Machine Learning Techniques

K S Ananthram

October 15, 2018

1 Introduction

Machine learning (ML) is a branch in computer science which describes how to learn from and make predictions about the data. Avavilibility of large datasets in modern science enable computers to learn based on statistical techniques. The main advantage of ML is that it gives progressive and improved performance on a specific task without human interference. The name machine learning was first introduced in 1952 by Arthur samuel. In later stage of development, the algorithms in ML which were basically built to study pattern recognition was improved on the basis of principles of statistical learninig theory in artificial inteligence to perform various tasks. At present ML techniques are playing increasingly important roles in modern technology ranging from computer artificial inteligent(AI) games to biotechnology, self-driving cars and smart devices. In physics where the data analysis plays its role, ML techniques are widely used. For example experimental particle physics, observational astronomy and cosmology, condensed matter physics, biophysics and quantum computing.

2 Why to study ML?

In general, there are several reasons to study and impliment computer based techniques to real world. Important reasons will be

- Modern computers have the ability to generate and analyse large datasets. Because they are equipped with computational power and memory.
- Availability of large datasets which are the source of information waiting to be interpreted and analysed. For example: Experiments such as CMS and ATLAS at LHC generate petabytes of data per year. Digital Sky Servey(SDSS) has terabytes of data about the properties of near a billion stars and galaxies.
- The most important reason will be, the core concepts and techniques in physics such as Monte-Carlo method, Variational method and the models in statistical physics are the backbone of many of ML and DL methods.

3 Deep learning

Deep learning is a subset of machine learning and functions in a similar way. A deep learning model can continuously analyse data and make decisions similar to how a human would think. To make this happen, deep learning uses an Artificial Neural Netwok (ANN) which uses a layered structure of algorithm. The design of ANN is inspired by the biological neural network of human brain.

4 Understanding DL

To understand working of deep learning, one must understand the concept of ANN and what makes an ANN?

4.1 Perceptron

Perceptron is a single layer neural network. In mathematical logic they are binary linear classifiers. A perceptron consists of 4 parts,

- Input values or one input layers
- Weights and biases
- Net Sum
- Activation function

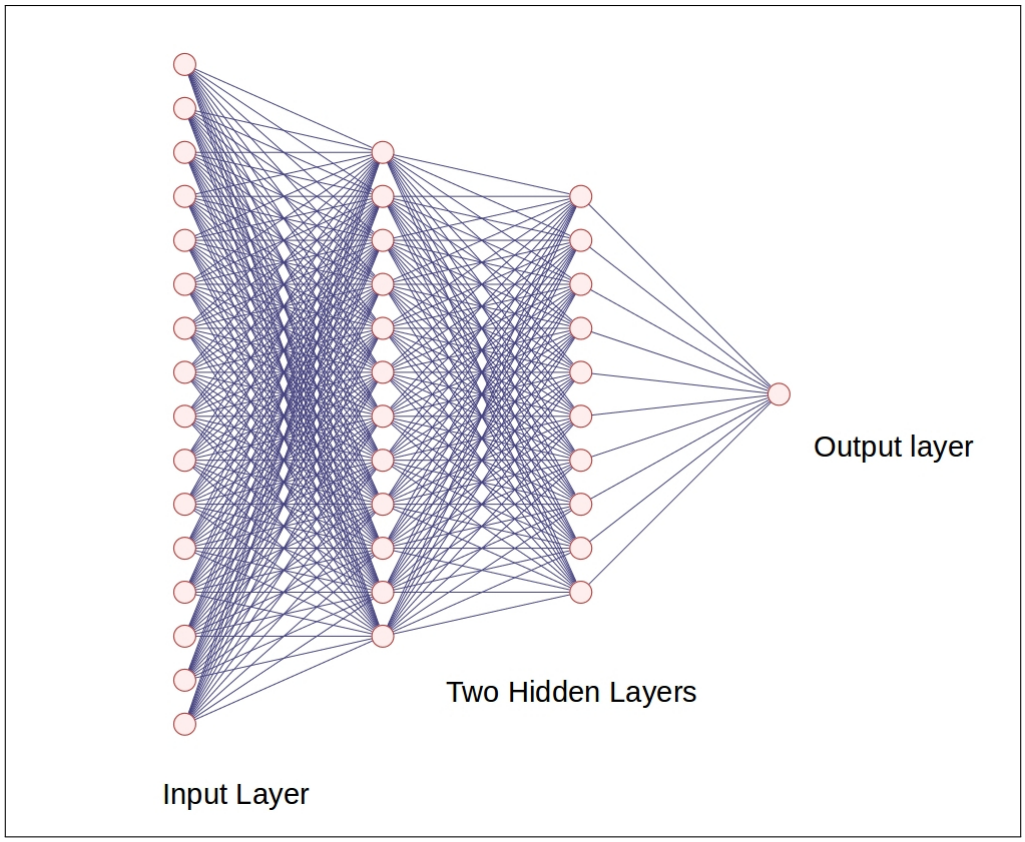
A single perceptron takes several binary inputs x_1,x_2, ... and produces a single binary output.
Weights w_1, w_2 ... are real numbers expressing the importance of respective inputs to output.
Net sum or Weighted sum is $\sum_i w_i x_i$.
The output is either '0' or '1' is determined by setting up a threshold value(a real number). Algebraically

$$\text{output} = \begin{cases} 0 & \text{if } \sum_i w_i x_i \leq threshold \\ 1 & \text{if } \sum_i w_i x_i > threshold \end{cases}$$

The threshold value can be either positive or negative. If we make it more and more negative, then the probability of respective output to be '1' increases. To put everything in mathematical sense, we introduce bias, which is equal to negative of threshold value. The output equation becomes

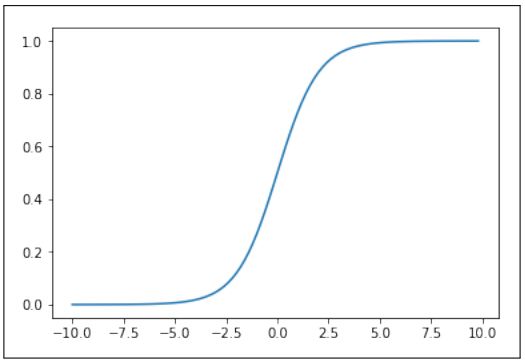
$$\text{output} = \begin{cases} 0 & \text{if } \sum_i w_i x_i + b \leq 0 \\ 1 & \text{if } \sum_i w_i x_i + b > 0 \end{cases}$$

The bias can easily determine the state of the output. Overall an ANN will have a layers of perceptrons. Each of those is making a decision by weighing up the inputs and taking the previous layer results as inputs to produce the output. In this way a many layer perceptrons can make desicions.



The above figure shows an ANN with one input layer, two hidden layers and one ouput layer. Each of the layers is made out of perceptrons, and they are interconnected from input layer to output layer through hidden layers. In general an ANN has one input layer and one output layer with intermediate 'n' number of hidden layers. This 'n' can be very large and each perceptron in the hidden layer will have a weight that ultimately carried to the output layer. Final Output is produced upon calculating the Weighted sum or by an activation function.

In general a neural network is a nonlinear function that depends on many varriables. Or one can say that a neural network is a combination of linear mapping with a very simple nonlinear function (activation function). Examples for activation functions are sigmoid function $f(z) = 1/(1 + e^{-z})$, a unit step function etc. Sigmoid activation function is shown bellow.



5 Visualization Of Simple Networks

Initially we will setup a neural network, and try to understand the output. Here we are taking two input layer and one output layer. And Sigmoid function is used as an activation function.

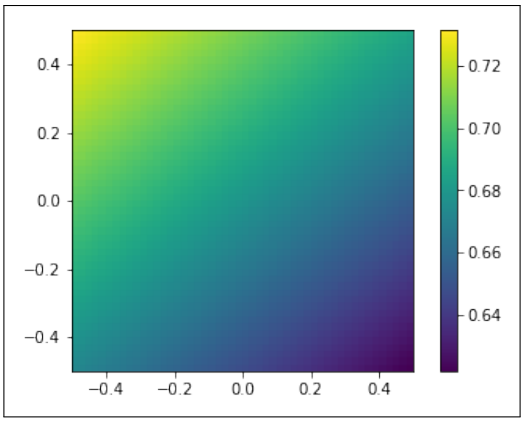
```
In [85]: from numpy import *
         N0=2 #input layer
         N1=1 #output layer

In [86]: W=random.uniform(low=-1,high=+1,size=(N1,N0)) #randomly assign weights
         B=random.uniform(low=-1,high=+1,size=N1) #randomly assign biases

In [87]: def apply_NET(Y_in):
             global W,B
             z=dot(W,Y_in)+B # matrix and a vector multiplication added to biases
             return(1/(1+exp(-z))) # calculated the output ; sigmoid function
```

Output is calculated as $Z = W.Y_{in} + B$, and activation function $f(Z)$ is plotted as output. Where W has two values for two inputs, which connects the weights from input to output.

```
In [88]: apply_NET([0.9,0.2])

Out[88]: array([0.72378781])

In [89]: import matplotlib.pyplot as plt
         %matplotlib inline

In [90]: M=50
         Y_out=zeros([M,M]) # create picture of 50X50 and write the results in it
         for j1 in range(M):
             for j2 in range(M):
                 value0=float(j1)/M-0.5
                 value1=float(j2)/M-0.5
                 Y_out[j1,j2]=apply_NET([value0,value1])[0]
```

We created a $50X50$ pixels image (in the form of a matrix) and write the output in every pixel using a 'for loop'.

```
In [91]: plt.imshow(Y_out,origin='lower',extent=(-0.5,0.5,-0.5,0.5))
         plt.colorbar()
         plt.show()
```



The plot tells us that after $j_2 = 0$ the activation function chooses different value then $j_2 < 0$. Typpically one can fix these values as '1' for $j_2 \geq 0$ and '0' for $j_2 < 0$. Since sigmoid function is a smoothly varrying function, a linear combination of many such sigmoid functions can be used to represent a very complex output with high resolution. To understand that see the example bellow.

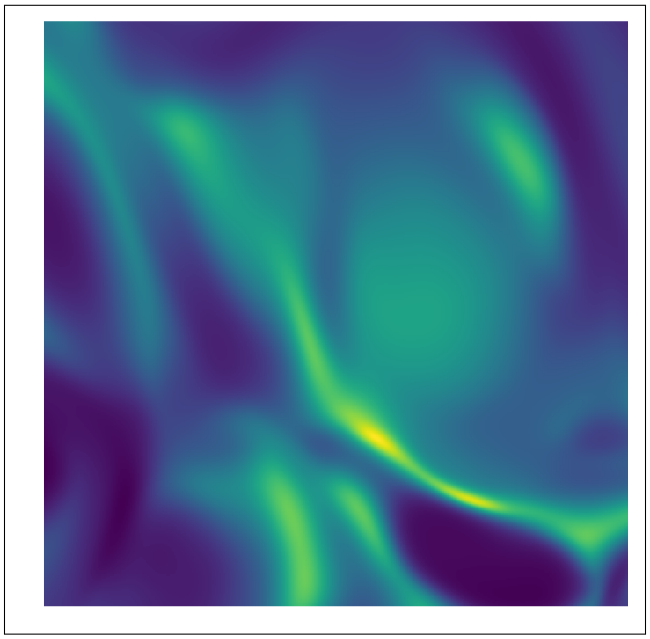
Here we have taken a neural network with 20 layers, each containing 100 neurons. And we have used the batchwise processing trick to minimize the computational time. All weights and biases are rondomly generated and a visual output of 400X400 pixel image is obtained.

```
In [61]: from numpy import *

In [62]: import matplotlib.pyplot as plt
         %matplotlib inline

In [63]: def apply_layer_new(y_in,w,b): # a function that applies a layer
             z=dot(y_in,w)+b # note different order in matrix multiplication
             return(1/(1+exp(-z)))

In [64]: Nlayers=20 # number of layer
         LayerSize=100

In [65]: Weights=random.uniform(low=-2,high=2,size=[Nlayers,LayerSize,LayerSize])
         Biases=random.uniform(low=-1,high=1,size=[Nlayers,LayerSize])

In [66]: WeightsFirst=random.uniform(low=-2,high=2,size=[2,LayerSize])
         BiasesFirst=random.uniform(low=-1,high=1,size=LayerSize)

In [67]: WeightsFinal=random.uniform(low=-2,high=2,size=[LayerSize,1])
         BiasesFinal=random.uniform(low=-1,high=1,size=1)
```

In [68]:
```
def apply_multi_net(y_in):
    global Weights, Biases, WeightsFinal, BiasesFinal, Nlayers

    y=apply_layer_new(y_in,WeightsFirst,BiasesFirst)
    for j in range(Nlayers):
        y=apply_layer_new(y,Weights[j,:,:],Biases[j,:])
    output=apply_layer_new(y,WeightsFinal,BiasesFinal)
    return(output)
```

In [69]:
```
M=400 # size of image
# Generate a 'mesh grid', i.e. x,y values in an image
v0,v1=meshgrid(linspace(-0.5,0.5,M),linspace(-0.5,0.5,M))
batchsize=M**2 # number of samples = number of pixels = M^2
y_in=zeros([batchsize,2])
y_in[:,0]=v0.flatten() # fill first component (index 0)
y_in[:,1]=v1.flatten() # fill second component
```

In [70]:
```
y_out=apply_multi_net(y_in) # apply net to all these samples!

y_2D=reshape(y_out[:,0],[M,M]) # back to 2D image

plt.figure(figsize=[10,10])
plt.axes([0,0,1,1]) # fill all of the picture with the image
plt.imshow(y_2D,origin='lower',extent=[-0.5,0.5,-0.5,0.5],
interpolation='nearest')
plt.axis('off') # no axes
plt.show()
```



We get a complex image. That means this above simple neural network can generate a complex meaningful output by a proper choice of weights and biases for the given input. The process of finding those proper wieghts and biases for which the network gives the desired output is called Training. There are different methods and different algorithms to train a network.

The underlying idea is to approximate any arbitrary smooth function (differentiable) using Neural network. And the training is done by learning alogorithms which uses labled data to trian the network.

6 Stochastic gradient descent

To optimize the weights to get a desired output in training, take the network output and desired output, construct a function called cost function, which is defined as

$$C(w) = \frac{1}{2}\langle \|F_N(y_{in}) - F_D(y_{in})\|^2 \rangle$$

Where $F_N(y_{in})$ is neural network output and $F_D(y_{in})$ is desired output. A cost function is the norm square average of difference between the neural network output and the desired output. Here the average is taken over all training inputs.

Cost function can determine how close is a constructed neural network output to the desired output. Or it tells the deviation from the actual output.

Cost function depends only on weights and biases. If we optimize our weights that implies that cost function will have a minimum value. This optimization is done by gradient descent method. Since we will be dealing with large input data, this method adopts an stochastic approximation to calculate the gradient of cost function. Instead of large number of input data a randomly choosen finite number of samples are taken to construct the cost function. optimization of weights is done by repeatedly correcting the weights in each layer in iterations so that cost function converges to a very small number.

If W_n are the weights in n^{th} iteration, then in $(n+1)^{th}$ iteration, weights will have a value

$$W_{n+1} = W_n - \eta \frac{\partial C}{\partial W_n}$$

Where η is called stepsize parameter (decides the convergence of the cost function).

The same step is repeated till the cost function attains a convergent global minima. And that is the condition for optimized weights. This process is called stochastic gradient descent.

7 Calculation of derivatives and Backpropogation algorithm

In the privious section we see that to optimize weights we need to calculate the derivative of cost function with respect to each weights. In this section we will see how we can calculate derivatives.

Consider a simple network with two inputs (y_1, y_2) and an output $f(z)$. Where

$$z = y_1 w_1 + y_2 w_2 + b$$

with w_1, w_2, b are weights and bias.
Cost function

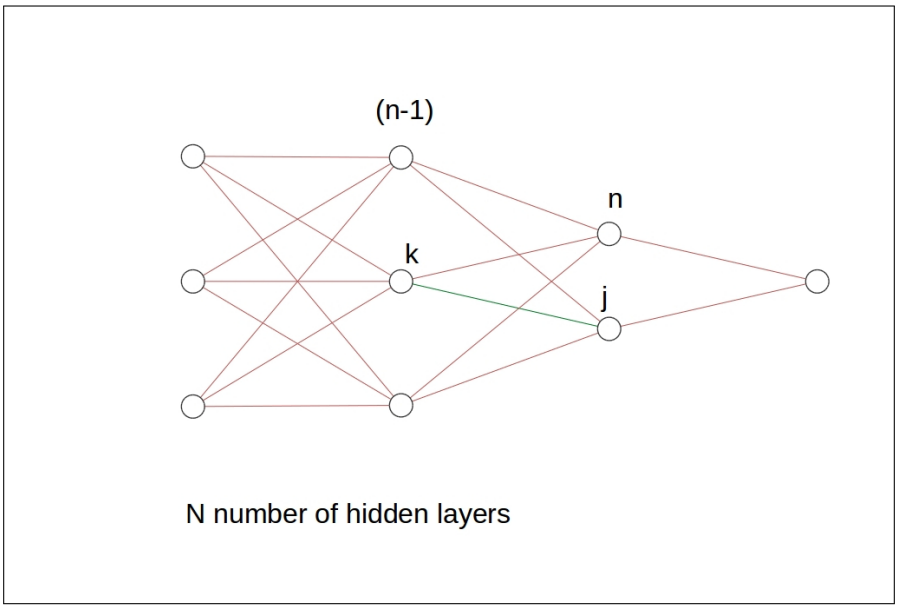
$$C(w) = \frac{1}{2}\langle \|f(z) - F(y_1, y_2)\|^2 \rangle$$

The derivative will be

$$\frac{\partial C}{\partial w_1} = \langle (f(z) - F) f'(z) \frac{\partial z}{\partial w_1} \rangle$$

Derivative is calculated by chain rule and here $\partial z / \partial w_1 = y_1$. For the weights and bias the same procedure is followed. Notice that for biases $\partial z / \partial b = 1$.

Now we can extend the same idea to a general neural network with 'n' hidden layers.



From the figure one can notice,

- If y_j^n is the value of the neuron in n^{th} layer.
- If z_j^n is the linear superposition output of j^{th} neuron in layer 'k'. z_j^n is input for $y_j^n = f(z_j^n)$
- The connected weight is $w_{jk}^{n,n-1}$; weights of neuron 'k' in $(n-1)^{th}$ layer feeding into neuron 'j' in in n^{th} layer.

The cost function we need to differentiate is

$$C(w) = \langle C(w, y_{in}) \rangle$$

is a sample specific cost function (depends on a particular input and all weights). Define derivative with respect to some weight w_* (or bias) as

$$\begin{aligned}\frac{\partial C(w, y_{in})}{\partial w_*} &= \sum_j (y_j^{(n)} - F_j(y_{in})) \frac{\partial y_j^{(n)}}{\partial w_*} \\ &= \sum_j (y_j^{(n)} - F_j(y_{in})) f'(z_j^{(n)}) \frac{\partial z_j^{(n)}}{\partial w_*}\end{aligned}$$

Here we have used chain rule and $y_j^{(n)} = f(z_j^{(n)})$ Important point to notice here is,

- If the weight $w_* = w_{jk}$ connecting j^{th} and k^{th} neuron in 'n' and '$(n-1)^{th}$ layer. Then $\partial z_j^{(n)} / \partial w_*$ can be evaluated directly.
- If the weight w_* is a weight connecting any two neurons in any previous layer, then $z_j^{(n)}$ dependson the value of w_*. Hence it needed to be evaluated using chain rule (because $z_j^{(n)}$ is a linear superposition of all the weights in the netwok and inputs).

$$\begin{aligned}\frac{\partial z_j^{(n)}}{\partial w_*} &= \sum_k \frac{\partial z_j^{(n)}}{\partial y_k^{(n-1)}} \frac{\partial y_k^{(n-1)}}{\partial w_*} \\ &= \sum_k w_{jk}^{(n,n-1)} f'(z_k^{(n-1)}) \frac{\partial z_k^{(n-1)}}{\partial w_*}\end{aligned}$$

This is a recursion relation.

Each pair of layers [n, n-1] contributes a matrix of the form,

$$M_{jk}^{(n,n-1)} = w_{jk}^{(n,n-1)} f'(z_k^{(n-1)})$$

If we proceed then, repeated matrix multiplication will propogate backwards from n^{th} hidden layer to the 1^{st} layer (if the weight w_* is a weight connecting two neurons from first layer to the second layer). The general expression for the above recursive matrix multiplication for a weight w_* being a weight connecting two neurons in the layer $(\tilde{n})$ and $(\tilde{n}-1)$ will be

$$\frac{\partial z_j^{(n)}}{\partial w_*} = \sum_{k,l\ldots u,v} M_{jk}^{(n,n-1)} M_{kl}^{(n-1,n-2)} \ldots M_{uv}^{(\tilde{n}+1,\tilde{n})} \frac{\partial z_v^{(\tilde{n})}}{\partial w_*}$$

For example if w_* is a weight in 1^{st} layer, then

$$\frac{\partial z_v^{(1)}}{\partial w_{12}^{2,1}} = y_2^{(1)}$$

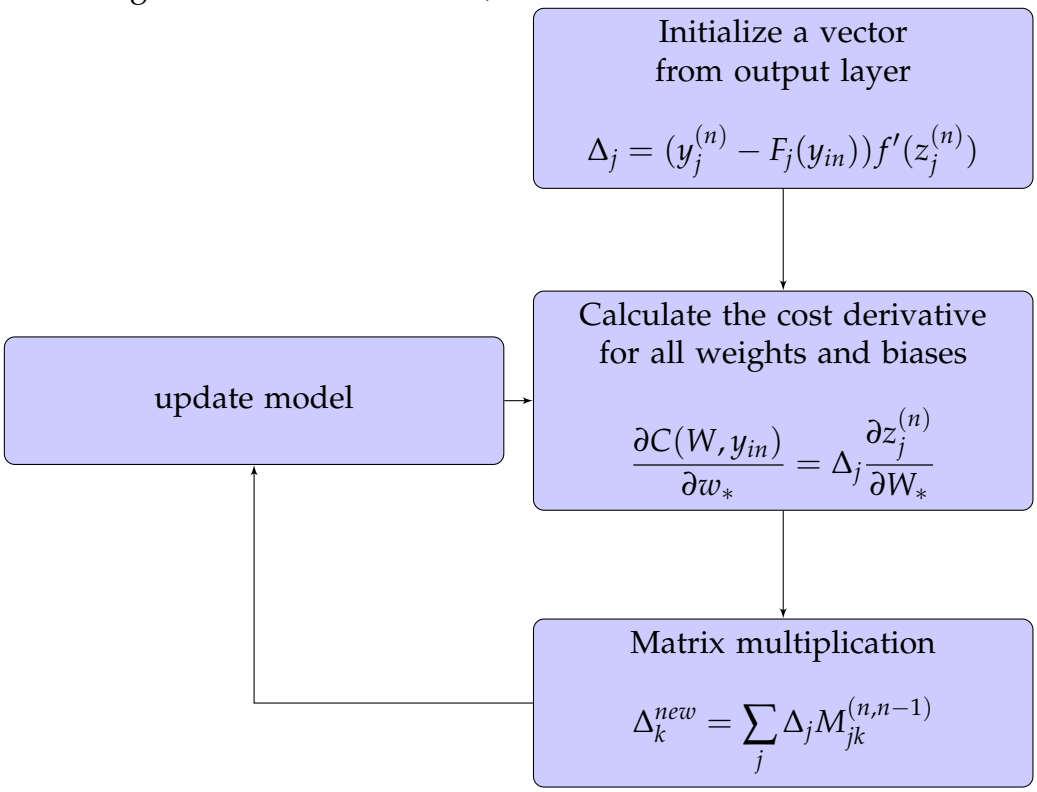
Or in general for a weight:

$$\frac{\partial z_v^{(\tilde{n})}}{\partial w_{vk}^{\tilde{n},\tilde{n}-1}} = y_k^{(\tilde{n}-1)}$$

For a bias:

$$\frac{\partial z_v^{(\tilde{n})}}{\partial b_v^{\tilde{n}}} = 1$$

Summerizing this idea in a flow chart,



7.1 Backpropogation

One single backpropogation pass through the network yeilds all the derivatives of the cost function with respect to all the weights and biases. One can implement this alogorithm efficiently by using batch processing.

Take a sample specific cost function $C(w, y_{in})$ and average over all samples,

$$\frac{\partial C(w)}{\partial w_*} = \langle \Delta_j \frac{\partial z_j^{(n)}}{\partial w_*} \rangle$$

For w_* to be weights,

$$\frac{\partial C(w)}{\partial w_*} = \langle \Delta_j y_k^{(n-1)} \rangle$$

For w_* to be biases,

$$\frac{\partial C(w)}{\partial w_*} = \langle \Delta_j \rangle$$

7.1.1 Batch processing of many samples

If y[layer] is a vector in a specific layer of dimension=number of neurons in that layer. And if we call number of samples as batchsize then, one can implement batch processing by a matrix multiplication in python. Some of the changes we need to do so in case of dimensionality of input and weight matrices are listed bellow.

Variable	Dimension
y[layer]	$Batchsize \times Neurons[layer]$
Delta (deviation from the correct output)	$Batchsize \times Neurons[layer]$
Weights[layer]	$Neuron[lowerlayer] \times Neuron[layer]$
Biases[layer]	Neuron[layer]

Basic python code which can calculate the sample average for above described process is given by

dWeights[layer]=dot[transpose(y[layer],Delta)/Batchsize $\Rightarrow \frac{\partial C(w)}{\partial w_*} = \langle \Delta_j y_k^{(n-1)} \rangle$

dBiases[layer]=Delta.sum(0)/Batchsize $\Rightarrow \frac{\partial C(w)}{\partial b_j^{(n)}} = \langle \Delta_j \rangle$

If we can take care of dimensions involving matrix multiplications then we can implement Backpropogation alorithm.

7.1.2 Example I-

```
In [894]: from numpy import * # get the "numpy" library for linear algebra

In [895]: import matplotlib.pyplot as plt # for plotting
          %matplotlib inline
          # the second line tells jupyter to display pictures within notebook

In [896]: def net_f_df(z): # calculate f(z) and f'(z)
              val=1/(1+exp(-z))
              return(val,exp(-z)*(val**2)) # return both f and f'

In [897]: def forward_step(y,w,b): # calculate values in next layer, from input y
              z=dot(y,w)+b # w=weights, b=bias vector for next layer
              return(net_f_df(z)) # apply nonlinearity and return result

In [898]: def apply_net(y_in): # one forward pass through the network
              global Weights, Biases, NumLayers
              global y_layer, df_layer # for storing y-values and df/dz values

              y=y_in # start with input values
              y_layer[0]=y
              for j in range(NumLayers): # loop through all layers [not counting input]
                  # j=0 corresponds to the first layer above the input
                  y,df=forward_step(y,Weights[j],Biases[j]) # one step, into layer j
                  df_layer[j]=df # store f'(z) [needed later in backprop]
                  y_layer[j+1]=y # store f(z) [also needed in backprop]
              return(y)

In [899]: def apply_net_simple(y_in): # one forward pass through the network
              # no storage for backprop (this is used for simple tests)

              y=y_in # start with input values
              y_layer[0]=y
              for j in range(NumLayers): # loop through all layers
                  # j=0 corresponds to the first layer above the input
                  y,df=forward_step(y,Weights[j],Biases[j]) # one step
              return(y)

In [900]: def backward_step(delta,w,df):
              # delta at layer N, of batchsize x layersize(N))
              # w between N-1 and N [layersize(N-1) x layersize(N) matrix]
              # df = df/dz at layer N-1, of batchsize x layersize(N-1)
              return( dot(delta,transpose(w))*df )
```

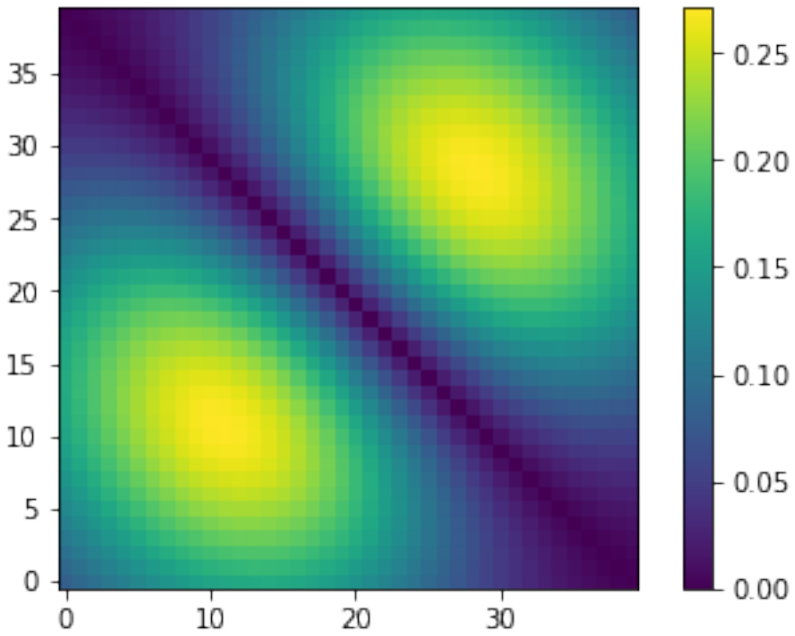
```
In [901]: def backprop(y_target): # one backward pass through the network
              # the result will be the 'dw_layer' matrices that contain
              # the derivatives of the cost function with respect to
              # the corresponding weight
              global y_layer, df_layer, Weights, Biases, NumLayers
              global dw_layer, db_layer # dCost/dw and dCost/db (w,b=weights,biases)
              global batchsize

              delta=(y_layer[-1]-y_target)*df_layer[-1]
              dw_layer[-1]=dot(transpose(y_layer[-2]),delta)/batchsize
              db_layer[-1]=delta.sum(0)/batchsize
              for j in range(NumLayers-1):
                  delta=backward_step(delta,Weights[-1-j],df_layer[-2-j])
                  dw_layer[-2-j]=dot(transpose(y_layer[-3-j]),delta)/batchsize
                  db_layer[-2-j]=delta.sum(0)/batchsize

In [902]: def gradient_step(eta): # update weights & biases (after backprop!)
              global dw_layer, db_layer, Weights, Biases

              for j in range(NumLayers):
                  Weights[j]-=eta*dw_layer[j]
                  Biases[j]-=eta*db_layer[j]

In [903]: def train_net(y_in,y_target,eta): # one full training batch
              # y_in is an array of size batchsize x (input-layer-size)
              # y_target is an array of size batchsize x (output-layer-size)
              # eta is the stepsize for the gradient descent
              global y_out_result

              y_out_result=apply_net(y_in)
              backprop(y_target)
              gradient_step(eta)
              cost=0.5*((y_target-y_out_result)**2).sum()/batchsize
              return(cost)

In [904]: # implement a ReLU unit (rectified linear), which
          # works better for training in this case
          def net_f_df(z): # calculate f(z) and f'(z)
              val=z*(z>0)
              return(val,z>0) # return both f and f'

In [905]: # set up all the weights and biases

          NumLayers=5 # does not count input-layer (but does count output)
          LayerSizes=[2,30,30,30,30,1] # input-layer,hidden-1,hidden-2,...,output-layer
```

```
Weights=[random.uniform(low=-0.5,high=+0.5,size=[ LayerSizes[j],LayerSizes[j+1
Biases=[zeros(LayerSizes[j+1]) for j in range(NumLayers)]
```

In [906]:
```
# set up all the helper variables

y_layer=[zeros(LayerSizes[j]) for j in range(NumLayers+1)]
df_layer=[zeros(LayerSizes[j+1]) for j in range(NumLayers)]
dw_layer=[zeros([LayerSizes[j],LayerSizes[j+1]]) for j in range(NumLayers)]
db_layer=[zeros(LayerSizes[j+1]) for j in range(NumLayers)]
```

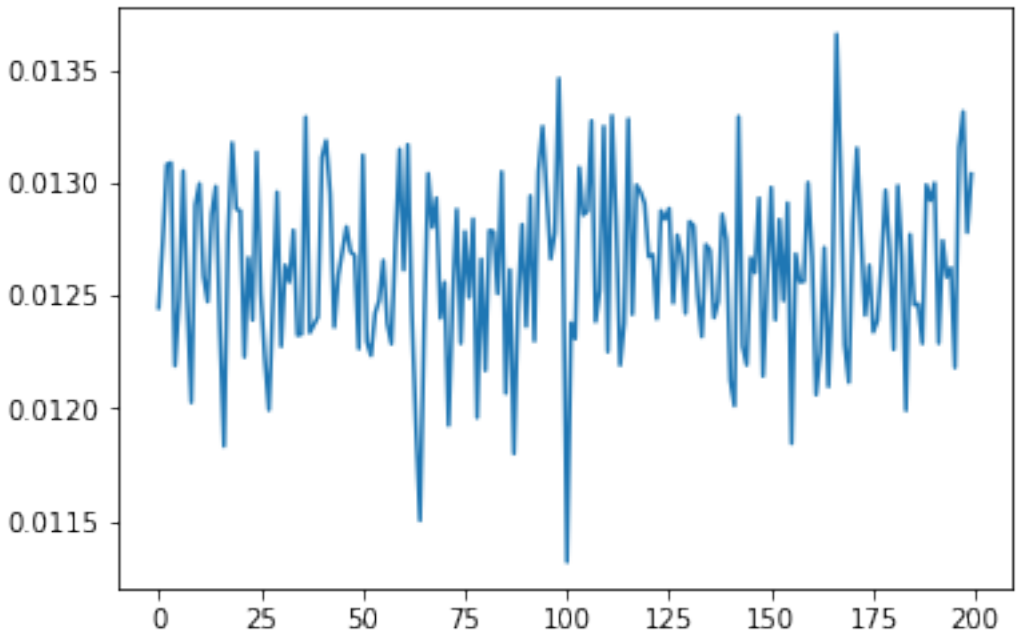
In [907]:
```
# define the batchsize
batchsize=1000
```

In [908]:
```
# the function we want to have (desired outcome)
def myFunc(x0,x1):
    r2=x0**2+x1**2
    return(exp(-5*r2)*abs(x1+x0))

xrange=linspace(-0.5,0.5,40)
X0,X1=meshgrid(xrange,xrange)
plt.imshow(myFunc(X0,X1),interpolation='nearest',origin='lower')
plt.colorbar()
plt.show()
```



In [909]:
```
# pick 'batchsize' random positions in the 2D square
def make_batch():
    global batchsize

    inputs=random.uniform(low=-0.5,high=+0.5,size=[batchsize,2])
    targets=zeros([batchsize,1]) # must have right dimensions
    targets[:,0]=myFunc(inputs[:,0],inputs[:,1])
    return(inputs,targets)
```

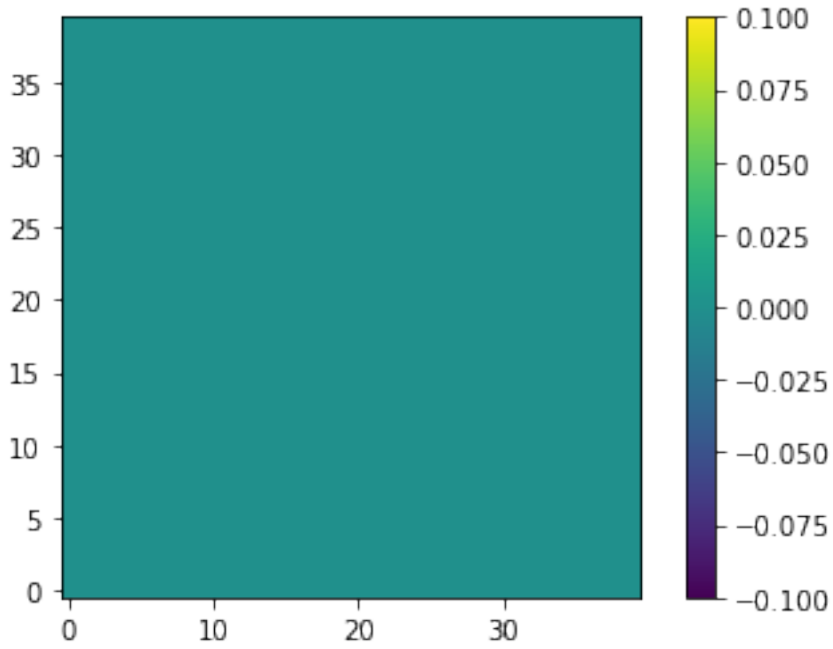
In [910]:
```
# Now: the training! (and plot the cost function)
eta=.1
batches=200
costs=zeros(batches)

for k in range(batches):
    y_in,y_target=make_batch()
    costs[k]=train_net(y_in,y_target,eta)

plt.plot(costs)
plt.show()
```

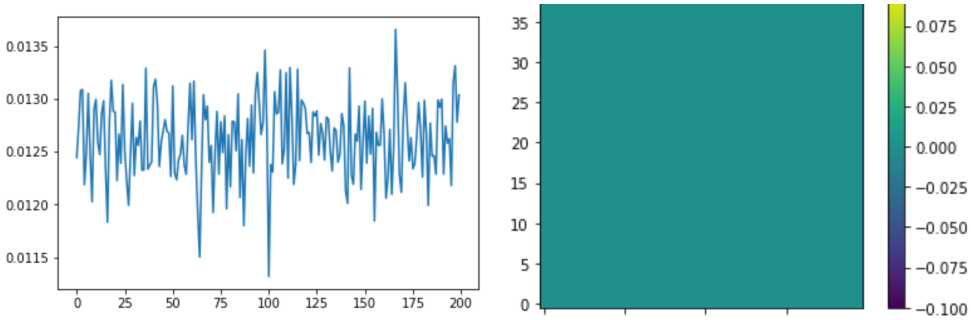


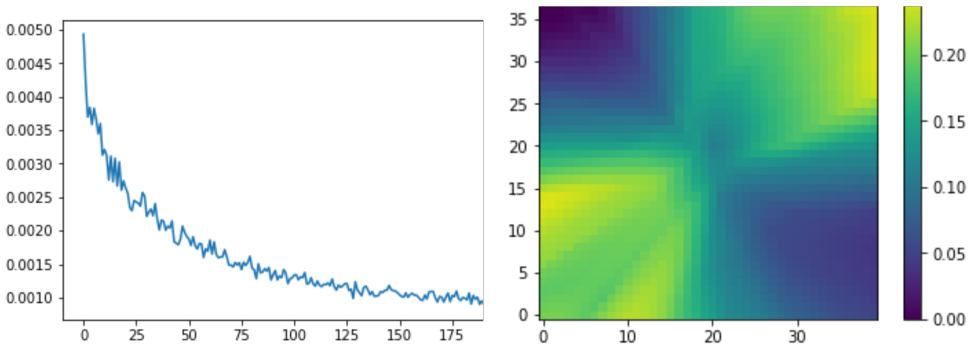
In [911]:
```
# a 'test' batch that includes all the points on the image grid
test_batchsize=shape(X0)[0]*shape(X0)[1]
testsample=zeros([test_batchsize,2])
```

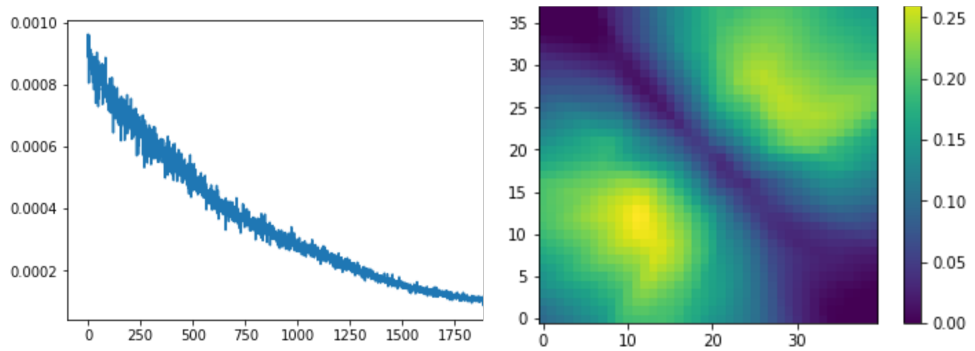
```
testsample[:,0]=X0.flatten()
testsample[:,1]=X1.flatten()
```

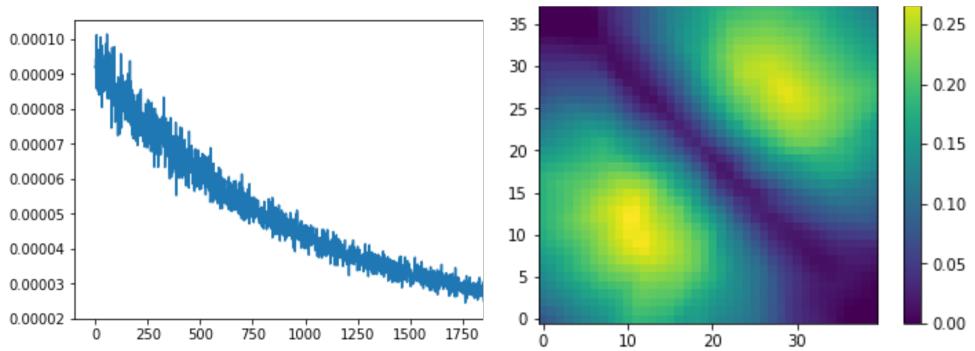
In [912]:
```
# show the output of this net
testoutput=apply_net_simple(testsample)
myim=plt.imshow(reshape(testoutput,shape(X0)),origin='lower',interpolation='no
plt.colorbar()
plt.show()
```

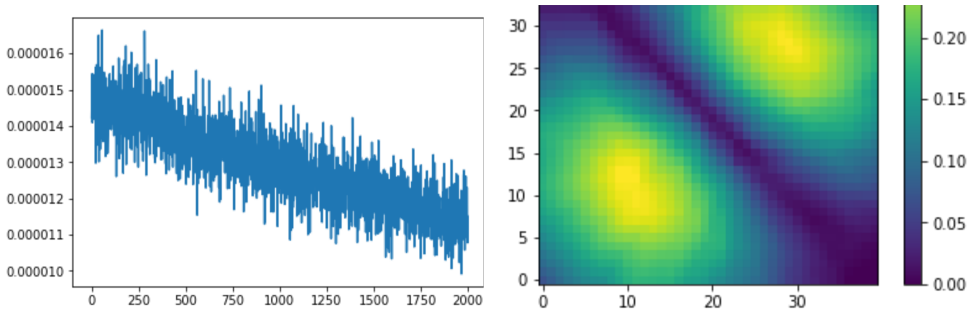


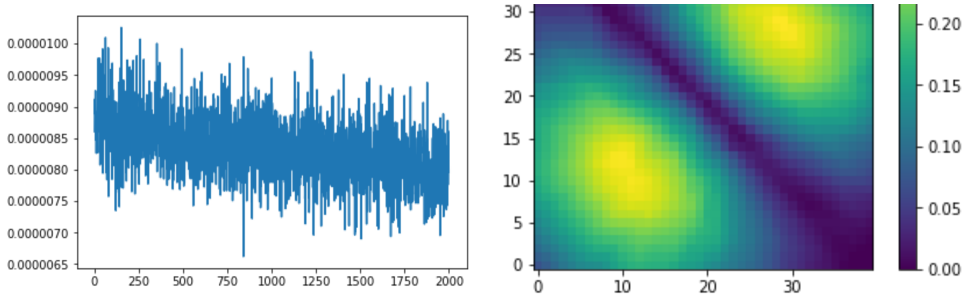



0.0050
0.0045
0.0040
0.0035
0.0030
0.0025
0.0020
0.0015
0.0010
0 25 50 75 100 125 150 175
35 30 25 20 15 10 5 0
0 10 20 30
0.20 0.15 0.10 0.05 0.00


0.0010
0.0008
0.0006
0.0004
0.0002
0 250 500 750 1000 1250 1500 1750
35 30 25 20 15 10 5 0
0 10 20 30
0.25 0.20 0.15 0.10 0.05 0.00


0.00010
0.00009
0.00008
0.00007
0.00006
0.00005
0.00004
0.00003
0.00002
0 250 500 750 1000 1250 1500 1750
35 30 25 20 15 10 5 0
0 10 20 30
0.25 0.20 0.15 0.10 0.05 0.00


0.000016
0.000015
0.000014
0.000013
0.000012
0.000011
0.000010
0 250 500 750 1000 1250 1500 1750 2000
30 25 20 15 10 5 0
0 10 20 30
0.20 0.15 0.10 0.05 0.00


0.0000100
0.0000095
0.0000090
0.0000085
0.0000080
0.0000075
0.0000070
0.0000065
0
250
500
750
1000
1250
1500
1750
2000
30
25
20
15
10
5
0
0
10
20
30
0.20
0.15
0.10
0.05
0.00