Deep Learning

1. Logistic Regression(Binary Classification)

input : x(vector)

expected output : y(0 or 1)

output : y’[0,1] or a

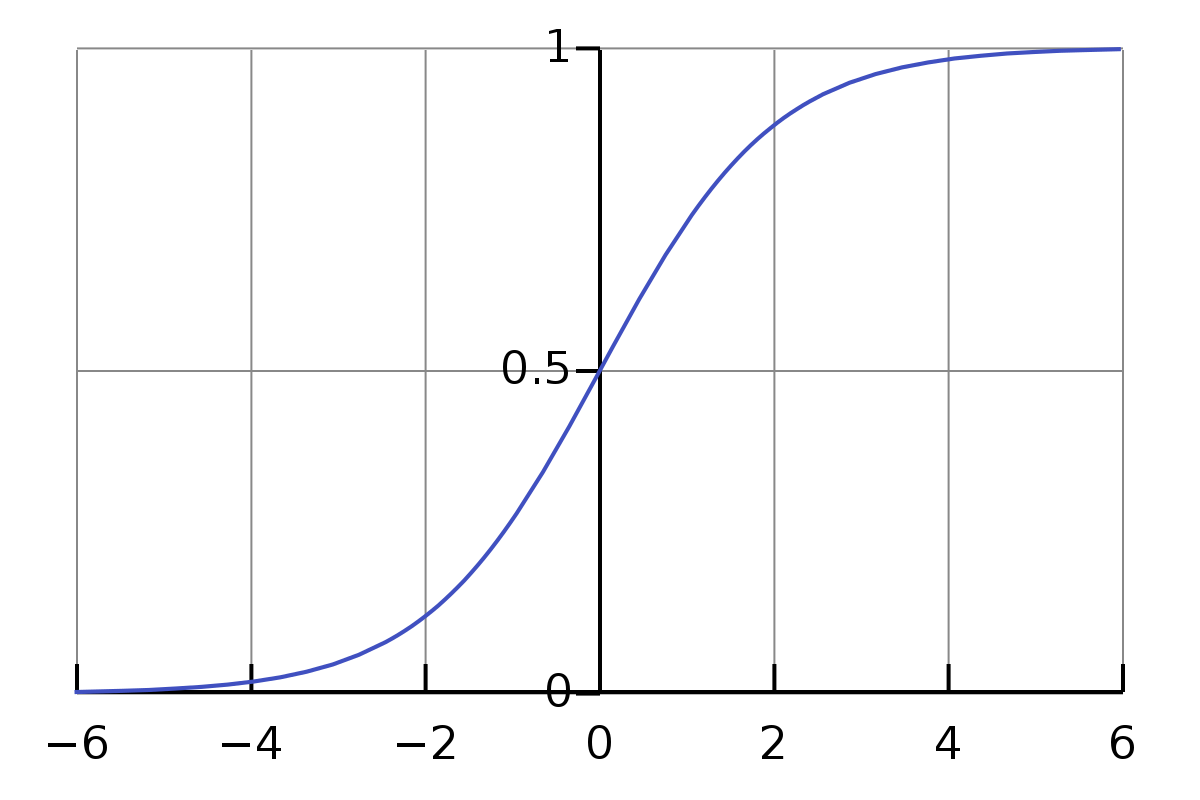
given x(1) :

z(1) = wt x(1)+b (where w(vector), b(real) are parameters)

if y(1) = 1: y’(1) = p( y(1)  | x(1)) (where 0<=y<=1)

if y(1) = 0: 1 - y’(1)= p( y(1)  | x(1))

=>p(y | x) = y’y \*(1-y’)(1-y)

 y’(1)= σ(z(1))

sigmoid function => σ(z) = 1/(1+ e-z)

z is large (+∞) => y’(1)=1

z is small( -∞) => y’(1)=0

∂ L(y’,y)/∂z = y’- y

∂ y’/∂z = y’(1-y’)

**Loss Function**

In optimization problem, Loss function should be convex(single local optima)

L(y’,y)= -( ylog(y’) + (1-y)log(1-y’) )

**Cost Funtion**

J(w,b)=(1/m)\* mΣi=1 L(y’(i),y(i))

**Gradient Descent**

w = w - α\* ( ∂ J(w,b)/∂w ) (where α is learning rate)

b = b - α \* ( ∂ J(w,b)/∂b )

**Algorithm**

j=0,dw=numpy.zeros(nx,1),db=0

for i = 1 to m

{

z(i) = wtx(i)+b

y’(i) = σ(z(i))

j += - (y’(i)\*log(y(i)) + (1-y’(i))\*log(1-y(i)) )

dz(i) = y’(i) – y(i)

dw += x(i)\*dz(i) // here dw & x(i) are a vectors

db += dz(i)

}

j /= m

dw /= m

db /= m

w = w - α\*dw

b = b - α\*db

**Vectorization**

It is faster due to parallel processing(SIMD - single instruction multiple data) of CPU and GPU.

z = numpy.dot(w,x)+b // z = wtx+b

u = numpy.exp(v) // v = [v1,v2,v3..vn] u=[ev1,ev2,ev3,... evn]

u = numpy.log(v)

u = numpy.abs(v)

u = numpy.maximum(v,0)

u = v\*\*2

u = 1/v

**Better Algorithm using vectorization**

// Z = [z(1),z(2),...,z(n)]

// b = [b,b,...,b] broadcasting

// dz=[dz1,dz2,dz3,..,dzn]

// Y = [y1,y2,y3,..,yn]

X = [x(1),x(2),..,x(n)]

Z = numpy.dot(w.T,X)+b //.T =>transpose

Y’ = σ(Z)

dZ = Y’-Y

dW= numpy.dot(X,dZ.T)/m //changed W to X

db= np.sum(dZ)/m

w = w - α\*dW

b = b - α\*db

**Neural Network Representation**

z1[1] = w1[1].T\*a1[0]+b1[1] => a1[1] = σ(z1[1])

z2[1] = w2[1].T\*a2[0]+b2[1] => a2[1] = σ(z2[1]) (layer 1,node 2)

w[1] = [ w1[1].T

w2[1].T

.......

.......

wn[1].T]

b[1] = [ b1[1]

b2[1]

.......

.......

bn[1] ]

z[1] = np.dot(w[1],a[0])+b[1]

a[1] = σ(z[1])

z[2] = np.dot(w[2],a[1])+b[2]

a[2] = σ(z[2])

for m training examples

for i from 1 to m:

{

z[1](i) = np.dot(w[1],a[0](i))+b[1]

a[1](i) = σ(z[1](i))

z[2](i) = np.dot(w[2],a[1](i))+b[2]

a[2](i) = σ(z[2](i))

}

vectorized

A[1] = [ [a[1](1)] [a[1](2)] [a[1](3)] ... [a[1](m)] ]

Z[1] = [ [z[1](1)] [z[1](2)] [z[1](3)] ... [z[1](m)] ]

Z[1] = np.dot(w[1],A[0])+b[1]

A[1] = σ(Z[1])

Z[2] = np.dot(w[2],A[1])+b[2]

A[2] = σ(Z[2])

**Activation Functions**

sigmoid function(binary classification)

tanh function(a=tanh(z)=(ez-e-z)/(ez+e-z)) )

Rectified linear unit(ReLU – max(0,z))

leaky ReLU ( max(0.01z,z) )(no dead neuron problem)

dZ[1] = ∂ L(y’,y)/∂Z[1] = np.dot(W[2].T,dZ[2]) \* g’[1](Z[1])

**Back Propagation in Deep Neural Network**

dz[l] = da[l] \* g[l]’(z[l])

dw[l] = dz[l] \* a[l-1].T

db[l] = dz[l]

da[l-1] = w[l].T \* dz[l] //not required in coding

dz[l-1] = np.dot(w[l].T , dz[l]) \* g[l-1]’(z[l-1])

Vectorised

dZ[l] = dA[l] \* g[l]’(Z[l])

dW[l] = np.dot(dZ[l] , A[l-1].T)/m

db[l] = np.sum(dZ[l])/m

dA[l-1] = np.dot(W[l].T , dZ[l] ) //not required in coding

dZ[l-1] = np.dot(W[l].T , dZ[l]) \* g[l-1]’(Z[l-1])

**Notation**:

* Superscript [l] denotes a quantity associated with the lth layer.
  + Example: a[L] is the Lth layer activation. W[L] and b[L] are the Lth layer parameters.
* Superscript (i) denotes a quantity associated with the ith example.
  + Example: x(i) is the ith training example.
* Lowerscript i denotes the ith entry of a vector.
  + Example: a[l]i denotes the ith entry of the lth layer's activations).

**Training** Set – used to calculate W,B using cost funtion (60% if data is less)

**Development** Set – used to calculate which algo does the best (20%)

**Test** Set – used to test the predictions of the selected algorithm.

High **variance** – overfitting the data(best in training set but very bad in test set)

high **bias** – very bad predictions(even in training set, error is very high)

**Basic Recipe for machine learning**

high bias

* bigger network
* train it longer
* try more advanced optimization algo
* new network architecture

high variance

* get more data
* regularization (create bias – variance trade off(only if network is small))
* new architechture

**Regularization**

cost function

J(w,b) = (1/m) x mΣi=1 L(y’(i),y(i)) + (λ/2m) ||w||22

λ(hyper parameter) – regularisation parameter

* L2 regularization(common)

euclidean norm or L2 norm - ||w||22 =  nΣi=1 wi2 = np.dot(w.T,w)

* L1 regularization(makes most of w 0)

L1 norm - (λ/2m) ||w||1 = (λ/2m) \* nΣi=1 |wi|

Forbenius norm of a matrix

J(w[l ],b) = (1/m) x mΣi=1 L(y’(i),y(i)) + (λ/2m) LΣi=1||w[l ]||2F

||w[l]||F2 = n[l]Σi=1n[l−1]Σj=1(wi,j[l])2

dw[l] = from back propagation + (λ/m)w[l]

also called weight decay because w = (1-αλ/m)w – α\*dW

since w is less(nearly 0), so many neurons will be dead, so the network becomes less complex and z also approaches 0, so activation function approaches linear.

Dropout regularization

drops some of the neurons as random and makes the network simple.

By dropout, each neuron will depend less each other neurons

Inverted dropouts

D[n] = np.random.rand(a[n].shape[0],a[n].shape[1])<keepProb[n]

a[n] = np.multiply(a[n],d[n])

a[n] /= keepProb

keepProb should be high for layers having high matrix size(w)

test time doesnt need dropout

data augmentation

reshaping,flipping,zooming,distortion the image to get more inputs.

Early stopping

stop training when dev set error increases.

**Optimization**

normalization of input

when the variables vary differently(x1(1,1000),x2(0,1)) if normalised learning will be faster.

X = (X – μ)/σ

μ = (1/m)nΣi xi

σ2 = (1/m)nΣi (xi - μ)2

vanishing and exploding gradients(weight initialisation for deep network)

make the weigths near to 1.

variance(w) = 1/n -> better(Xavier initialisation)

variance(w) = 2/n -> better for relu

w[l] = np.random.randn(shape)\*np.sqrt(1 or 2/n[l-1])

minibatches

dividing the trainingset into diffeerent mini batches, so as to speed up the training.

Y will be divided into Y{1},Y{2},Y{3},...,Y{t}

epoch – no of iterations through the training set

in batch processing 1 epoch => 1 gradient descent step

in minibatch processing 1 epoch => t gradient descent steps

suggested minibatch – 64,128,256,512

exponentially weighted average: vt = βvt-1+(1-β)θt

bias correction: vt= vt/(1-βt)

exponential weighted average in gradient descent:

vdw = βvdw + (1-β)dw ~ βvdw + dw

w=w-αvdw

RMS prop

Sdw= βSdw + (1-β)dw2

w=w-α dw/(√Sdw+10-8)

b=b-α db/(√Sdb+10-8)

Adams Optimisation

vdw =0, Sdw=0,vdb =0, Sdb=0,β1=0.9,β2=0.999

on iteration t:

{

compute dw,db using current minibatch

vdw = β1vdw + (1-β1)dw

vdb = β1vdb + (1-β1)db

Sdw= β2Sdw + (1-β2)dw2

Sdb= β2Sdb + (1-β2)db2

vdwcorrected= vdw /(1-β1t)

vdbcorrected= vdb /(1-β1t)

w=w-α vdwcorrected/(√Sdw+10-8)

b=b-α vdbcorrected/(√Sdb+10-8)

}

Learning Rate decay

α = α0/(1+decayRate×epochNumber)

α = 0.95epochNumber α0

α = α0 \* k/epochNumber

manual decay, descrete decay

saddle points – happens in high dimenstions. Its better than local optimas which occur in say 2 dimentional inputs. But problem of plateu still exist.

**Tuning of hyper parameters**

dont use grid, use random – if one of the parameter is not related to cost function, then we may get only n values in a grid by n2 values in random.

Use coarse to fine : Zoom into the preferred set of values and rerun the process to get better values.

Picking of α:

if we want it in a range of 0.0001 to 1 => 10-4 to 100

r = -4\*np.random.rand()

α = 10r

it is done to get equal proportion in all th logarithmic range. i.e. 100, 10-1, 10-2, 10-3, 10-4. we need to get in which range it would be. 0.1 to 1 is considered equal to 0.0001 to 0.001.

picking of β

if we want it in a range of 0.9 to 0.9999 => 1-β => 10-1 to 10-4

r = -4/2\*(np.random.rand()+1)

β = 1-10r

Batch Normalisation

znorm=(z-μ)/σ

zbnorm = γ\*znorm+β

β, γ change just like w, with dβ, dγ

when batch normalised there is no need of b

mini batch normalisation works because it enhances regularisation(removes the over dependance on single datapoint by adding noise(since the mean and variance will be different for different batch hence noise is added)) and learning becomes faster when normalised(so when each layer is normalised)

while testing we will use weighted average of μ and σ(from each batches in each level) to calculate z norm in each level

**softmax**

activation funtion : a[L] = eZ[L]/Σ4i=i ti

L(y’,y)= -Σ4i yilog(yi’)

J(w,b)=(1/m)\* mΣi=1 L(y’(i),y(i))

**Orthogonalisation**

training set(bias) – bigger network,adam, changing hyperparameters, NN architecture

dev set(variance) – regularisation, bigger test set,changing hyperparameters, NN architecture

test set – increase dev set

real world – change dev set, cost function

early stopping,epoch no is not orthogonal – it affects both training,dev set.

**Single no evalualtion**

instead of taking true, flase-positive,false-negative consider the harmonic mean(F1 score) of flase-positive and false-negative, since they are rates(like speed)

**optimizing and satisfising value/matrix**

minimize/maximize the optizing matrix and satisfy the condition of satisfying value.

true as optimizing matrix and false-negative,false-positive as satisfying value

take dev set and test set from the same distribution

**Bayes Optimal Error** – max accuracy achievable (above human accuracy)