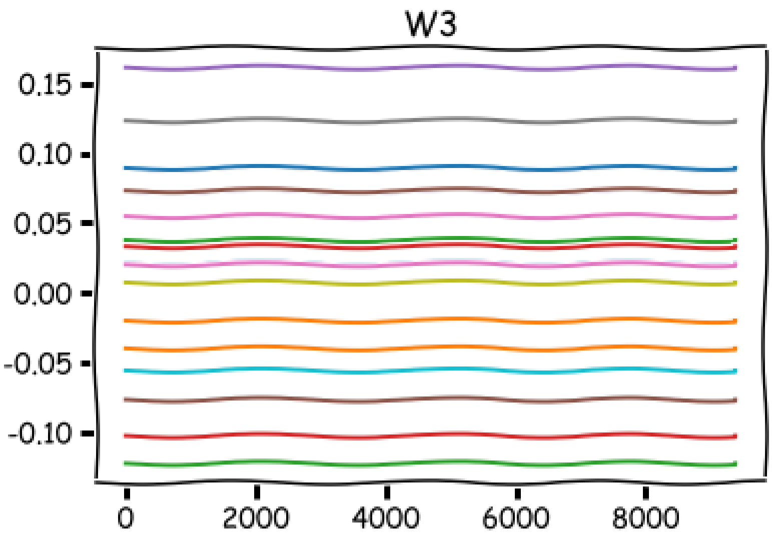


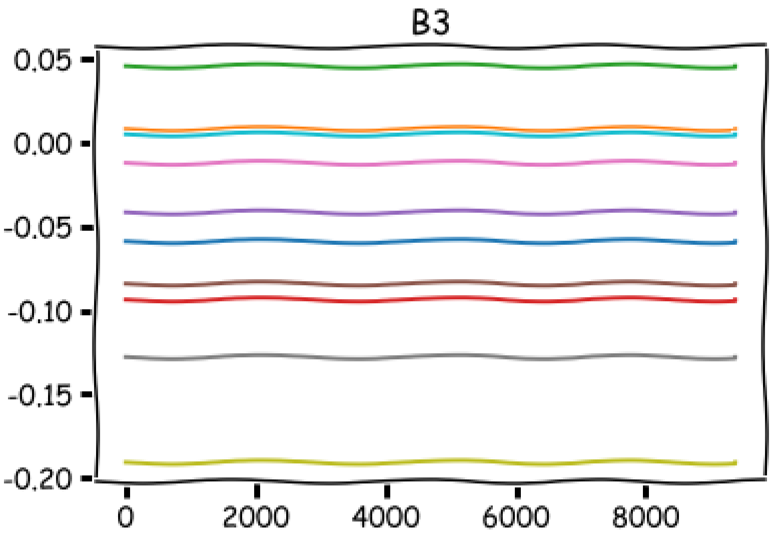



B1
0.60
0.55
0.50
0.45
0.40
0.35
0
2000
4000
6000
8000


W2
0.50
0.45
0.40
0.35
0.30
0
2000
4000
6000
8000


B2
0.7
0.6
0.5
0.4
0.3
0
2000
4000
6000
8000






```
test acc: 11.35%
train acc: 11.24%
```

In the next assignment we try to implement CNN and solve Fashion-MNIST problem with it. Please follow up with this [link (https://github.com/ArefMq/simple-nn/blob/master/Day-5.ipynb)](https://github.com/ArefMq/simple-nn/blob/master/Day-5.ipynb).

Simple CNN

Solving Fashion-MNIST dataset through CNN

In this section, we try to solve the Fashion-MNIST via a CNN and compare it with an MLP network.

In [1]:

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D

from random import randint
```

```
Using TensorFlow backend.
```

First, we normalize the input between 0 and 1. This will result in better and lower weights for the out network.

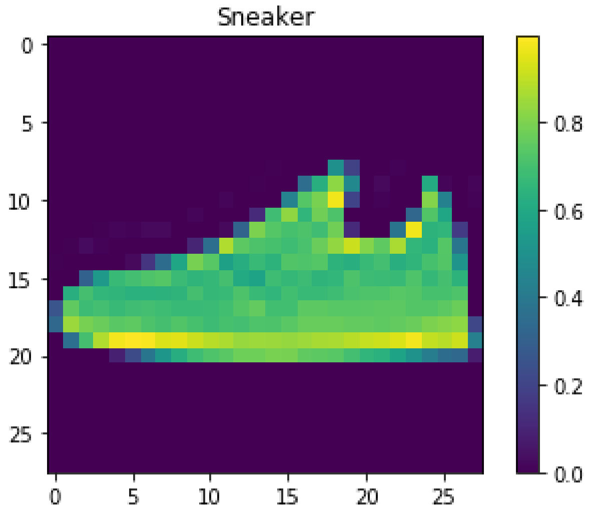
In [2]:

```
dataset = keras.datasets.fashion_mnist
(x_train, y_train), (x_test, y_test) = dataset.load_data()

x_train = x_train / 255.0
x_test = x_test / 255.0


print "x_train.shape =", x_train.shape
print "x_test.shape  =", x_test.shape
```

```
x_train.shape = (60000, 28, 28)
x_test.shape  = (10000, 28, 28)
```

In [3]:

```
%matplotlib inline
def get_label_name(y):
    class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
                   'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
    return class_names[y]

rnd_index = randint(0, x_train.shape[0]-1)
plt.imshow(x_train[rnd_index])
plt.title(get_label_name(y_train[rnd_index]))
plt.colorbar()
plt.show()
```



MLP

Below we have a two-layered MLP network with 128 hidden units. The hidden units have the ReLU activation function. The reason for choosing ReLU the distribution of the input data which are between 0 and 1. The practical tests showed that ReLU works best with images. Another thing to note here is that we have used a 35% dropout connection rate for hidden units which made the performance of the network significantly higher.

In [4]:

```
model = Sequential()

model.add(Flatten(input_shape=(28, 28)))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.35))
model.add(Dense(10, activation='softmax'))

print 'Training...'
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=
['accuracy'])
history = model.fit(
    x_train,
    y_train,
    epochs=45,
    batch_size=128,
#     verbose=0,
    validation_data=(x_test, y_test)
)
print 'Training Finished!'

train_loss, train_acc = model.evaluate(x_train,  y_train, verbose=0)
test_loss, test_acc = model.evaluate(x_test,  y_test, verbose=0)

print ''
print 'Train accuracy:', train_acc
print 'Test accuracy:', test_acc
```

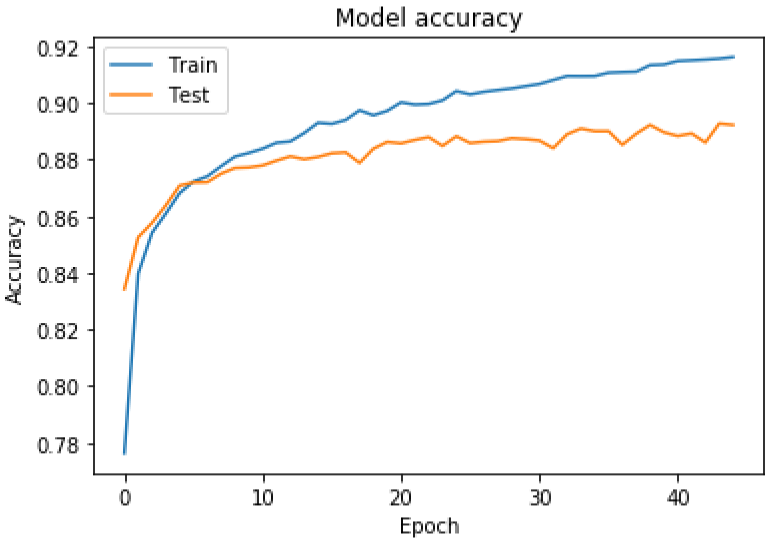
```
Training...
Train on 60000 samples, validate on 10000 samples
Epoch 1/45
60000/60000 [==============================] - 2s 33us/step - loss:
0.6439 - acc: 0.7763 - val_loss: 0.4698 - val_acc: 0.8343
Epoch 2/45
60000/60000 [==============================] - 2s 38us/step - loss:
0.4509 - acc: 0.8397 - val_loss: 0.4193 - val_acc: 0.8526
Epoch 3/45
60000/60000 [==============================] - 2s 36us/step - loss:
0.4088 - acc: 0.8542 - val_loss: 0.3955 - val_acc: 0.8577
Epoch 4/45
60000/60000 [==============================] - 2s 32us/step - loss:
0.3852 - acc: 0.8610 - val_loss: 0.3794 - val_acc: 0.8639
Epoch 5/45
60000/60000 [==============================] - 1s 25us/step - loss:
0.3658 - acc: 0.8682 - val_loss: 0.3611 - val_acc: 0.8709
Epoch 6/45
60000/60000 [==============================] - 3s 49us/step - loss:
0.3539 - acc: 0.8724 - val_loss: 0.3548 - val_acc: 0.8720
Epoch 7/45
60000/60000 [==============================] - 3s 47us/step - loss:
0.3432 - acc: 0.8742 - val_loss: 0.3526 - val_acc: 0.8721
Epoch 8/45
60000/60000 [==============================] - 3s 44us/step - loss:
0.3364 - acc: 0.8777 - val_loss: 0.3538 - val_acc: 0.8751
Epoch 9/45
60000/60000 [==============================] - 3s 43us/step - loss:
0.3277 - acc: 0.8811 - val_loss: 0.3458 - val_acc: 0.8771
Epoch 10/45
60000/60000 [==============================] - 3s 47us/step - loss:
0.3206 - acc: 0.8824 - val_loss: 0.3423 - val_acc: 0.8773
Epoch 11/45
60000/60000 [==============================] - 3s 55us/step - loss:
0.3146 - acc: 0.8839 - val_loss: 0.3370 - val_acc: 0.8780
Epoch 12/45
60000/60000 [==============================] - 3s 48us/step - loss:
0.3110 - acc: 0.8860 - val_loss: 0.3412 - val_acc: 0.8797
Epoch 13/45
60000/60000 [==============================] - 3s 55us/step - loss:
0.3036 - acc: 0.8865 - val_loss: 0.3286 - val_acc: 0.8812
Epoch 14/45
60000/60000 [==============================] - 3s 52us/step - loss:
0.2986 - acc: 0.8894 - val_loss: 0.3299 - val_acc: 0.8802
Epoch 15/45
60000/60000 [==============================] - 3s 56us/step - loss:
0.2915 - acc: 0.8931 - val_loss: 0.3354 - val_acc: 0.8810
Epoch 16/45
60000/60000 [==============================] - 3s 47us/step - loss:
0.2907 - acc: 0.8927 - val_loss: 0.3290 - val_acc: 0.8823
Epoch 17/45
60000/60000 [==============================] - 3s 43us/step - loss:
0.2852 - acc: 0.8941 - val_loss: 0.3276 - val_acc: 0.8826
Epoch 18/45
60000/60000 [==============================] - 4s 66us/step - loss:
0.2795 - acc: 0.8975 - val_loss: 0.3357 - val_acc: 0.8789
Epoch 19/45
60000/60000 [==============================] - 4s 63us/step - loss:
0.2800 - acc: 0.8957 - val_loss: 0.3281 - val_acc: 0.8839
Epoch 20/45
60000/60000 [==============================] - 3s 49us/step - loss:
```

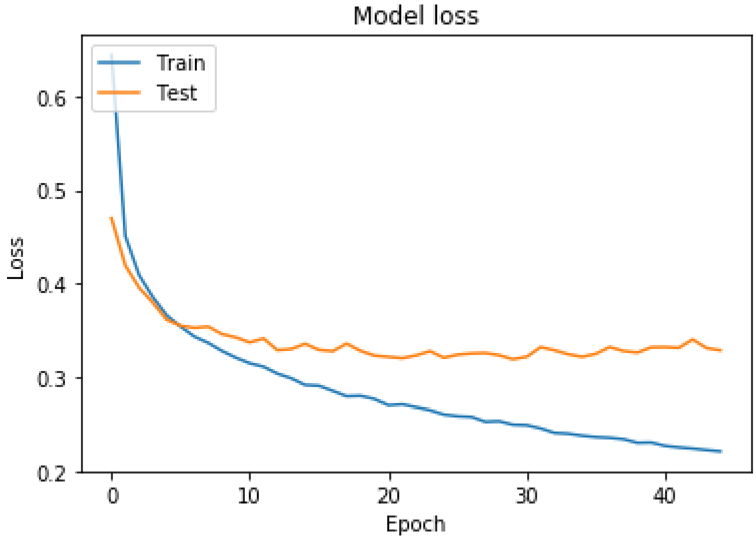
```
0.2767 - acc: 0.8972 - val_loss: 0.3229 - val_acc: 0.8863
Epoch 21/45
60000/60000 [==============================] - 3s 48us/step - loss: 
0.2698 - acc: 0.9003 - val_loss: 0.3215 - val_acc: 0.8858
Epoch 22/45
60000/60000 [==============================] - 3s 48us/step - loss: 
0.2709 - acc: 0.8995 - val_loss: 0.3202 - val_acc: 0.8870
Epoch 23/45
60000/60000 [==============================] - 3s 48us/step - loss: 
0.2679 - acc: 0.8997 - val_loss: 0.3231 - val_acc: 0.8880
Epoch 24/45
60000/60000 [==============================] - 3s 49us/step - loss: 
0.2647 - acc: 0.9010 - val_loss: 0.3275 - val_acc: 0.8849
Epoch 25/45
60000/60000 [==============================] - 3s 46us/step - loss: 
0.2602 - acc: 0.9042 - val_loss: 0.3206 - val_acc: 0.8883
Epoch 26/45
60000/60000 [==============================] - 3s 47us/step - loss: 
0.2584 - acc: 0.9031 - val_loss: 0.3238 - val_acc: 0.8859
Epoch 27/45
60000/60000 [==============================] - 3s 45us/step - loss: 
0.2577 - acc: 0.9040 - val_loss: 0.3252 - val_acc: 0.8864
Epoch 28/45
60000/60000 [==============================] - 3s 50us/step - loss: 
0.2525 - acc: 0.9046 - val_loss: 0.3256 - val_acc: 0.8866
Epoch 29/45
60000/60000 [==============================] - 3s 49us/step - loss: 
0.2531 - acc: 0.9052 - val_loss: 0.3232 - val_acc: 0.8876
Epoch 30/45
60000/60000 [==============================] - 3s 46us/step - loss: 
0.2494 - acc: 0.9060 - val_loss: 0.3190 - val_acc: 0.8873
Epoch 31/45
60000/60000 [==============================] - 3s 48us/step - loss: 
0.2489 - acc: 0.9068 - val_loss: 0.3216 - val_acc: 0.8868
Epoch 32/45
60000/60000 [==============================] - 3s 47us/step - loss: 
0.2454 - acc: 0.9082 - val_loss: 0.3319 - val_acc: 0.8841
Epoch 33/45
60000/60000 [==============================] - 3s 47us/step - loss: 
0.2407 - acc: 0.9096 - val_loss: 0.3286 - val_acc: 0.8890
Epoch 34/45
60000/60000 [==============================] - 3s 48us/step - loss: 
0.2398 - acc: 0.9095 - val_loss: 0.3243 - val_acc: 0.8910
Epoch 35/45
60000/60000 [==============================] - 3s 47us/step - loss: 
0.2377 - acc: 0.9096 - val_loss: 0.3215 - val_acc: 0.8901
Epoch 36/45
60000/60000 [==============================] - 3s 49us/step - loss: 
0.2362 - acc: 0.9108 - val_loss: 0.3248 - val_acc: 0.8901
Epoch 37/45
60000/60000 [==============================] - 3s 47us/step - loss: 
0.2354 - acc: 0.9110 - val_loss: 0.3320 - val_acc: 0.8853
Epoch 38/45
60000/60000 [==============================] - 3s 48us/step - loss: 
0.2339 - acc: 0.9111 - val_loss: 0.3278 - val_acc: 0.8892
Epoch 39/45
60000/60000 [==============================] - 3s 47us/step - loss: 
0.2301 - acc: 0.9134 - val_loss: 0.3261 - val_acc: 0.8923
Epoch 40/45
60000/60000 [==============================] - 3s 47us/step - loss: 
0.2304 - acc: 0.9136 - val_loss: 0.3317 - val_acc: 0.8897
```

```
Epoch 41/45
60000/60000 [==============================] - 3s 48us/step - loss:
0.2268 - acc: 0.9149 - val_loss: 0.3320 - val_acc: 0.8884
Epoch 42/45
60000/60000 [==============================] - 3s 46us/step - loss:
0.2252 - acc: 0.9151 - val_loss: 0.3312 - val_acc: 0.8893
Epoch 43/45
60000/60000 [==============================] - 3s 49us/step - loss:
0.2239 - acc: 0.9154 - val_loss: 0.3401 - val_acc: 0.8861
Epoch 44/45
60000/60000 [==============================] - 3s 48us/step - loss:
0.2223 - acc: 0.9157 - val_loss: 0.3310 - val_acc: 0.8928
Epoch 45/45
60000/60000 [==============================] - 3s 48us/step - loss:
0.2210 - acc: 0.9163 - val_loss: 0.3285 - val_acc: 0.8923
Training Finished!

Train accuracy: 0.9396166666666667
Test accuracy: 0.8923
```

In [5]:

```
%matplotlib inline

# Plot training & validation accuracy values
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()

# Plot training & validation loss values
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```





CNN

Although the MLP worked very good accordingly, the results were not very satisfactory. The accuracy for the MLP as it was showed above was 88% on test which is very low. Thus, we tried to solve the problem with CNN. The network we have used consists of convolutional layers with 32 and 64 units, both have 3x3 kernel size and ReLU activation and both have a 30% dropout rate. Then, these layers are followed by 512 dense hidden units.

In [6]:

```
x_train_ch = np.reshape(x_train, (x_train.shape[0], 28, 28, 1))
x_test_ch = np.reshape(x_test, (x_test.shape[0], 28, 28, 1))

model = Sequential()

model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28,
1)))
model.add(Dropout(0.3))
# model.add(MaxPooling2D(pool_size=(2, 2)))
# model.add(Conv2D(48, (3, 3), activation='relu'))
# model.add(Dropout(0.3))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Dropout(0.3))
# model.add(Conv2D(128, (3, 3), activation='relu'))
# model.add(Dropout(0.3))
model.add(MaxPooling2D(pool_size=(2, 2)))


model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.25))
model.add(Dense(10, activation='softmax'))


print 'Training...'
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=
['accuracy'])
history = model.fit(
    x_train_ch,
    y_train,
    epochs=40,
    batch_size=128,
#     verbose=0,
    validation_data=(x_test_ch, y_test)
)
print 'Training Finished!'

train_loss, train_acc = model.evaluate(x_train_ch,  y_train, verbose=0)
test_loss, test_acc = model.evaluate(x_test_ch,  y_test, verbose=0)

print ''
print 'Train accuracy:', train_acc
print 'Test accuracy:', test_acc
```

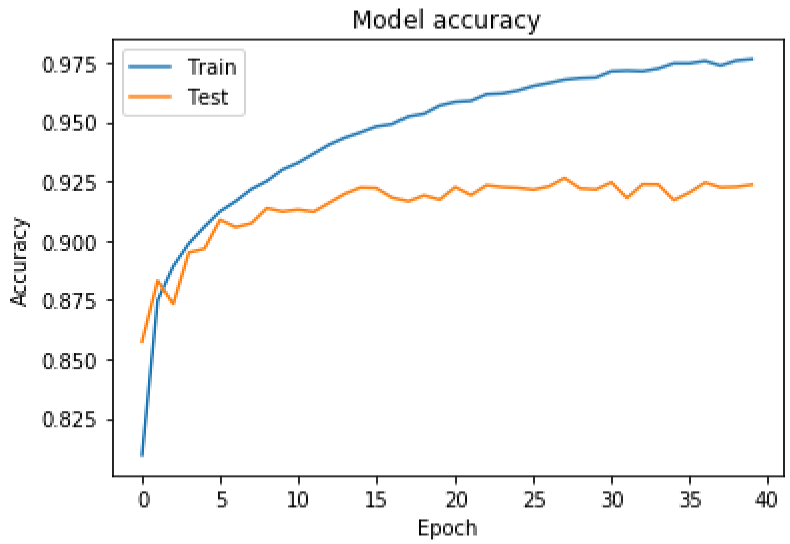
```
Training...
Train on 60000 samples, validate on 10000 samples
Epoch 1/40
60000/60000 [==============================] - 86s 1ms/step - loss:
0.5254 - acc: 0.8095 - val_loss: 0.4292 - val_acc: 0.8574
Epoch 2/40
60000/60000 [==============================] - 84s 1ms/step - loss:
0.3426 - acc: 0.8749 - val_loss: 0.3786 - val_acc: 0.8831
Epoch 3/40
60000/60000 [==============================] - 85s 1ms/step - loss:
0.3015 - acc: 0.8892 - val_loss: 0.3822 - val_acc: 0.8733
Epoch 4/40
60000/60000 [==============================] - 89s 1ms/step - loss:
0.2722 - acc: 0.8988 - val_loss: 0.3084 - val_acc: 0.8950
Epoch 5/40
60000/60000 [==============================] - 74s 1ms/step - loss:
0.2522 - acc: 0.9059 - val_loss: 0.3072 - val_acc: 0.8966
Epoch 6/40
60000/60000 [==============================] - 74s 1ms/step - loss:
0.2350 - acc: 0.9123 - val_loss: 0.2751 - val_acc: 0.9088
Epoch 7/40
60000/60000 [==============================] - 72s 1ms/step - loss:
0.2234 - acc: 0.9166 - val_loss: 0.2716 - val_acc: 0.9057
Epoch 8/40
60000/60000 [==============================] - 74s 1ms/step - loss:
0.2075 - acc: 0.9217 - val_loss: 0.2582 - val_acc: 0.9073
Epoch 9/40
60000/60000 [==============================] - 74s 1ms/step - loss:
0.1983 - acc: 0.9251 - val_loss: 0.2464 - val_acc: 0.9137
Epoch 10/40
60000/60000 [==============================] - 73s 1ms/step - loss:
0.1859 - acc: 0.9300 - val_loss: 0.2414 - val_acc: 0.9123
Epoch 11/40
60000/60000 [==============================] - 74s 1ms/step - loss:
0.1762 - acc: 0.9328 - val_loss: 0.2464 - val_acc: 0.9131
Epoch 12/40
60000/60000 [==============================] - 74s 1ms/step - loss:
0.1664 - acc: 0.9367 - val_loss: 0.2369 - val_acc: 0.9122
Epoch 13/40
60000/60000 [==============================] - 74s 1ms/step - loss:
0.1572 - acc: 0.9405 - val_loss: 0.2342 - val_acc: 0.9160
Epoch 14/40
60000/60000 [==============================] - 72s 1ms/step - loss:
0.1511 - acc: 0.9434 - val_loss: 0.2247 - val_acc: 0.9198
Epoch 15/40
60000/60000 [==============================] - 73s 1ms/step - loss:
0.1428 - acc: 0.9457 - val_loss: 0.2156 - val_acc: 0.9224
Epoch 16/40
60000/60000 [==============================] - 73s 1ms/step - loss:
0.1366 - acc: 0.9481 - val_loss: 0.2161 - val_acc: 0.9222
Epoch 17/40
60000/60000 [==============================] - 73s 1ms/step - loss:
0.1323 - acc: 0.9491 - val_loss: 0.2221 - val_acc: 0.9182
Epoch 18/40
60000/60000 [==============================] - 72s 1ms/step - loss:
0.1254 - acc: 0.9523 - val_loss: 0.2239 - val_acc: 0.9166
Epoch 19/40
60000/60000 [==============================] - 73s 1ms/step - loss:
0.1192 - acc: 0.9535 - val_loss: 0.2237 - val_acc: 0.9191
Epoch 20/40
60000/60000 [==============================] - 73s 1ms/step - loss:
```

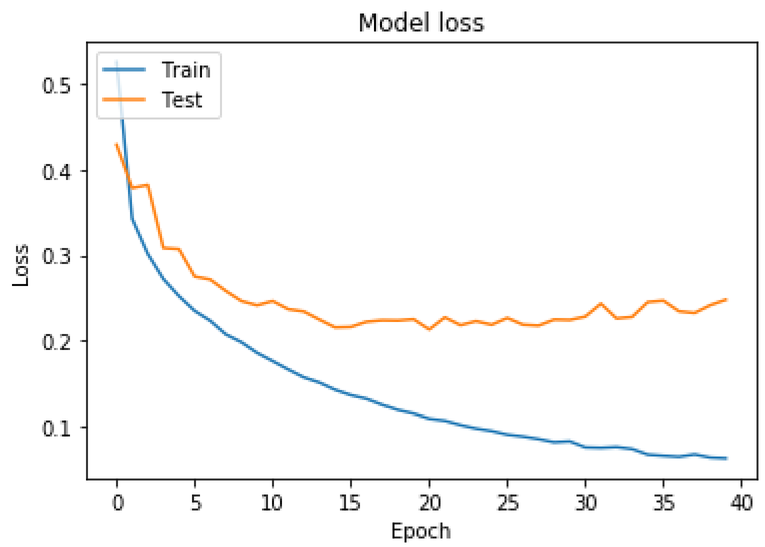
```
0.1151 - acc: 0.9569 - val_loss: 0.2251 - val_acc: 0.9173
Epoch 21/40
60000/60000 [==============================] - 73s 1ms/step - loss:
0.1091 - acc: 0.9585 - val_loss: 0.2132 - val_acc: 0.9226
Epoch 22/40
60000/60000 [==============================] - 72s 1ms/step - loss:
0.1068 - acc: 0.9589 - val_loss: 0.2274 - val_acc: 0.9192
Epoch 23/40
60000/60000 [==============================] - 73s 1ms/step - loss:
0.1018 - acc: 0.9617 - val_loss: 0.2182 - val_acc: 0.9234
Epoch 24/40
60000/60000 [==============================] - 73s 1ms/step - loss:
0.0977 - acc: 0.9621 - val_loss: 0.2229 - val_acc: 0.9226
Epoch 25/40
60000/60000 [==============================] - 73s 1ms/step - loss:
0.0948 - acc: 0.9633 - val_loss: 0.2187 - val_acc: 0.9223
Epoch 26/40
60000/60000 [==============================] - 72s 1ms/step - loss:
0.0906 - acc: 0.9652 - val_loss: 0.2266 - val_acc: 0.9215
Epoch 27/40
60000/60000 [==============================] - 73s 1ms/step - loss:
0.0884 - acc: 0.9664 - val_loss: 0.2188 - val_acc: 0.9229
Epoch 28/40
60000/60000 [==============================] - 73s 1ms/step - loss:
0.0854 - acc: 0.9679 - val_loss: 0.2177 - val_acc: 0.9264
Epoch 29/40
60000/60000 [==============================] - 73s 1ms/step - loss:
0.0818 - acc: 0.9685 - val_loss: 0.2247 - val_acc: 0.9220
Epoch 30/40
60000/60000 [==============================] - 71s 1ms/step - loss:
0.0828 - acc: 0.9688 - val_loss: 0.2243 - val_acc: 0.9216
Epoch 31/40
60000/60000 [==============================] - 74s 1ms/step - loss:
0.0758 - acc: 0.9714 - val_loss: 0.2283 - val_acc: 0.9246
Epoch 32/40
60000/60000 [==============================] - 73s 1ms/step - loss:
0.0753 - acc: 0.9716 - val_loss: 0.2433 - val_acc: 0.9180
Epoch 33/40
60000/60000 [==============================] - 73s 1ms/step - loss:
0.0762 - acc: 0.9714 - val_loss: 0.2261 - val_acc: 0.9238
Epoch 34/40
60000/60000 [==============================] - 72s 1ms/step - loss:
0.0739 - acc: 0.9725 - val_loss: 0.2278 - val_acc: 0.9237
Epoch 35/40
60000/60000 [==============================] - 74s 1ms/step - loss:
0.0673 - acc: 0.9747 - val_loss: 0.2452 - val_acc: 0.9171
Epoch 36/40
60000/60000 [==============================] - 75s 1ms/step - loss:
0.0660 - acc: 0.9748 - val_loss: 0.2469 - val_acc: 0.9203
Epoch 37/40
60000/60000 [==============================] - 74s 1ms/step - loss:
0.0648 - acc: 0.9757 - val_loss: 0.2344 - val_acc: 0.9245
Epoch 38/40
60000/60000 [==============================] - 73s 1ms/step - loss:
0.0676 - acc: 0.9738 - val_loss: 0.2324 - val_acc: 0.9225
Epoch 39/40
60000/60000 [==============================] - 74s 1ms/step - loss:
0.0639 - acc: 0.9759 - val_loss: 0.2416 - val_acc: 0.9227
Epoch 40/40
60000/60000 [==============================] - 74s 1ms/step - loss:
0.0630 - acc: 0.9765 - val_loss: 0.2480 - val_acc: 0.9236
```

```
Training Finished!

Train accuracy: 0.9915166666666667
Test accuracy: 0.9236
```

In [7]:

```
%matplotlib inline

# Plot training & validation accuracy values
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()

# Plot training & validation loss values
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```





In the next section we solve the CIFAR-10 using CNNs. Please follow up with this link (https://github.com/ArefMq/simple-nn/blob/master/Day-6.ipynb).

Day 6

CNN for solving CIFAR-10

In this section, we try to solve the CIFAR-10 via CNN. This dataset consists of 10 classes as below:

- airplane
- automobile
- bird
- cat
- deer
- dog
- frog
- horse
- ship
- truck

In [1]:

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D

from random import randint
```

```
Using TensorFlow backend.
```

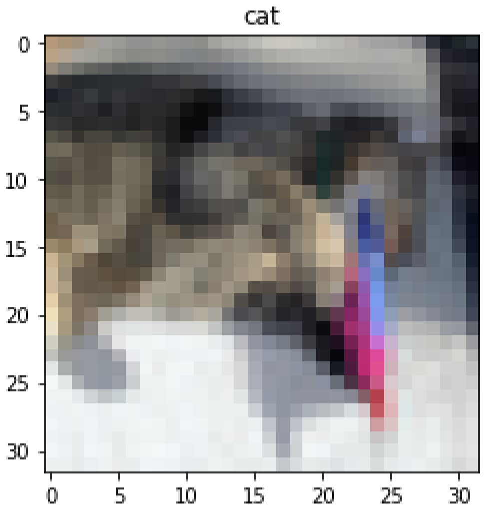
In [2]:

```
dataset = keras.datasets.cifar10
(x_train, y_train), (x_test, y_test) = dataset.load_data()

x_train = x_train / 255.0
x_test = x_test / 255.0


print "x_train.shape =", x_train.shape
print "x_test.shape  =", x_test.shape
```

```
x_train.shape = (50000, 32, 32, 3)
x_test.shape  = (10000, 32, 32, 3)
```

In [3]:

```
def get_label_name(y):
    class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
                   'dog', 'frog', 'horse', 'ship', 'truck']
    return class_names[int(y)]

rnd_index = randint(0, x_train.shape[0]-1)
plt.imshow(x_train[rnd_index])
plt.title(get_label_name(y_train[rnd_index]))
plt.show()
```



We have used a convolutional structure with two sets of convolutional layers each containing two Conv2D followed by a Maxx-Pooling layer. The kernel size of Conv layers are all 3x3 and the pooling layers have 2x2 kernel size. The activation function for all of the Conv layers is set to ReLU.

In [4]:

```
model = Sequential()

model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(32, 32,
3)))
model.add(Dropout(0.4))
model.add(Conv2D(48, (3, 3), activation='relu'))
model.add(Dropout(0.4))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Dropout(0.3))
model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(Dropout(0.3))
model.add(MaxPooling2D(pool_size=(2, 2)))


model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.25))
model.add(Dense(10, activation='softmax'))


print 'Training...'
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=
['accuracy'])
history = model.fit(
    x_train,
    y_train,
    epochs=40,
    batch_size=128,
#     verbose=0,
    validation_data=(x_test, y_test)
)
print 'Training Finished!'

train_loss, train_acc = model.evaluate(x_train,  y_train, verbose=0)
test_loss, test_acc = model.evaluate(x_test,  y_test, verbose=0)

print ''
print 'Train accuracy:', train_acc
print 'Test accuracy:', test_acc
```

```
Training...
Train on 50000 samples, validate on 10000 samples
Epoch 1/40
50000/50000 [==============================] - 304s 6ms/step - loss:
1.6938 - acc: 0.3772 - val_loss: 1.5675 - val_acc: 0.4980
Epoch 2/40
50000/50000 [==============================] - 319s 6ms/step - loss:
1.2629 - acc: 0.5489 - val_loss: 1.2831 - val_acc: 0.5810
Epoch 3/40
50000/50000 [==============================] - 289s 6ms/step - loss:
1.0724 - acc: 0.6206 - val_loss: 1.1331 - val_acc: 0.6394
Epoch 4/40
50000/50000 [==============================] - 291s 6ms/step - loss:
0.9486 - acc: 0.6666 - val_loss: 0.9984 - val_acc: 0.6769
Epoch 5/40
50000/50000 [==============================] - 288s 6ms/step - loss:
0.8425 - acc: 0.7025 - val_loss: 0.9394 - val_acc: 0.6973
Epoch 6/40
50000/50000 [==============================] - 287s 6ms/step - loss:
0.7549 - acc: 0.7353 - val_loss: 0.8302 - val_acc: 0.7361
Epoch 7/40
50000/50000 [==============================] - 288s 6ms/step - loss:
0.7023 - acc: 0.7528 - val_loss: 0.8379 - val_acc: 0.7292
Epoch 8/40
50000/50000 [==============================] - 288s 6ms/step - loss:
0.6454 - acc: 0.7724 - val_loss: 0.7660 - val_acc: 0.7485
Epoch 9/40
50000/50000 [==============================] - 289s 6ms/step - loss:
0.5974 - acc: 0.7887 - val_loss: 0.7639 - val_acc: 0.7466
Epoch 10/40
50000/50000 [==============================] - 292s 6ms/step - loss:
0.5461 - acc: 0.8091 - val_loss: 0.7287 - val_acc: 0.7547
Epoch 11/40
50000/50000 [==============================] - 246s 5ms/step - loss:
0.5055 - acc: 0.8207 - val_loss: 0.7065 - val_acc: 0.7677
Epoch 12/40
50000/50000 [==============================] - 179s 4ms/step - loss:
0.4747 - acc: 0.8293 - val_loss: 0.7123 - val_acc: 0.7639
Epoch 13/40
50000/50000 [==============================] - 181s 4ms/step - loss:
0.4480 - acc: 0.8386 - val_loss: 0.6943 - val_acc: 0.7684
Epoch 14/40
50000/50000 [==============================] - 178s 4ms/step - loss:
0.4171 - acc: 0.8501 - val_loss: 0.7500 - val_acc: 0.7483
Epoch 15/40
50000/50000 [==============================] - 178s 4ms/step - loss:
0.3897 - acc: 0.8608 - val_loss: 0.7098 - val_acc: 0.7595
Epoch 16/40
50000/50000 [==============================] - 179s 4ms/step - loss:
0.3748 - acc: 0.8649 - val_loss: 0.6855 - val_acc: 0.7700
Epoch 17/40
50000/50000 [==============================] - 178s 4ms/step - loss:
0.3532 - acc: 0.8740 - val_loss: 0.6890 - val_acc: 0.7679
Epoch 18/40
50000/50000 [==============================] - 179s 4ms/step - loss:
0.3387 - acc: 0.8798 - val_loss: 0.6955 - val_acc: 0.7678
Epoch 19/40
50000/50000 [==============================] - 179s 4ms/step - loss:
0.3263 - acc: 0.8844 - val_loss: 0.6799 - val_acc: 0.7751
Epoch 20/40
50000/50000 [==============================] - 178s 4ms/step - loss:
```

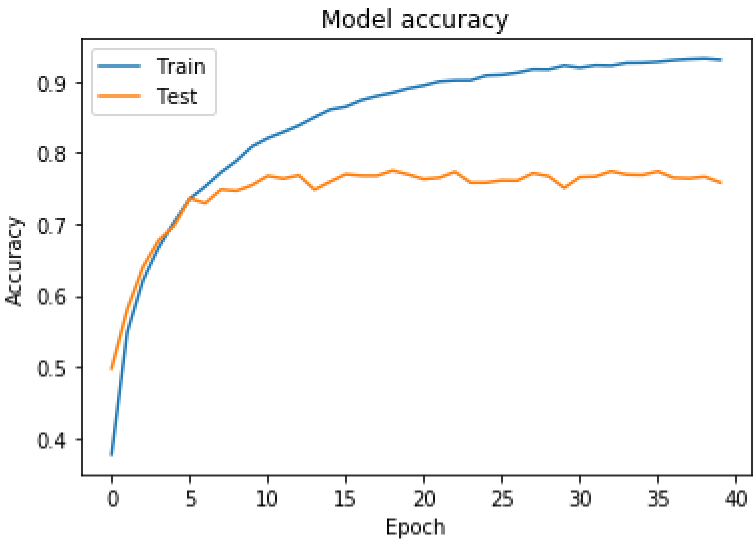
```
0.3098 - acc: 0.8903 - val_loss: 0.6872 - val_acc: 0.7696
Epoch 21/40
50000/50000 [==============================] - 178s 4ms/step - loss:
0.2991 - acc: 0.8946 - val_loss: 0.7172 - val_acc: 0.7631
Epoch 22/40
50000/50000 [==============================] - 178s 4ms/step - loss:
0.2817 - acc: 0.9002 - val_loss: 0.7012 - val_acc: 0.7651
Epoch 23/40
50000/50000 [==============================] - 178s 4ms/step - loss:
0.2752 - acc: 0.9018 - val_loss: 0.6814 - val_acc: 0.7733
Epoch 24/40
50000/50000 [==============================] - 178s 4ms/step - loss:
0.2739 - acc: 0.9019 - val_loss: 0.7457 - val_acc: 0.7581
Epoch 25/40
50000/50000 [==============================] - 178s 4ms/step - loss:
0.2571 - acc: 0.9087 - val_loss: 0.7397 - val_acc: 0.7582
Epoch 26/40
50000/50000 [==============================] - 178s 4ms/step - loss:
0.2570 - acc: 0.9097 - val_loss: 0.7152 - val_acc: 0.7613
Epoch 27/40
50000/50000 [==============================] - 178s 4ms/step - loss:
0.2497 - acc: 0.9126 - val_loss: 0.7255 - val_acc: 0.7610
Epoch 28/40
50000/50000 [==============================] - 178s 4ms/step - loss:
0.2397 - acc: 0.9174 - val_loss: 0.7013 - val_acc: 0.7714
Epoch 29/40
50000/50000 [==============================] - 178s 4ms/step - loss:
0.2356 - acc: 0.9171 - val_loss: 0.7246 - val_acc: 0.7672
Epoch 30/40
50000/50000 [==============================] - 178s 4ms/step - loss:
0.2234 - acc: 0.9227 - val_loss: 0.7740 - val_acc: 0.7508
Epoch 31/40
50000/50000 [==============================] - 178s 4ms/step - loss:
0.2292 - acc: 0.9196 - val_loss: 0.7363 - val_acc: 0.7660
Epoch 32/40
50000/50000 [==============================] - 175s 4ms/step - loss:
0.2188 - acc: 0.9230 - val_loss: 0.7249 - val_acc: 0.7668
Epoch 33/40
50000/50000 [==============================] - 179s 4ms/step - loss:
0.2202 - acc: 0.9224 - val_loss: 0.7214 - val_acc: 0.7741
Epoch 34/40
50000/50000 [==============================] - 179s 4ms/step - loss:
0.2126 - acc: 0.9264 - val_loss: 0.7598 - val_acc: 0.7695
Epoch 35/40
50000/50000 [==============================] - 178s 4ms/step - loss:
0.2102 - acc: 0.9266 - val_loss: 0.7318 - val_acc: 0.7689
Epoch 36/40
50000/50000 [==============================] - 180s 4ms/step - loss:
0.2087 - acc: 0.9279 - val_loss: 0.7244 - val_acc: 0.7739
Epoch 37/40
50000/50000 [==============================] - 179s 4ms/step - loss:
0.2014 - acc: 0.9302 - val_loss: 0.7762 - val_acc: 0.7650
Epoch 38/40
50000/50000 [==============================] - 179s 4ms/step - loss:
0.1952 - acc: 0.9317 - val_loss: 0.7654 - val_acc: 0.7642
Epoch 39/40
50000/50000 [==============================] - 179s 4ms/step - loss:
0.1928 - acc: 0.9326 - val_loss: 0.7557 - val_acc: 0.7665
Epoch 40/40
50000/50000 [==============================] - 180s 4ms/step - loss:
0.2017 - acc: 0.9307 - val_loss: 0.7908 - val_acc: 0.7583
```

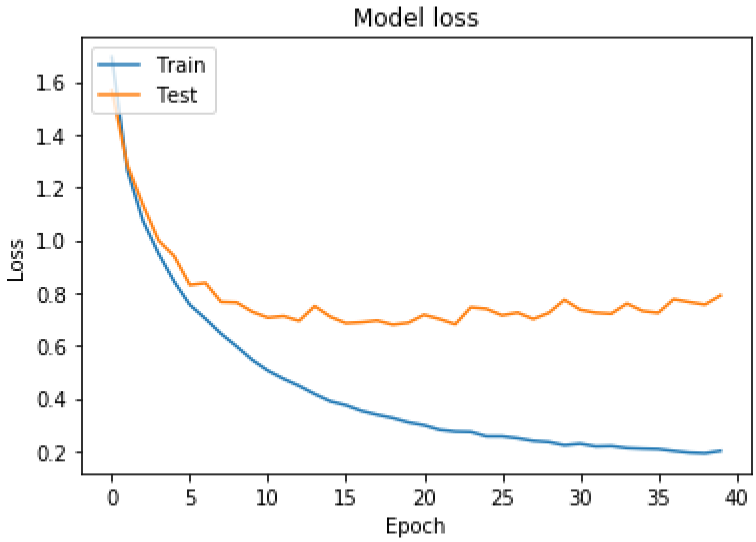
```
Training Finished!

Train accuracy: 0.9831
Test accuracy: 0.7583
```

As it is showed above the Train accuracy of the network is 98% and the test accuracy of it is 75%. Below, the accuracy values and loss values of the network are plotted throughout the time. We have to note that, the reason to run the network only for 40 epochs is that after this number the network tends to start overfitting the loss value of the train set seemed to stop decreasing.

In [5]:

```
%matplotlib inline

# Plot training & validation accuracy values
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()

# Plot training & validation loss values
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```


Model accuracy
Train
Test
Accuracy
Epoch


Model loss
Train
Test
Loss
Epoch