Shallow Neural Network

Neural Network Overview

What is a neural network?

as we saw in th last week in logistic regression model which is built out of single sigmoid unit as where you input the the feature x as well as the parameters w and b compute the linear combination Z and the apply the sigmoid function to Z and ultimately find loss between the prediction and the actual value as shown in figure (1):

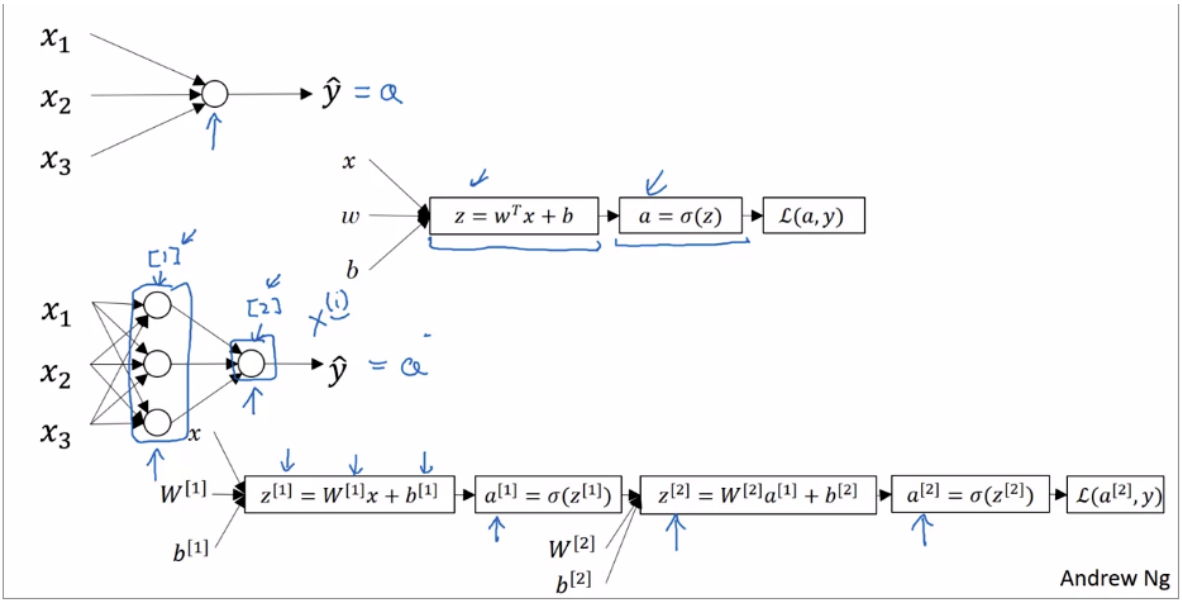


figure (1): neural network overview

on the other side, neural network is a stack set of logistic regression units as show figure 1 but there are alternative choices for activation functions to be used other than sigmoid activation.

each unit in the network is responsible for performing two steps of calculations in a single layer which are (Z and a) calculations and the output of the layer act as an input to the next layer until we reach the ultimate unit and compute the loss, such issue requires introducing new notation to indicate the layer, so that we will use squared brackets to indicate certain layer and its calculation such as W[1], b[1], z[1] and a[1] as shown in figure 1. each layer has its own parameters (W and b) and thus, back propagation is then performed with respect to each parameter in the network from the last layer to the first layer.

Neural Network Representation?

what do the part of neural network mean?

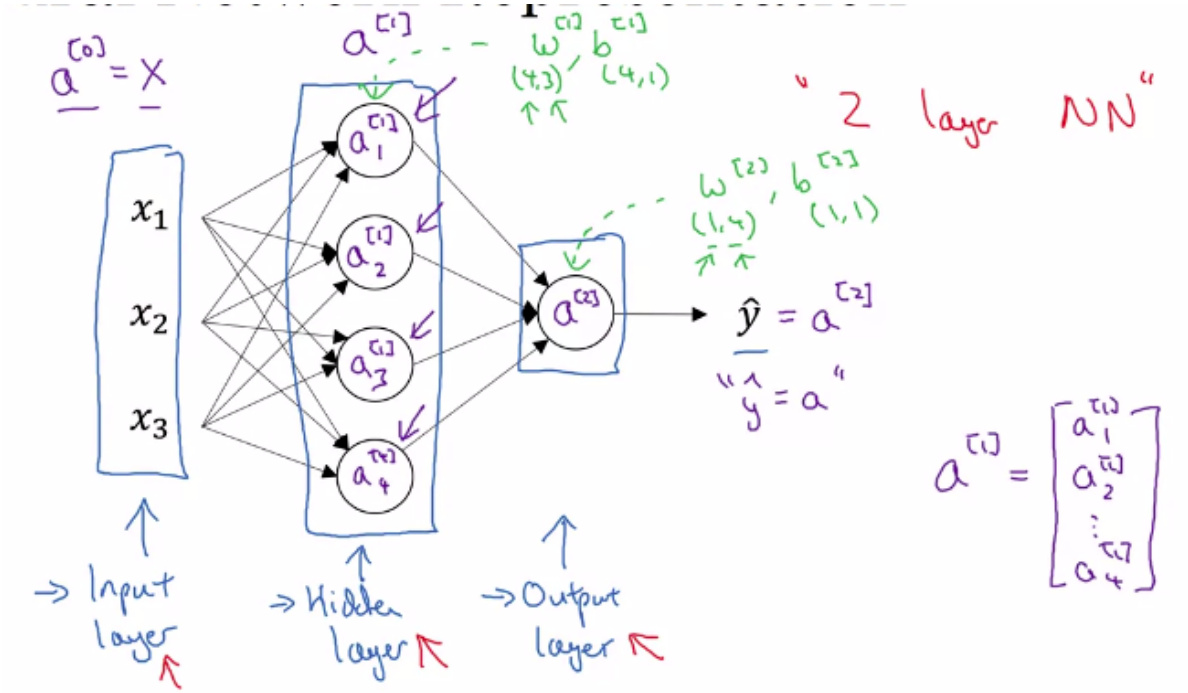


figure (2): neural network representation

figure 2 represents a simple neural network, the first part of the layer is called the **input layer** which hold the input feature values without any activation functions, the second layer is called the **hidden layer** which is responsible for linearly combining the input variables and then apply the activation to it, and the last one is called the **output layer** which is responsible for predicting the output value. to have better idea about the hidden layer and what does it mean, as you that training set contain both input features values (input layers) as well as the class label (output layer) which are explicitly known for you but in case of hidden layer its input and output is not readily available for you in the training set and the are inferred during the training phase, for this reason it is called a hidden layer.

lets introduce more notation, in logistic regression we were using vector X as input features representative, alternatively, the input feature vector will be represented by a[0] , the hidden layer will be denoted by a[1] which will generate the values a[1]*1, a[1]*2, a[1]3, a[1]4 and finally, the output layer is called a[2].

such structure in figure 2 is called **(2 layers neural network)** despite that we have defined 3 layers input hidden and output. the reason behind that is we do not count the input layer as an official layer because it does not have any activation function.

on more thing to mention is that the hidden and the output layers will have parameters (W and b) associated with each of them and the will be indicated as according to the notation of certain layer. for example the hidden layer a[1] will have the parameters w[1] and b[1] and so on. later on we will talk in details about the dimensions of each vector.

Computing a Neural Network's Output

The neural network basic computation

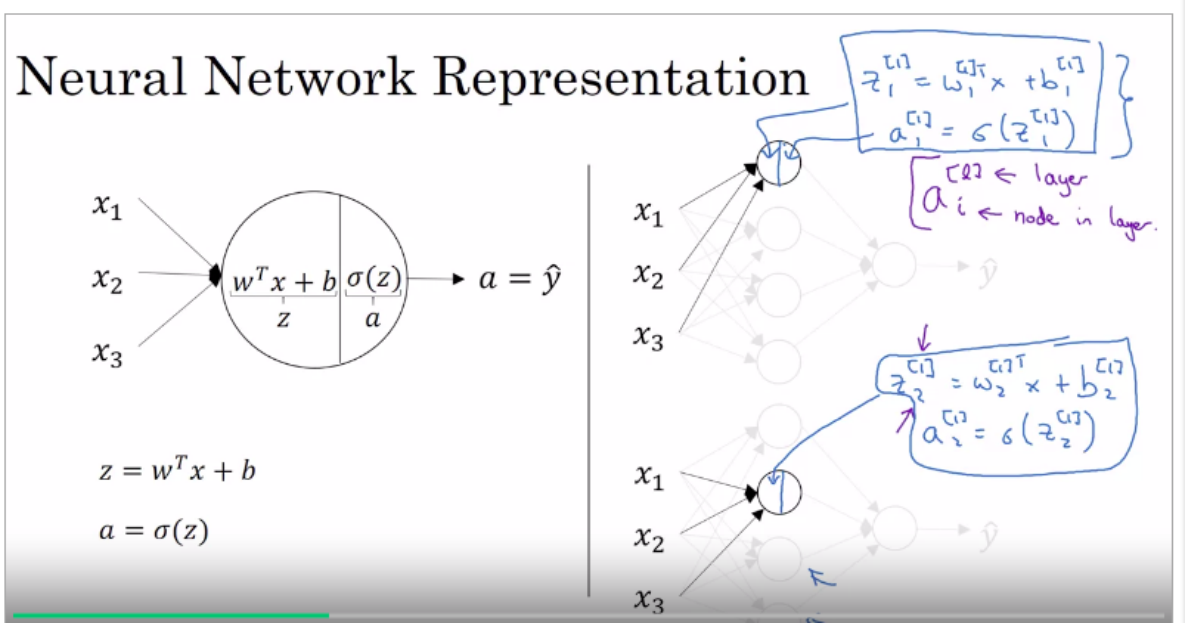


figure (3): What is going on inside the neural network??

In Logistic regression (LHS of figure 3), each single unit performs two steps of calculations including:

```
z =w^T * x +b           (1)
a = sigmoid(z)          (2)
```

The same computation is performed in the neural network but multiple times according to the number of units i the neural network, consider the RHS of figure (3) which represents a shallow neural network with three input variables, single training example, single hidden layer with four units and output layer with single unit, lets dive deeper into the calculation of each unit in the network, we well take the first unit as an example.

Similar to logistic regression, equation (1) and (2) are performed in the each unit in the network. bear in mind that the number in the squared brackets represents the layer index while the subscripted number represents the index of the unit in that layer.

```
for the first unit:
z[1]_1 = w^T[1]_1 * x + b[1]_1          (1)
a[1]_1 = sgmoid(z[1]_1)                 (2)

for the second unit:
z[1]_2 = w^T[1]_2 * x + b[1]_2          (3)
a[1]_2 = sgmoid(z[1]_2)                 (4)

for the third unit:
z[1]_3 = w^T[1]_3 * x + b[1]_3          (5)
a[1]_3 = sgmoid(z[1]_3)                 (6)

for the fourth unit:
z[1]_4 = w^T[1]_4 * x + b[1]_4          (7)
a[1]_4 = sgmoid(z[1]_4)                 (8)
```

to perform these computations, you need start a for loop to do all the computation (1-8), but using for loop is not efficient practice with neural networks, so lets vectorize these equations:

```
for the hidden layer:
vectorizing the linear equations(1, 3, 5, 7)
w^T[i]_j is a row vector of shape (1,n) or (1,3) in our example, in vectorized
version the weight vectors are stacked
together to form a matrix of shape (r[i],n) or (4,3) in our example where
(r[i]) represents the number of units in the
layers and (n) is the number of input variables, then the weights matrix is
multiplied by the input matrix which has
the shape (n,1) or (3,1) in our case, ultimately, the bias vector of shape
(r[1],1) to produce a column vector contains
the Z's values with shape of (r[1],1) or (4,1) in our case.
    (r[1],1)              (r[1],n)      (n,1)          (r[1],1)
    (4,1)                 (4,3)         (3,1)          (4,1)
    Z[1]            =     W[1]      *     x           b[1]
z[1] =  [z[1]_1,    =   [w^T[1]_1,  *   [x_1,   +   [b[1]_1,        =
[w^T[1]_1 * x + b[1]_1,
         z[1]_2,         w^T[1]_2,       x_2,        b[1]_2,
w^T[1]_2 * x + b[1]_2,
         z[1]_3,         w^T[1]_3,       x_3]        b[1]_3,
w^T[1]_3 * x + b[1]_3,
         z[1]_4]         w^T[1]_4]                   b[1]_4]
w^T[1]_4 * x + b[1]_4]
```

```
vectorizing the linear equations(2, 4, 6, 8)
applying the sigmoid function element-wisely to z[1] will produce a column
vector of shape(r,1)
        (r[1],1)
        (4,1)
a[1] = [a[1]_1,
        a[1]_2,
        a[1]_3,
        a[1]_4]

for the output layer:
the output layer has the parameter w[2] of shape (1,4) and b of shape (1,1)
(r[2],1)   (r[2],r[1])  (r[1],1) + (r[2],1)
(1,1)       (1,4)       (4,1)    + (1,1)
z[2] =      W[2]    *    a[1]    +  b[2]

(r[2],1)    (r[2],1)
(1,1)       (1,1)
a[2]    = sigmoid(z[2])
```

remember from the previous section that we indicated x as a[0] vector so that, this notation can substitute x in our equations

Vectorizing Along Multiple examples

Completely vectorized shallow neural network

in the previous section we saw how to find a[2] = y-hat for a single training example, in this section we will complete the same mission but for (m) training examples, let's see how:

```
to find predict the prediction of all training example we have to do the
following computations:
x_1 ===============> a[2]_1 = y-hat
x_2 ===============> a[2]_2 = y-hat
x_3 ===============> a[2]_3 = y-hat
x_4 ===============> a[2]_4 = y-hat
.
.
x_m ===============> a[2]_m = y-hat

which can be obtained by for loop through each example:
for i 1 to m:
    z[1]_i = w[1] * x_i +b[1]
    a[1]_i = sigmod(z[1]_i)
    z[2]_i = w[2] * a[1]_i +b[2]
    a[2]_i = sigmod(z[2]_i)

Vectorized implementation:
recall that X is a matrix of training examples stacked as columns with shape of
(n,m) as follow:
X = [x_1, x_2, ......x_m ]
to vectorize the above for loop we have to compute:

Z[1] = W[1] * X + b[1]
A[1] = sigmod(Z[1])
```

```
Z[2] = W[2] * A[1] +b[2]
A[2] = sigmod(Z[2])

substituting single training example by m training example will result in
matrix Z[1] and A[1] of shape (r[1],m) where each column represents the
computations of a
single training example. in addition Z[2] and A[2] are matrix of shape (r[2],m)
```

Explanation for Vectorized Implementation

Vectorized Implementation justified

```
for each training example you end up computing :
z[1]_1 = w^T[1] * x_1 + b[1]
z[1]_2 = w^T[1] * x_2 + b[1]
z[1]_m = w^T[1] * x_3 + b[1]

w
W[1] is a matrix of shape (r[i],r[j])
and perform
w^T[1] * x_1 = column vector
w^T[1] * x_2 = column vector
w^T[1] * x_m = column vector

X is a matrix of all training examples stacked as column for each example and
hence
W[1] * X  = will results in a matrix that contains the Z values stacked as
columnns

= [z[1]_1 z[1]_2 z[1]_m] = Z[1]
```

Activation Functions

Activation functions other than sigmoid

up to the moment, we have been using sigmoid as an activation for our neural network. sigmoid is good choice to regularize the data to be between (0 and 1), however, in some cases sigmoid may not work properly for your model so that other activation functions are required. there several activation functions other than sigmoid but those which proved to to give a good performance are:

1- Tanh

2- Rectified Liner unit (ReLU) and leaky Rectified Linear Unit (LRelu)

lets deep diver into each of them:

```
Tanh Function : the tanh function regularize its input to be between (-1 and 1)
according to
the following formula
a = tanh(z) = (e^z - e^-z)/(e^z + e^-z)
tanh function is actuall a shifted and scaled version of the sigmoid function

using tanh as an activation for the hidden unit improve the pefformance of the
network hhan
using the sigmoid function as its output lies between 1 and -1 and thus the
mean of the activation is closer to haveng
0 value which in turn make it easier for the next layer to
learn. one more thing to note that tanh is prefred t be used in the hidden
units but not the
output unit as it produce values between (-1 and 1) where you have to set the
output activation according to the class
label values in your data (sigmoid for binary class, softmax for n class)
```

in our implementations, we would refer to the activation as g[i] where g represents the activation and [i] is represents the layer it refers to .

on of the down side of both sigmoid and tanh functions if z is very large or small value, the derivative of the function comes closer to zero and thus the gradient descent becomes slower in learning. so that, the other choice is the ReLU function

```
ReLue function: regularize the deriveative between 0 and z according to the
following formula
g = max(0,z)
so the derivative is 1 so long as z is positive and zero if z is negative.
however in implemantation when z is
exactly 0 the derivative is not defined but when it is implemneted in computeer
you get 0.000000000000000000 very small, in practice you can pretend the
derivative when z iz 0 that the derivative is equal 1 or 0 and your work will
be fine. this is due to the fact ReLU is not diffrentiable at 0
on disadvantage of ReLU is that the derivative is equal to 0 when z is negative
but there is another version of ReLU is called Leaky ReLU which modify the
original function when z is negative by slighting the slope in the neagtive
side

tha advantage of bot ReLU and LReLU for most of the space of z is the
derivative of the activation is very different from 0 which in turn makes the
learning faster
```

rule of thumb for selecting activation functions:

1- if the output is (0,1) >>>>>sigmoid

2- ReLU is a good choice for the hidden layer

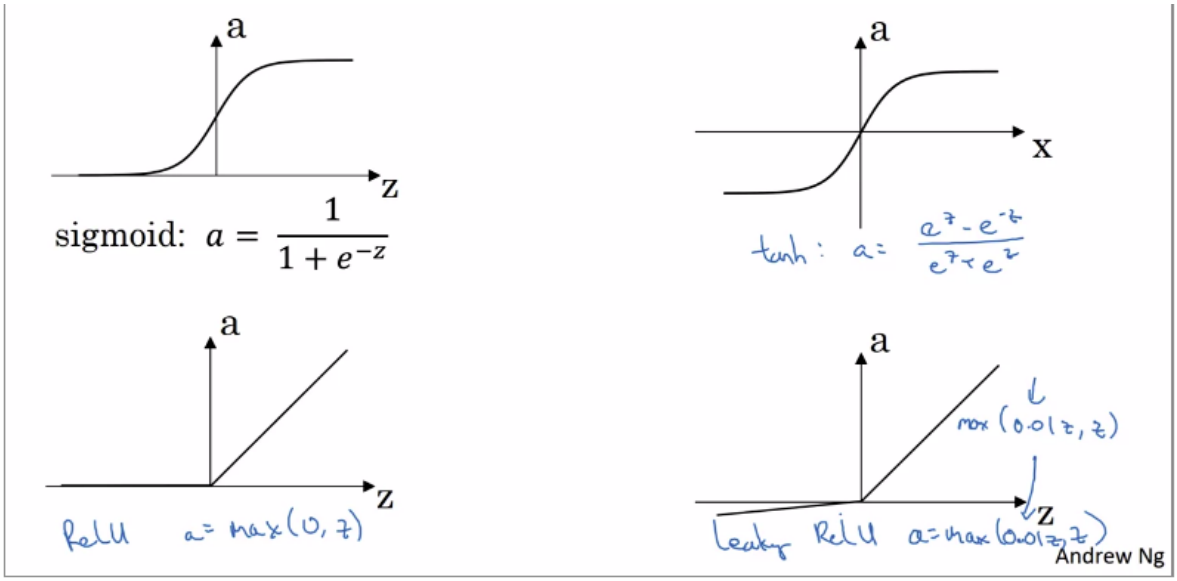


figure (4) activation functions graphical representation

Why do we need a non-linear activation functions?

Explanation

given X:

```
z[1] =W[1]^T * x + b[1]      (1)
a[1] = g[1](z[1])            (2)
z[2] =W[2]^T * a[1] + b[2]   (3)
a[2] = g[2](z[2])            (4)

suppose in function 2 we get rid of the activation functtion an set a[1] = z[1]
or alternatively
g(z) = z (linear activation function or identity activation fynction) beacuase
it outputs
the same valu of its input.
and tha same is performed for a[2]

then y-hat willend up computing a linear function, lets dive deeper:
a[1] = z[1] = w[1] * x +b[1]         (5)
a[2] = z[2] = w[2] * a[1] +b2        (6)
the 5 into 6 we end up with:
a[2] = w[2] (w[1] * x +b[1]) +b[2]
by simplification:
= (w[2] * w[1]) * x + (w[2] * b[1] +b[2])
= w~ * x + b~

and thus using linear activation function will end up uotputting a linear
function of the
input even in case of many hidden neural networks, such neural network with
linear
activation function is useless beacuase it will end up with a linear function.

there is just one place where you might use a linear activation function when
you are doing a machine learning on the regression
```

```
problem, so if y is  areal number you can use a linear activation function on
the output layer but not the hidden layer which can be any other
choice (sig, tanh, ReLU)
```

Derivative of the Activation Functions

Sigmoid activation function

```
g(z) = 1/1+e^-z
the slope = d/dz g(z)
from the previous week we know that:
d/dz g(z)  = g(z)(1-g(z)) try to check it yourself
lets check it
if z is very large say 10:
g(z) ~= 1
d/dz g(z) ~= 1*(1-1) ~= 0

if z is very small say -10:
g(z) ~= 0
d/dz g(z) ~= 0*(1-0) ~= 0

if z is 0:
g(z) = 0.5
d/dz g(z) ~= 0.5*(1-0.5) ~= 0.25
```

Tanh activation function

```
g(z) = (e^z - e^-z)/(e^z + e^-z)
the slope = d/dz g(z)
by calculas :
d/dz g(z) = 1 - (g(z))^2
follow this link for detailed tanh deriveaton:
https://blogs.cuit.columbia.edu/zp2130/derivative_of_tanh_function/
lets check it

if z is very large say 10:
g(z) ~= 1
d/dz g(z) ~= 1 - (1)^2 ~= 0

if z is very small say -10:
g(z) ~= -1
d/dz g(z) ~= 1- (-1)^2  ~= 0

if z is 0:
g(z) = 0
d/dz g(z) ~=  1 - 0 ~= 1
```

ReLU and LReLU

```
ReLU
g(z) = max(0,z)
the slope = d/dz g(z)
by calculas :
0 if z < 0
1 if z > 0
undifined if z = 0

-----------------------------------------
LReLU
g(z) = max(0.01z,z)
the slope = d/dz g(z)
by calculas :
0.01 if z < 0
1 if z > 0
undifined if z = 0
```

Gradient Descent for Neural Networks

Gradient descent implementation

```
for shallow neural network we have the folloing:
r[0] = the number of input features = nx
r[1] = the number of hidden units
r[2] = the number of output unit = 1
w[1]: the hidden layer weight matrix which is of shape (r[1],r[0])
b[1]: the hidden layer bias vector which is of shape (r[1],1)
w[2]: the output layer weight matrix which is of shape (r[2],r[1])
b[2]: the output layer bias vector which is of shape (r[2],1)

the cost function is:
J(w[1],b[1],w[2],b[2]) =(-1/m) * sum_i_to_m(L(y-hat,y))

to implement the gradient descent algorithm:
1-initialize the weight matrices randomly
repeat:
2- compute (y-hat^(i)) for all training examples
3- compyte the derivatives:
dw[1] = dj/dw[1]
db[1] = dj/db[1]
dw[2] = dj/dw[2]
db[2] = dj/db[2]
4- udate the weights:
w[1] = w[1] - alpha * dw[1]
b[1] = b[1] - alpha * db[1]
w[2] = w[2] - alpha * dw[2]
b[2] = b[2] - alpha * db[2]

to implement the gradient descent algorithm we need the following equations but
first lets
remember the forward propagation equations:
```

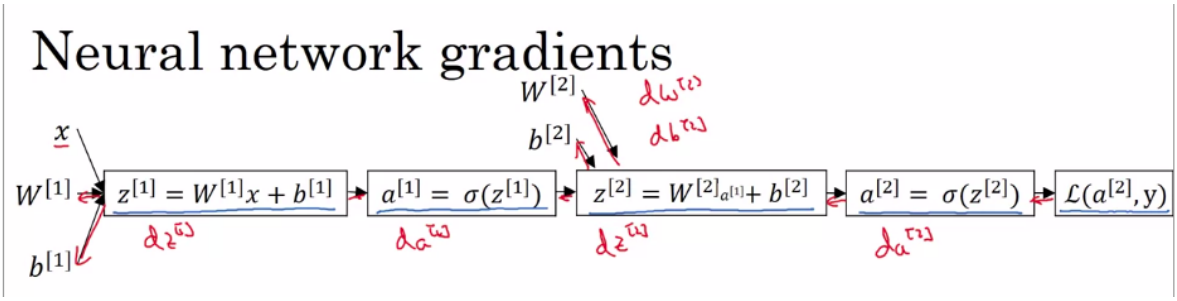
```
forward propagation:
Z[1] =W[1]^T * x + b[1]          (1)
A[1] = g[1](Z[1])                (2)
Z[2] =W[2] * A[1]^T + b[2]       (3)
A[2] = g[2](Z[2])                (4)

dZ[2] = A[2] -Y                  (5)
dw[2] = (1/m) * A[1] * dz[2]^T   (6)
db[2] = (1/m) * np.sum(dz[2],axis =1, keepdims = True)   (7)

dZ[1] = (w[2]^T * dZ[2]) x g[1](z[1])           (8)
        (r[1],m)*           (r[1],m)
(x) her stand for element-wise product
dw[1] = (1/m) * X^T * dz[1] (9)
db[1] = (1/m) * np.sum(dz[1],axis =1, keepdims = True)   (10)
```

Back-propagation Intuition

Neural network gradients



Figure(5): Shallow neural network computational graph

to derive th back propagation equations, as shown in figure 5 we have to derive back through the neural network with respect w[2], b[2], w[1] and b[1]. the process is quite similar to what we have already done in logistic regression but this time in two step rather than one step as we have 2 layers, lets go through:

```
in order to update the weights of our neural network we have to find:
for the output layer: dl/da[2], dl/dz[2] dl/dw[2], dl/db[2]
remmber that from logistic regression :
dl/dw[2] = (dl/da[2]) * (da[2]/dz[2]) * (dz[2]/dw[2])
dl/db[2] = (dl/da[2]) * (da[2]/dz[2]) * (dz[2]/db[2])
and
dl/da[2] = ((a[2]-y)/a[2](1-a[2]))
da[2]/dz[2] = a[2](1-a[2])
and
dl/dz[2] = (dl/da[2]) * (da[2]/dz[2])
         = ((a[2]-y)/a[2](1-a[2])) * a[2](1-a[2]) = (a[2] -y)
dl/dw[2] = (a[2] -y) * a[1]^T = dz[2] * a[2]^T
dl/db[2] =dz[2]
all these derivatives are the same of what we have performed in logistic
regression

for the hidden layer: dl/da[1], dl/dz[1] dl/dw[1], dl/db[1]
dl/da[1] = (dl/da[2]) * (da[2]/dz[2]) * dz[2]/da[1]
```

```
dz[2]/da[1] = w[2]
dl/da[1] = ((a[2]-y)/a[2](1-a[2])) * a[2](1-a[2])  * w[2]
         = (a[2] - y) * w[2] = dz[2]* w[2]^T
dl/dz[1] = (dl/da[2]) * (da[2]/dz[2]) * dz[2]/da[1] * da[1]/dz[1]
         =  dz[2]* w[2]^T x g[1]~(z[1])
g[1](z[1]) replaces a[1] which can be any activation function you select, so it
derivative
is g~
ultimately:
dw[1] = dz[1] * x^T
db[1] = dz[1]
```

up to the moment, all these computations are considered for a single training example, let's have a look how to vectorize them for m training example by considering that all calculations we perform are in matrix form:

```
dZ[2] = A[2] - Y
dW[2] = (1/m) * np.dot(dZ[2], A[1]^T)
db[2] = (1/m) * np.sum(dZ[2], axis = 1. keepdims= True)

dZ[1] = W[2]^T * dZ[2] x g~[1](Z[1])
dW[1] = (1/m) * dZ[1] * X^T
db[1] = (1/m) * np.sum(dZ[1], axis = 1. keepdims= True)
```

Random Initialization

What happens if you initialize weights to zero?

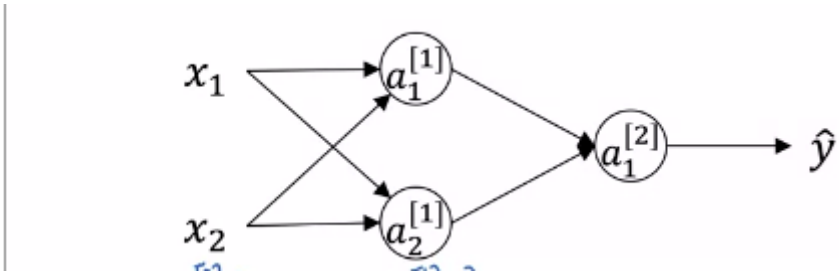


Figure (6): simple neural network

assume we have a simple neural network as shown in figure (6) with two input features and two units in the hidden layer and single unit in the output unit, let's see what happens if we initialize the weights to zeros :

```
we have:
r[0] =2 , r[1] = 2 , r[2] = 1
if we initilize with 0 we will have:
w[1] = [0 0
        0 0]
b[1] = [0
        0]
w[2] = [0 0]
```

```
for any ecample you feed int the neural network, you will have:
a[1]_1 = a[1]_2 becuase both units are computing the same values

when computing the packpropagation
dz[1]_1 = dz[1]_2
and
dw[1] = [u v
         u v]
when you update the weights
w[1] = w[1] alpha * dw[1]
then you will have a symetric matrix where the first row = the second row
and thus after many iteration wil still compute the same function as we have
just a single unit in the hidden layer
```

Random initialization

```
to overcome such issue, wee need to randomly initialize the weights
W[1] = np.random.randn((r[i],r[j])) * 0.01
moltiplication by 0.01 is just to have smaller random values as larger values
may affect the learning process and make it a bit smaller
b[1] = np.zeros(r[i],1)
there is no problem to initilaize b[1] with zeros as long as w[1] is
initialized randomly
w[2] = np.random.randn((r[i],r[j])) * 0.01
b[1] = np.zeros(r[j],1)
```