In [1]:
```
%pylab inline
import numpy as np
import matplotlib.pyplot as mp
import time
```

```
Populating the interactive namespace from numpy and matplotlib
```

Dataset

The dataset mnist.pkl.npy was downloaded using download.py.

In [2]:
```
train_data, valid_data, test_data = np.load('mnist.pkl.npy')
print("Datasets (train, valid, test) -",np.shape(train_data[0]), np.shape(valid_data[0]), np.shape(te
st_data[0]))
```

```
Datasets (train, valid, test) - (50000, 784) (10000, 784) (10000, 784)
```

Building the Model[35] Consider an MLP with two hidden layers withh1andh2hidden units.For the MNIST dataset, the number of features of the input datah0is 784. The output of theneural network is parameterized by a softmax of h3= 10 classes.

1. Build an MLP and choose the values ofh1andh2such that the total number of parameters(including biases) falls within the range of [0.5M, 1.0M].
2. Implement the forward and backward propagation of the MLP in numpy without using any ofthe deep learning frameworks that provides automatic differentiation. Use the class structureprovided here.
3. Train the MLP using the probability loss (cross entropy) as training criterion. We minimizethis criterion to optimize the model parameters usingstochastic gradient descent.

In [3]:
```
### class for MLP
class NN(object):
    def __init__(self, hidden_dims=(512,768), weight_type = "Glorot", learning_rate = 0.1, silent_sea
rch = 0):
        ## Initialising the hidden layer sizes
        self.h0 = 784
        self.h1 = int(hidden_dims[0])
        self.h2 = int(hidden_dims[1])
        self.h3 = 10

        ## learning rate
        self.eta = learning_rate

        ## initializing the bias
        self.b0 = np.zeros(self.h1).reshape(-1,1)
        self.b1 = np.zeros(self.h2).reshape(-1,1)
        self.b2 = np.zeros(self.h3).reshape(-1,1)

        ## Weight variables
        self.W0 = 0
        self.W1 = 0
        self.W2 = 0

        ## initializing the weights depending on the weight_type
        self.initialize_weights(weight_type = weight_type)

        ## To obtain the total count of parameters int he model and restric it to between 0.5-1 Milli
on.
        print("parameter count =", sum([len(self.b0),len(self.b1),len(self.b2),len(self.W0.reshape(-1
,1)),len(self.W1.reshape(-1,1)),len(self.W2.reshape(-1,1))]))

        self.silent = silent_search

    def initialize_weights(self, weight_type = "Glorot"):
        if weight_type == "Zeros":
            self.W0 = np.zeros(self.h1*self.h0).reshape(self.h1,self.h0)
            self.W1 = np.zeros(self.h2*self.h1).reshape(self.h2,self.h1)
            self.W2 = np.zeros(self.h3*self.h2).reshape(self.h3,self.h2)
        elif weight_type == "Normal":
            self.W0 = np.random.normal(0.0,1.0e2,size = self.h1*self.h0).reshape(self.h1,self.h0)
            self.W1 = np.random.normal(0.0,1.0e2,size = self.h2*self.h1).reshape(self.h2,self.h1)
            self.W2 = np.random.normal(0.0,1.0e2,size = self.h3*self.h2).reshape(self.h3,self.h2)
```

```
        elif weight_type == "Glorot":
            dl0 = np.sqrt(6.0/(self.h1+self.h0))
            self.W0 = np.random.uniform(-dl0,dl0,size= self.h1*self.h0).reshape(self.h1,self.h0)
            dl1 = np.sqrt(6.0/(self.h2+self.h1))
            self.W1 = np.random.uniform(-dl1,dl1,size= self.h2*self.h1).reshape(self.h2,self.h1)
            dl2 = np.sqrt(6.0/(self.h3+self.h2))
            self.W2 = np.random.uniform(-dl2,dl2,size= self.h3*self.h2).reshape(self.h3,self.h2)

    def forward(self, input_data):
        input_data = np.array(input_data)                                  # [d,n]
        cache = {}
        cache['input'] = input_data

        cache['ha_1'] = np.matmul(self.W0,cache['input']) + self.b0
        cache['hs_1'] = self.activation(cache['ha_1'])

        cache['ha_2'] = np.matmul(self.W1,cache['hs_1']) + self.b1
        cache['hs_2'] = self.activation(cache['ha_2'])

        cache['ha_3'] = np.matmul(self.W2,cache['hs_2']) + self.b2
        cache['output'] = self.softmax(cache['ha_3'])

        return cache


    def activation(self, input_data):
        ## relu activation
        x = np.array(input_data)
        x[x<0] = 0
        return x

    def softmax(self, input_data):
        ## thanks to https://deepnotes.io/softmax-crossentropy
        exps = np.exp(input_data - np.max(input_data,axis=0))
        return np.true_divide(exps, np.sum(exps,axis = 0))


    def loss(self, prediction_list, labels):
        ## probability loss (cross entropy)
        loss_values = [-np.log(prediction[label]) for label,prediction in zip(labels,prediction_list.
T)]
        return np.array(loss_values)
```

```
    def backward(self, cache, labels):
        n = float(len(labels))
        grads = {}
        grads['ha_3'] = cache['output'] - np.transpose(np.eye(self.h3)[labels])

        grads['W2'] = np.matmul(grads['ha_3'], cache['hs_2'].T)/n
        grads['b2'] = np.array(np.sum(grads['ha_3'],axis = 1)/n).reshape(-1,1)
        grads['hs_2'] = np.matmul(self.W2.T, grads['ha_3'])
        grads['ha_2'] = np.multiply(grads['hs_2'], np.where(cache['ha_2'] > 0,1.0,0.0))

        grads['W1'] = np.matmul(grads['ha_2'], cache['hs_1'].T)/n
        grads['b1'] = np.array(np.sum(grads['ha_2'],axis = 1)/n).reshape(-1,1)
        grads['hs_1'] = np.matmul(self.W1.T, grads['ha_2'])
        grads['ha_1'] = np.multiply(grads['hs_1'], np.where(cache['ha_1'] > 0,1.0,0.0))

        grads['W0'] = np.matmul(grads['ha_1'], cache['input'].T)/n
        grads['b0'] = np.array(np.sum(grads['ha_1'],axis = 1)/n).reshape(-1,1)

        return grads

    def update(self, grads):
        # Stocastic gradient descent
        self.W2 -= self.eta*grads['W2']
        self.W1 -= self.eta*grads['W1']
        self.W0 -= self.eta*grads['W0']
        self.b2 -= self.eta*grads['b2']
        self.b1 -= self.eta*grads['b1']
        self.b0 -= self.eta*grads['b0']

    def classification_accuracy(self, predicted_labels, true_labels):
        # Classification accuracy in percent
        return np.sum(np.array(predicted_labels)==np.array(true_labels))*100.0/np.array(predicted_lab
els).shape[0]

    def train(self, input_data, labels, epochs = 10, mini_batch = 10):
        n = len(labels)
        input_data = input_data.T
        loss = []

        for e in range(epochs):
            i = 0
            while(i < n):
                if i + mini_batch <= n:
```

```
                    minibatch_data = input_data[:,i:i + mini_batch]
                    minibatch_labels = labels[i:i + mini_batch]
                else:
                    minibatch_data = input_data[:,i:]
                    minibatch_labels = labels[i:]

                cache = self.forward(minibatch_data)
                grads = self.backward(cache,minibatch_labels)

                self.update(grads)
                i = i + mini_batch

            predictions, meanloss = self.test(input_data,labels, run_for = "training")
            loss.append(meanloss)
            if (self.silent == 0):
                print("Classification accuracy [%] on training = ",self.classification_accuracy(predi
ctions,labels))
        return loss

    def test(self, input_data, labels, run_for = "testing"):
        if (self.silent == 0):
            print("\n\n"+run_for+"..")

        cache = self.forward(input_data)

        mean_loss = np.mean(self.loss(cache['output'],labels))
        if (self.silent == 0):
            print("Mean Loss for "+run_for+" = ",mean_loss)

        return np.argmax(cache['output'],axis = 0), mean_loss
```

In the following sub-questions, please specify the model architecture (number of hidden units per layer, and the total number of parameters), the nonlinearity chosen as neuron activation, learning rate, mini-batch size.

Model Architecture - 2 layer MLP with input size 784, first hiddent layer of size 512, second hidden layer of size 768 and output layer of size 10.

Total number of parameters - 804352.

Neuron activation non-linearity - Rectified linear Unit (ReLu).

Output non-linearity - Softmax.

Learning rate (eta) - 0.1.

Mini-Batch size - 128.

Splitting features and labels

In [4]:
```
n = len(train_data[0])
print(n)
train_data_features = train_data[0][:n]
train_data_labels = train_data[1][:n]

valid_data_features = valid_data[0]
valid_data_labels = valid_data[1]
```

```
50000
```

Function to run a model

In [5]:
```
def run(epoch = 10, mini_batch = 128, weight_type = "Glorot"):
    since = time.time()

    NN_0 = NN(weight_type = weight_type)
    print("Loading model : ",NN_0.h1,"x",NN_0.h2,"lr : ", NN_0.eta)

    loss_val = NN_0.train(train_data_features,train_data_labels,epochs = epoch, mini_batch = mini_bat
ch)

    valid_predictions, _ = NN_0.test(valid_data_features.T, valid_data_labels)
    acc = NN_0.classification_accuracy(valid_predictions, valid_data_labels)
    print("Classification accuracy [%] on validation = ",acc)

    time_elapsed = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.format(
          time_elapsed // 60, time_elapsed % 60))

    return loss_val
```

Initialization: In this sub-question, we consider different initial values for the weight param-eters. Set the biases to be zeros, and consider the following settings for the weight parameters:

1. Train the model for 10 epochs3using the initialization methods above and record the average loss measured on the training data at the end of each epoch (10 values for each setup).

In [6]:
```
zeros_losses = run(weight_type = "Zeros")
```

```
parameter count = 803594
Loading model :  512 x 768 lr :  0.1


training..
Mean Loss for training =  2.30106604987257
Classification accuracy [%] on training =  11.356


training..
Mean Loss for training =  2.3010626883570473
Classification accuracy [%] on training =  11.356


training..
Mean Loss for training =  2.301062639269737
Classification accuracy [%] on training =  11.356


training..
Mean Loss for training =  2.30106263826223
Classification accuracy [%] on training =  11.356


training..
Mean Loss for training =  2.301062638237786
Classification accuracy [%] on training =  11.356


training..
Mean Loss for training =  2.3010626382371333
Classification accuracy [%] on training =  11.356


training..
Mean Loss for training =  2.301062638237115
Classification accuracy [%] on training =  11.356


training..
Mean Loss for training =  2.301062638237115
Classification accuracy [%] on training =  11.356
```

```
training..
Mean Loss for training =  2.301062638237115
Classification accuracy [%] on training =  11.356


training..
Mean Loss for training =  2.301062638237115
Classification accuracy [%] on training =  11.356


testing..
Mean Loss for testing =  2.301854740465508
Classification accuracy [%] on validation =  10.64
Training complete in 1m 13s
```

In [7]:
```
normal_losses = run(weight_type = "Normal")
```

```
parameter count = 803594
Loading model :  512 x 768 lr :  0.1


training..
Mean Loss for training =  2.301148133809196
Classification accuracy [%] on training =  11.356


training..
Mean Loss for training =  2.3010627737389466
Classification accuracy [%] on training =  11.356


training..
Mean Loss for training =  2.3010626412812285
Classification accuracy [%] on training =  11.356


training..
Mean Loss for training =  2.3010626383428696
Classification accuracy [%] on training =  11.356


training..
Mean Loss for training =  2.3010626382406802
Classification accuracy [%] on training =  11.356


training..
Mean Loss for training =  2.301062638237229
Classification accuracy [%] on training =  11.356


training..
Mean Loss for training =  2.3010626382371187
Classification accuracy [%] on training =  11.356


training..
Mean Loss for training =  2.301062638237115
Classification accuracy [%] on training =  11.356
```

```
training..
Mean Loss for training =  2.301062638237115
Classification accuracy [%] on training =  11.356


training..
Mean Loss for training =  2.301062638237115
Classification accuracy [%] on training =  11.356


testing..
Mean Loss for testing =  inf
Classification accuracy [%] on validation =  10.64
Training complete in 1m 1s

//anaconda/envs/helios/lib/python3.7/site-packages/ipykernel_launcher.py:79: RuntimeWarning: divide
by zero encountered in log
```

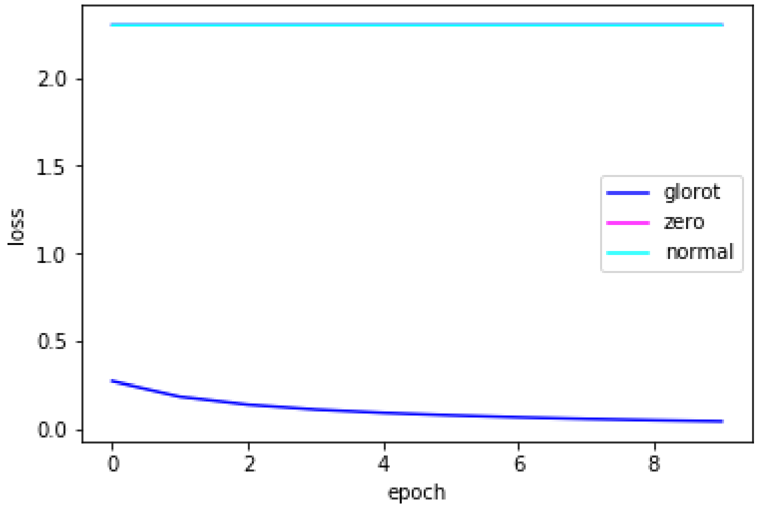
In [8]:
```
glorot_losses = run(weight_type = "Glorot")
```

```
parameter count = 803594
Loading model :  512 x 768 lr :  0.1


training..
Mean Loss for training =  0.2724472669813704
Classification accuracy [%] on training =  91.848


training..
Mean Loss for training =  0.18523987693175825
Classification accuracy [%] on training =  94.536


training..
Mean Loss for training =  0.1400291295956501
Classification accuracy [%] on training =  95.918


training..
Mean Loss for training =  0.11225144840363874
Classification accuracy [%] on training =  96.736


training..
Mean Loss for training =  0.09331997012000898
Classification accuracy [%] on training =  97.302


training..
Mean Loss for training =  0.07926168895761627
Classification accuracy [%] on training =  97.692


training..
Mean Loss for training =  0.06788345141223112
Classification accuracy [%] on training =  98.048


training..
Mean Loss for training =  0.05868652831809437
Classification accuracy [%] on training =  98.34
```

```
training..
Mean Loss for training =  0.0512073964103037
Classification accuracy [%] on training =  98.576


training..
Mean Loss for training =  0.04491940048679391
Classification accuracy [%] on training =  98.756


testing..
Mean Loss for testing =  0.08393884677883673
Classification accuracy [%] on validation =  97.53
Training complete in 1m 4s
```

2. Compare the three setups by plotting the losses against the training time (epoch) and commenton the result

In [9]:
```
plt.plot(glorot_losses, label = "glorot", color = 'blue')
plt.plot(zeros_losses, label = "zero", color = 'magenta')
plt.plot(normal_losses, label = "normal", color = 'cyan')

plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend()
plt.show()
```



We find that for both Zeros and Normal initializations, the loss stays high and the accuracy is 10 as the model is training at a very slow pace. This is because, in the case of Zero initialization, the gradients and the weights are zero. The changes in loss are due to training of bias to the majority class.

Same phenomenon happens for normal initialization. Due to large values of weights (due to the non zero probability of normal distribution for large values), the gradients of the weights explode (move towards infinity and negative infinity) due to the unbounded nature of ReLu activation function. This can be solved with a bounded activation function like sigmoid. This trend will eventually leads to zero weight gradients and the training slows to similar to the zero initialization case.

With glorot initialization, the network trains easily reaching high accuracy and low mean loss.

Hyperparameter Search - From now on, use the Glorot initialization method.

1. Find out a combination of hyper-parameters (model architecture, learning rate, nonlinearity,etc.) such that the average accuracy rate on the validation set (r_valid)) is at least 97%.

2. Report the hyper-parameters you tried and the corresponding r_(valid).

In [10]:
```
# Grid of parameters - grid should be an array of integer, as the numpy will return only a list of th
e same type
parameter_space = {
    # grid of {392,512,768,1024}x{256,512,768,1024}
    'hidden_1': [392, 512, 768, 1024],
    'hidden_2': [256, 512, 768, 1024],
    'inv_learning_rate' : [1,10,100]
}

def invRate(dot):
    ## function to convert from inv_learning_rate to learning_rate
    if (dot[2] == 0):
        raise ValueError('Inverse learning should not be 0')
    return 1/dot[2]


def parameter_count(n0 = 784, n1 = 0, n2 = 0, n3 = 10):
    return ((n0+1)*n1)+((n1+1)*n2)+(n2+1)*n3

def validParam(dot):
    ## invalid values (number of paramters > 1M or < 500K) are eliminated
    n1 = dot[0]
    n2 = dot[1]
    size = parameter_count(n0 = 784, n1 = n1, n2 = n2, n3 = 10)
    return (size > 500000 and size < 1000000)

# Creating the grid of model parameters
## Cartesian product : for each grid point, (possible combination in the parameter space) run n epoch
s and store the validation score
grid = np.array(np.meshgrid(parameter_space['hidden_1'],parameter_space['hidden_2'], parameter_space[
'inv_learning_rate'])).T.reshape(-1,len(parameter_space))
grid = [x for x in grid if validParam(x)]


print("Grid of model parameters - [hidden_layer_1,hidden_layer_2,inv_learning_rate]")
grid
```

```
Grid of model parameters - [hidden_layer_1,hidden_layer_2,inv_learning_rate]
```

Out[10]:
```
[array([392, 512,   1]),
 array([392, 768,   1]),
 array([ 392, 1024,    1]),
 array([512, 256,   1]),
 array([512, 512,   1]),
 array([512, 768,   1]),
 array([ 512, 1024,    1]),
 array([768, 256,   1]),
 array([392, 512,  10]),
 array([392, 768,  10]),
 array([ 392, 1024,   10]),
 array([512, 256,  10]),
 array([512, 512,  10]),
 array([512, 768,  10]),
 array([ 512, 1024,   10]),
 array([768, 256,  10]),
 array([392, 512, 100]),
 array([392, 768, 100]),
 array([ 392, 1024,  100]),
 array([512, 256, 100]),
 array([512, 512, 100]),
 array([512, 768, 100]),
 array([ 512, 1024,  100]),
 array([768, 256, 100])]
```

In [11]:
```
len(grid)
```

Out[11]:
```
24
```

In [12]:
```
## for every dot on the grid, run 10 epochs and save validation accuracy

dot_predictions = {}

#this grid search takes about an hour for 24 search points
since = time.time()

for index, dot in enumerate(grid):
    print("")
    NN_0 = NN(hidden_dims=(dot[0],dot[1]), weight_type = "Glorot", learning_rate = invRate(dot), sile
nt_search = 1)
    print("Loading model : ",NN_0.h1,"x",NN_0.h2,"lr : ", NN_0.eta)

    NN_0.train(train_data_features, train_data_labels, epochs = 10, mini_batch = 128)

    valid_predictions, _ = NN_0.test(valid_data_features.T,valid_data_labels)
    acc = NN_0.classification_accuracy(valid_predictions,valid_data_labels)
    print("Classification Accuracy on validation = ",acc,"%")

    dot_predictions["model_"+str(index)] = {}
    dot_predictions["model_"+str(index)]["hidden_1"] = dot[0]
    dot_predictions["model_"+str(index)]["hidden_2"] = dot[1]
    dot_predictions["model_"+str(index)]["parameters"] = parameter_count(n0 = 784, n1 = dot[0], n2 =
dot[1], n3 = 10)
    dot_predictions["model_"+str(index)]["lr"] = invRate(dot)
    dot_predictions["model_"+str(index)]["r_valid [%]"] = acc

    time_elapsed = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.format(
      time_elapsed // 60, time_elapsed % 60))
```

```
parameter count = 514066
Loading model :  392 x 512 lr :  1.0
Classification Accuracy on validation =  96.56 %
Training complete in 0m 39s

parameter count = 617234
Loading model :  392 x 768 lr :  1.0
Classification Accuracy on validation =  97.58 %
Training complete in 1m 31s

parameter count = 720402
Loading model :  392 x 1024 lr :  1.0
Classification Accuracy on validation =  97.39 %
Training complete in 2m 34s

parameter count = 535818
Loading model :  512 x 256 lr :  1.0
Classification Accuracy on validation =  97.57 %
Training complete in 3m 11s

parameter count = 669706
Loading model :  512 x 512 lr :  1.0
Classification Accuracy on validation =  97.74 %
Training complete in 4m 1s

parameter count = 803594
Loading model :  512 x 768 lr :  1.0
Classification Accuracy on validation =  97.65 %
Training complete in 5m 6s

parameter count = 937482
Loading model :  512 x 1024 lr :  1.0
Classification Accuracy on validation =  97.5 %
Training complete in 6m 21s

parameter count = 802314
Loading model :  768 x 256 lr :  1.0
Classification Accuracy on validation =  97.74 %
Training complete in 7m 12s

parameter count = 514066
Loading model :  392 x 512 lr :  0.1
Classification Accuracy on validation =  97.5 %
```

```
Training complete in 7m 55s

parameter count = 617234
Loading model :  392 x 768 lr :  0.1
Classification Accuracy on validation =  97.61 %
Training complete in 8m 48s

parameter count = 720402
Loading model :  392 x 1024 lr :  0.1
Classification Accuracy on validation =  97.54 %
Training complete in 9m 51s

parameter count = 535818
Loading model :  512 x 256 lr :  0.1
Classification Accuracy on validation =  97.67 %
Training complete in 10m 31s

parameter count = 669706
Loading model :  512 x 512 lr :  0.1
Classification Accuracy on validation =  97.69 %
Training complete in 11m 24s

parameter count = 803594
Loading model :  512 x 768 lr :  0.1
Classification Accuracy on validation =  97.74 %
Training complete in 12m 31s

parameter count = 937482
Loading model :  512 x 1024 lr :  0.1
Classification Accuracy on validation =  97.67 %
Training complete in 13m 47s

parameter count = 802314
Loading model :  768 x 256 lr :  0.1
Classification Accuracy on validation =  97.54 %
Training complete in 14m 42s

parameter count = 514066
Loading model :  392 x 512 lr :  0.01
Classification Accuracy on validation =  93.54 %
Training complete in 15m 23s

parameter count = 617234
```

```
Loading model :  392 x 768 lr :  0.01
Classification Accuracy on validation =  93.67 %
Training complete in 16m 14s

parameter count = 720402
Loading model :  392 x 1024 lr :  0.01
Classification Accuracy on validation =  93.28 %
Training complete in 17m 15s

parameter count = 535818
Loading model :  512 x 256 lr :  0.01
Classification Accuracy on validation =  93.87 %
Training complete in 17m 53s

parameter count = 669706
Loading model :  512 x 512 lr :  0.01
Classification Accuracy on validation =  93.79 %
Training complete in 18m 43s

parameter count = 803594
Loading model :  512 x 768 lr :  0.01
Classification Accuracy on validation =  93.46 %
Training complete in 19m 45s

parameter count = 937482
Loading model :  512 x 1024 lr :  0.01
Classification Accuracy on validation =  93.77 %
Training complete in 20m 56s

parameter count = 802314
Loading model :  768 x 256 lr :  0.01
Classification Accuracy on validation =  93.99 %
Training complete in 21m 49s
```

In [13]:
```
for row in dot_predictions.items():
    print(row)
```

```
('model_0', {'hidden_1': 392, 'hidden_2': 512, 'parameters': 514066, 'lr': 1.0, 'r_valid [%]': 96.5
6})
('model_1', {'hidden_1': 392, 'hidden_2': 768, 'parameters': 617234, 'lr': 1.0, 'r_valid [%]': 97.5
8})
('model_2', {'hidden_1': 392, 'hidden_2': 1024, 'parameters': 720402, 'lr': 1.0, 'r_valid [%]': 97.3
9})
('model_3', {'hidden_1': 512, 'hidden_2': 256, 'parameters': 535818, 'lr': 1.0, 'r_valid [%]': 97.5
7})
('model_4', {'hidden_1': 512, 'hidden_2': 512, 'parameters': 669706, 'lr': 1.0, 'r_valid [%]': 97.7
4})
('model_5', {'hidden_1': 512, 'hidden_2': 768, 'parameters': 803594, 'lr': 1.0, 'r_valid [%]': 97.6
5})
('model_6', {'hidden_1': 512, 'hidden_2': 1024, 'parameters': 937482, 'lr': 1.0, 'r_valid [%]': 97.
5})
('model_7', {'hidden_1': 768, 'hidden_2': 256, 'parameters': 802314, 'lr': 1.0, 'r_valid [%]': 97.7
4})
('model_8', {'hidden_1': 392, 'hidden_2': 512, 'parameters': 514066, 'lr': 0.1, 'r_valid [%]': 97.
5})
('model_9', {'hidden_1': 392, 'hidden_2': 768, 'parameters': 617234, 'lr': 0.1, 'r_valid [%]': 97.6
1})
('model_10', {'hidden_1': 392, 'hidden_2': 1024, 'parameters': 720402, 'lr': 0.1, 'r_valid [%]': 97.
54})
('model_11', {'hidden_1': 512, 'hidden_2': 256, 'parameters': 535818, 'lr': 0.1, 'r_valid [%]': 97.6
7})
('model_12', {'hidden_1': 512, 'hidden_2': 512, 'parameters': 669706, 'lr': 0.1, 'r_valid [%]': 97.6
9})
('model_13', {'hidden_1': 512, 'hidden_2': 768, 'parameters': 803594, 'lr': 0.1, 'r_valid [%]': 97.7
4})
('model_14', {'hidden_1': 512, 'hidden_2': 1024, 'parameters': 937482, 'lr': 0.1, 'r_valid [%]': 97.
67})
('model_15', {'hidden_1': 768, 'hidden_2': 256, 'parameters': 802314, 'lr': 0.1, 'r_valid [%]': 97.5
4})
('model_16', {'hidden_1': 392, 'hidden_2': 512, 'parameters': 514066, 'lr': 0.01, 'r_valid [%]': 93.
54})
('model_17', {'hidden_1': 392, 'hidden_2': 768, 'parameters': 617234, 'lr': 0.01, 'r_valid [%]': 93.
67})
('model_18', {'hidden_1': 392, 'hidden_2': 1024, 'parameters': 720402, 'lr': 0.01, 'r_valid [%]': 9
3.28})
('model_19', {'hidden_1': 512, 'hidden_2': 256, 'parameters': 535818, 'lr': 0.01, 'r_valid [%]': 93.
87})
('model_20', {'hidden_1': 512, 'hidden_2': 512, 'parameters': 669706, 'lr': 0.01, 'r_valid [%]': 93.
79})
('model_21', {'hidden_1': 512, 'hidden_2': 768, 'parameters': 803594, 'lr': 0.01, 'r_valid [%]': 93.
```

```
46})
('model_22', {'hidden_1': 512, 'hidden_2': 1024, 'parameters': 937482, 'lr': 0.01, 'r_valid [%]': 9
3.77})
('model_23', {'hidden_1': 768, 'hidden_2': 256, 'parameters': 802314, 'lr': 0.01, 'r_valid [%]': 93.
99})
```

From the different parameters, we find that for learning rate smaller than 0.1, the accuracy is lower after 10 epoch, i.e. training process is slower. For learning rate of 1, a few models - 512 x 512 (model_4) , 768 x 256 (model_7) reaches an accuracy of 97.74. For the learning rate of 0.1, the highest accuracy is reached by 512 x 786 (model_13) with an accuracy of 97.74 percent.

Lower learning rate helps in training better and thus we continue with 512 x 786 (model_13).

Validate Gradients using Finite Difference

1. Compute finite difference gradients using epsilon=1/N for different values of N. Use at least 5 values of N from the set{k10^i : i∈{0,...,5}, k∈{1,5}}.

In [14]:
```python
N_list = [10.0**i for i in range(6)]#[1,1.0e1,1.0e2,1.0e3,1.0e4,1.0e5,1.0e6,1.0e7]
epsilon_list = [1.0/i for i in N_list]
print(epsilon_list)
```

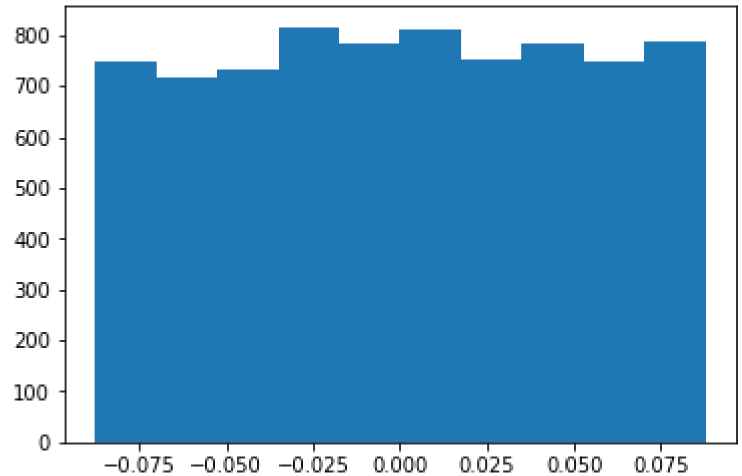
```
[1.0, 0.1, 0.01, 0.001, 0.0001, 1e-05]
```

In [15]:
```python
# model
NN1 = NN(hidden_dims=(512,768), weight_type = "Glorot", learning_rate = 0.1)
print("Loading model : ",NN1.h1,"x",NN1.h2,"lr : ", NN1.eta)
```

```
parameter count = 803594
Loading model :  512 x 768 lr :  0.1
```

In [16]:
```
plt.hist(NN1.W2.flatten())
```

Out[16]:
```
(array([748., 718., 733., 816., 785., 809., 753., 782., 748., 788.]),
 array([-8.77566535e-02, -7.02024187e-02, -5.26481839e-02, -3.50939491e-02,
        -1.75397143e-02,  1.45204861e-05,  1.75687553e-02,  3.51229901e-02,
         5.26772249e-02,  7.02314597e-02,  8.77856945e-02]),
 <a list of 10 Patch objects>)
```



In [17]:
```
# Taking one input point
ind = 0
x_i = train_data_features[ind].reshape(-1,1)
y_i = [train_data_labels[ind]]
```

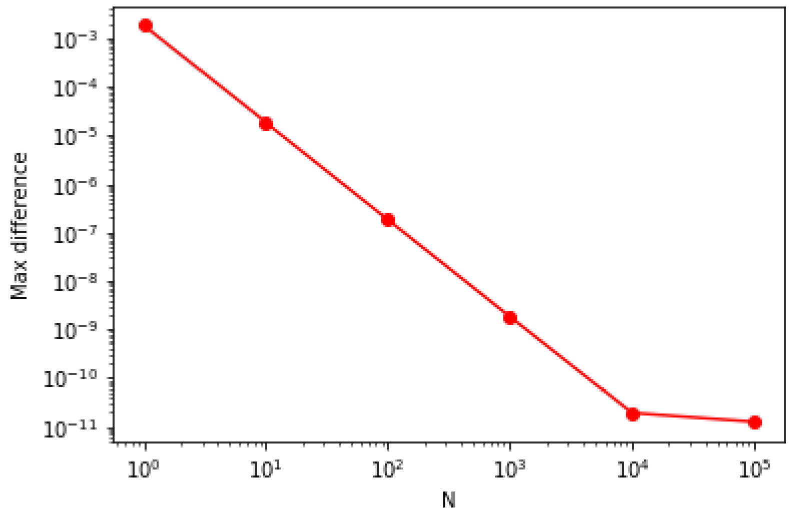
In [18]:
```
max_absolute_diff = []
for epsilon in epsilon_list:
    print("\nEpsilon = ",epsilon)
    absolute_diff = []
    for i in [0]:
        for j in range(10):

            NN1.W2[i][j] -= epsilon
            cache = NN1.forward(x_i)
            before = NN1.loss(cache['output'],y_i)
            NN1.W2[i][j] += epsilon

            NN1.W2[i][j] += epsilon
            cache = NN1.forward(x_i)
            after = NN1.loss(cache['output'],y_i)
            NN1.W2[i][j] -= epsilon

            finite_difference_gradient = (after-before)/(2*epsilon)

            cache = NN1.forward(x_i)
            grads = NN1.backward(cache,y_i)


            print('W2['+str(i)+','+str(j)+'] finite difference =', finite_difference_gradient
                  ,'\t finite difference/grads[W2]['+str(i)+','+str(j)+'] = ',
                  finite_difference_gradient/grads['W2'][i][j])

            ## computing the absolute difference between finite difference and backprop computed grad
ient.
            absolute_diff.append(np.abs(finite_difference_gradient - grads['W2'][i][j]))


    ## Maximum of the absolute differnce seen for this epsilon value.
    max_absolute_diff.append(np.max(absolute_diff))
```

```
Epsilon =  1.0
W2[0,0] finite difference = [0.]         finite difference/grads[W2][0,0] =  [nan]
W2[0,1] finite difference = [0.05484835]         finite difference/grads[W2][0,1] =  [1.0350778]
W2[0,2] finite difference = [0.]         finite difference/grads[W2][0,2] =  [nan]
W2[0,3] finite difference = [0.0067867]          finite difference/grads[W2][0,3] =  [1.00057534]
W2[0,4] finite difference = [0.01999971]         finite difference/grads[W2][0,4] =  [1.00495239]
W2[0,5] finite difference = [0.01013697]         finite difference/grads[W2][0,5] =  [1.00128175]
W2[0,6] finite difference = [0.]         finite difference/grads[W2][0,6] =  [nan]
W2[0,7] finite difference = [0.]         finite difference/grads[W2][0,7] =  [nan]
W2[0,8] finite difference = [0.00431851]         finite difference/grads[W2][0,8] =  [1.00023312]
W2[0,9] finite difference = [0.00253696]         finite difference/grads[W2][0,9] =  [1.00008048]

Epsilon =  0.1
W2[0,0] finite difference = [0.]         finite difference/grads[W2][0,0] =  [nan]
W2[0,1] finite difference = [0.0530082]          finite difference/grads[W2][0,1] =  [1.00035115]
W2[0,2] finite difference = [0.]         finite difference/grads[W2][0,2] =  [nan]
W2[0,3] finite difference = [0.00678284]         finite difference/grads[W2][0,3] =  [1.00000575]
W2[0,4] finite difference = [0.01990214]         finite difference/grads[W2][0,4] =  [1.00004953]
W2[0,5] finite difference = [0.01012413]         finite difference/grads[W2][0,5] =  [1.00001282]
W2[0,6] finite difference = [0.]         finite difference/grads[W2][0,6] =  [nan]
W2[0,7] finite difference = [0.]         finite difference/grads[W2][0,7] =  [nan]
W2[0,8] finite difference = [0.00431751]         finite difference/grads[W2][0,8] =  [1.00000233]
W2[0,9] finite difference = [0.00253676]         finite difference/grads[W2][0,9] =  [1.0000008]

Epsilon =  0.01
W2[0,0] finite difference = [0.]         finite difference/grads[W2][0,0] =  [nan]
W2[0,1] finite difference = [0.05298978]         finite difference/grads[W2][0,1] =  [1.00000351]
W2[0,2] finite difference = [0.]         finite difference/grads[W2][0,2] =  [nan]
W2[0,3] finite difference = [0.0067828]          finite difference/grads[W2][0,3] =  [1.00000006]
W2[0,4] finite difference = [0.01990117]         finite difference/grads[W2][0,4] =  [1.0000005]
W2[0,5] finite difference = [0.010124]   finite difference/grads[W2][0,5] =  [1.00000013]
W2[0,6] finite difference = [0.]         finite difference/grads[W2][0,6] =  [nan]
W2[0,7] finite difference = [0.]         finite difference/grads[W2][0,7] =  [nan]
W2[0,8] finite difference = [0.0043175]          finite difference/grads[W2][0,8] =  [1.00000002]
W2[0,9] finite difference = [0.00253675]         finite difference/grads[W2][0,9] =  [1.00000001]

Epsilon =  0.001
W2[0,0] finite difference = [0.]         finite difference/grads[W2][0,0] =  [nan]
W2[0,1] finite difference = [0.05298959]         finite difference/grads[W2][0,1] =  [1.00000004]
W2[0,2] finite difference = [0.]         finite difference/grads[W2][0,2] =  [nan]
W2[0,3] finite difference = [0.0067828]          finite difference/grads[W2][0,3] =  [1.]
W2[0,4] finite difference = [0.01990116]         finite difference/grads[W2][0,4] =  [1.]
W2[0,5] finite difference = [0.010124]   finite difference/grads[W2][0,5] =  [1.]
```

```
W2[0,6] finite difference = [0.]         finite difference/grads[W2][0,6] =  [nan]
W2[0,7] finite difference = [0.]         finite difference/grads[W2][0,7] =  [nan]
W2[0,8] finite difference = [0.0043175]          finite difference/grads[W2][0,8] =  [1.]
W2[0,9] finite difference = [0.00253675]         finite difference/grads[W2][0,9] =  [1.]

Epsilon =  0.0001
W2[0,0] finite difference = [0.]         finite difference/grads[W2][0,0] =  [nan]
W2[0,1] finite difference = [0.05298959]         finite difference/grads[W2][0,1] =  [1.]
W2[0,2] finite difference = [0.]         finite difference/grads[W2][0,2] =  [nan]
W2[0,3] finite difference = [0.0067828]          finite difference/grads[W2][0,3] =  [1.]
W2[0,4] finite difference = [0.01990116]         finite difference/grads[W2][0,4] =  [1.]

//anaconda/envs/helios/lib/python3.7/site-packages/ipykernel_launcher.py:26: RuntimeWarning: invalid
value encountered in true_divide

W2[0,5] finite difference = [0.010124]   finite difference/grads[W2][0,5] =  [1.]
W2[0,6] finite difference = [0.]         finite difference/grads[W2][0,6] =  [nan]
W2[0,7] finite difference = [0.]         finite difference/grads[W2][0,7] =  [nan]
W2[0,8] finite difference = [0.0043175]          finite difference/grads[W2][0,8] =  [1.]
W2[0,9] finite difference = [0.00253675]         finite difference/grads[W2][0,9] =  [1.]

Epsilon =  1e-05
W2[0,0] finite difference = [0.]         finite difference/grads[W2][0,0] =  [nan]
W2[0,1] finite difference = [0.05298959]         finite difference/grads[W2][0,1] =  [1.]
W2[0,2] finite difference = [0.]         finite difference/grads[W2][0,2] =  [nan]
W2[0,3] finite difference = [0.0067828]          finite difference/grads[W2][0,3] =  [1.]
W2[0,4] finite difference = [0.01990116]         finite difference/grads[W2][0,4] =  [1.]
W2[0,5] finite difference = [0.010124]   finite difference/grads[W2][0,5] =  [1.]
W2[0,6] finite difference = [0.]         finite difference/grads[W2][0,6] =  [nan]
W2[0,7] finite difference = [0.]         finite difference/grads[W2][0,7] =  [nan]
W2[0,8] finite difference = [0.0043175]          finite difference/grads[W2][0,8] =  [1.]
W2[0,9] finite difference = [0.00253675]         finite difference/grads[W2][0,9] =  [1.]
```

2. Plot the maximum difference between the true gradient and the finite difference gradient(max1≤i≤p|∇Ni−∂L∂θi|) as a function of N. Comment on the result.

In [19]:
```
plt.loglog(N_list,max_absolute_diff,'ro-')
plt.ylabel('Max difference')
plt.xlabel('N')
plt.show()
```



Here we can see that for larger N (i.e. smaller epsilon), the finite difference gradient is closer to the gradient obtained by back propagation. Therefore, we can confirm that our backpropagation function is working correctly.