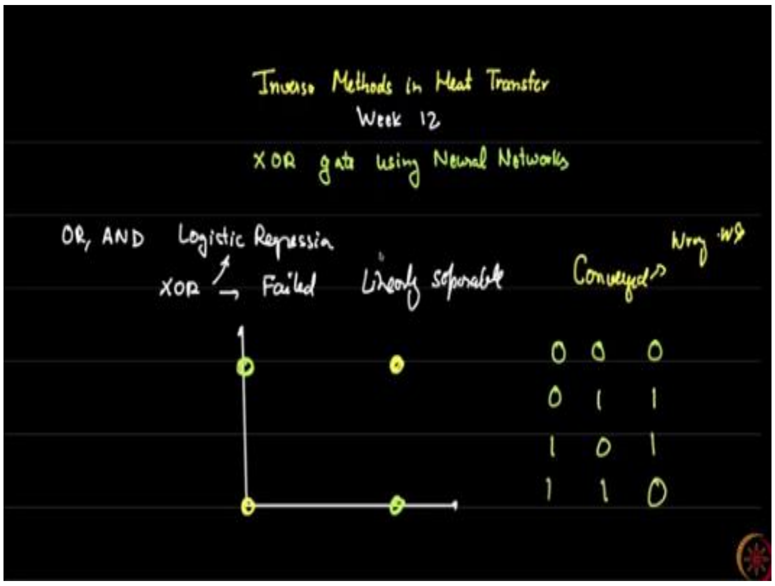
Inverse Methods in Heat Transfer

Prof. Balaji Srinivasan
Department of Mechanical Engineering
Indian Institute of Technology-Madras

Lecture - 63
Code Example of Shallow Neural Network - XOR Gate

Welcome back. This is week 12 of inverse methods in heat transfer.

(Refer Slide Time: 00:29)

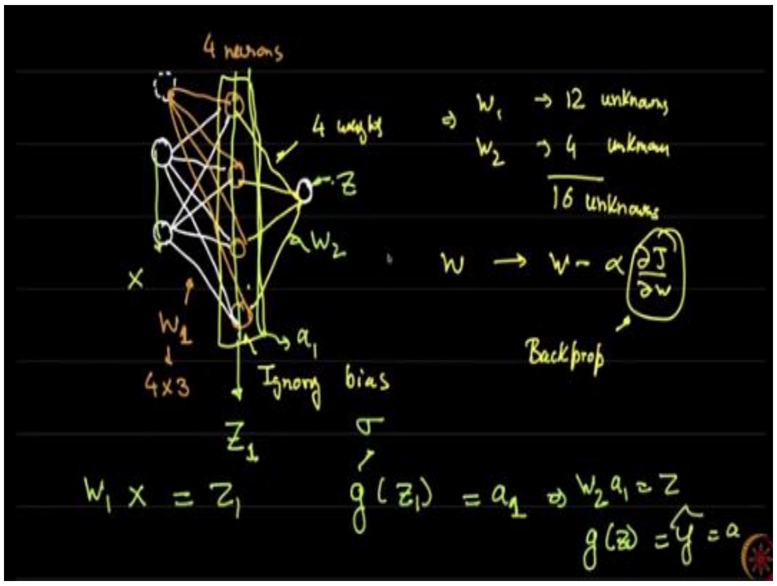

Inverse Methods in Heat Transfer
Week 12
XOR gate using Neural Networks
OR, AND Logistic Regression
XOR → Failed
Linearly separable
Converged?
Wrong W's
0 0 0
0 1 1
1 0 1
1 1 0

In the last video you would have seen that we had done OR gate and AND gate using logistic regression. We also tried to do XOR gate using logistic regression. And of course, we saw that it failed, because this is not linearly separable. So, because it was not linearly separable, we could no longer do XOR gate.

But what really happened when we tried to do XOR gate using logistic regression, especially gradient descent is that the answer converged but the wrong peaks, okay? So, it is not as if it did not converge gradient descent, it converged, but it converged to the wrong weights. Now recollect that XOR gate is one type of data for this diagonal and another type of data for this diagonal.

So, we all know that it is 0 0 gives you 0. 0 1 gives you 1.1 0 gives you 1, but 1 1 gives you back 0. So, this data is obviously not linearly separable.

(Refer Slide Time: 01:47)



Now that is why we introduced some intermediate layers here, okay? So, we need at least one single hidden layer. So, we can introduce this hidden layer, let us put four neurons here in the hidden layer, just arbitrarily, okay? So, we put four hidden neurons arbitrarily. And we can have one output neuron in this extra layer.

So now if we create a fully connected network this way, we of course need a bias unit and we can put this in a separate color. So let us call all these initial weights as w_1. So, w_1 as size, as you can see, depending on how you wish to write it as 4 cross 3 or 3 cross 4. Next, we have these four. So, for now I am ignoring bias here. This is what you will see in the code that I am showing here.

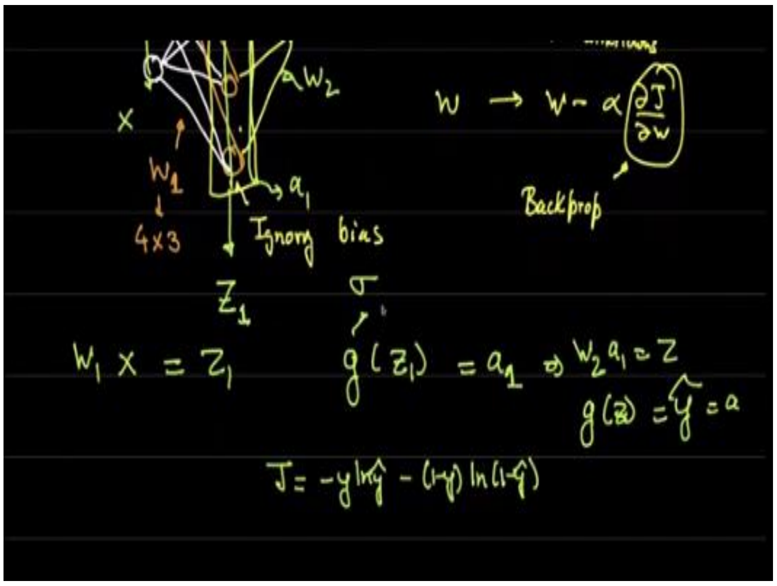
Why four, we can use any number of neurons if we want. I have just chosen four just for convenience. But you can use any number of neurons because it is really speaking just some arbitrary interpretation of what is the intermediate calculation. So, this of course, is four weights. This gives you w_1 as 12 unknowns and w_2 as another 4 unknowns.

So, we have a total of 16 unknowns here, all of which will be updated as $w = w\text{-}\alpha \frac{\partial J}{\partial w}$. This of course is what requires backprop. Now recollect how we did the forward propagation through this network. The forward propagation was supposed to be called, this layer as z_1 that is the linear activation here is z_1.

And this was x, so then x multiplied by w_1 gives usz_1. So that is a linear transformation gives us z_1. Then you take g (z_1), which was typically we can take a sigmoid here. So sigmoid of z_1 gives us the nonlinear activation. Now the outputs here are a_1, then you do w_2, which is the set of weights, times a_1 gives you z, which is the output here.

And then of course, sigmoid of z gives you $\hat{y}$, which you can call a if you wish as the output.

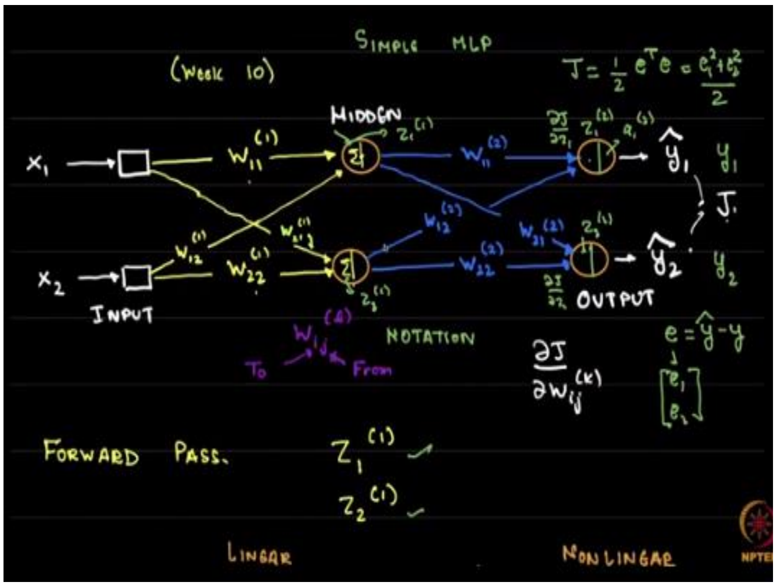
(Refer Slide Time: 04:43)



W_2
x
W_1
4×3
a_1
Ignore bias
Z_1
σ
$W_1 x = Z_1$
$g(Z_1) = a_1 \Rightarrow W_2 a_1 = Z$
$g(z) = \hat{y} = a$
$J = -y \ln \hat{y} - (1-y) \ln(1-\hat{y})$
$w \rightarrow w - \alpha \frac{\partial J}{\partial w}$
Backprop

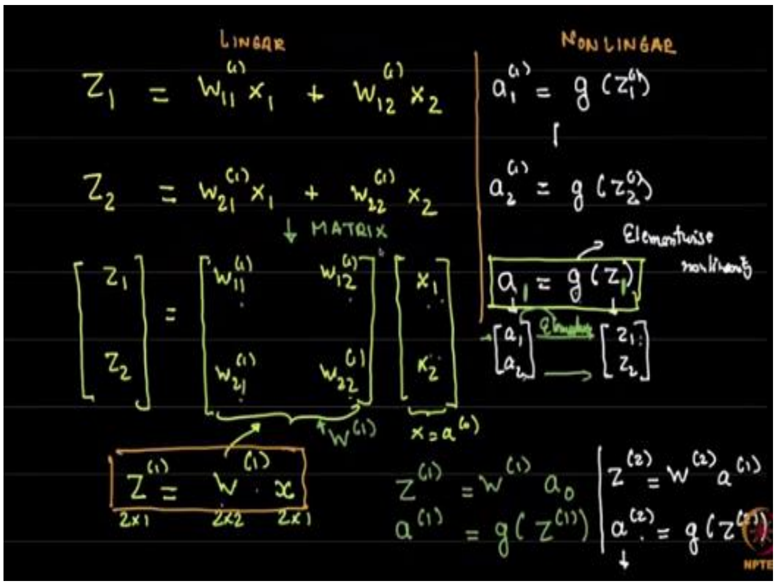
Remember that the loss function here was the binary cross entropy loss function $-y ln \hat{y} - (1-y) ln(1-\hat{y})$. The important thing was how we do backprop in order to determine these weights.

(Refer Slide Time: 05:00)



Now I have just copied and pasted exactly what I had shown you on page 10. So, remember, this is all from week 10 of our notes.

(Refer Slide Time: 05:12)



LINEAR | NONLINEAR

$Z_1 = W_{11}^{(1)} x_1 + W_{12}^{(1)} x_2$ | $a_1^{(1)} = g(z_1^{(1)})$

$Z_2 = W_{21}^{(1)} x_1 + W_{22}^{(1)} x_2$ | $a_2^{(1)} = g(z_2^{(1)})$

↓ MATRIX

$$\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \underbrace{\begin{bmatrix} W_{11}^{(1)} & W_{12}^{(1)} \\ W_{21}^{(1)} & W_{22}^{(1)} \end{bmatrix}}_{W^{(1)}} \underbrace{\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}}_{x = a^{(0)}}$$

Elementwise nonlinearity

$\boxed{a_1 = g(z_1)}$

$\begin{bmatrix} a_1 \\ a_2 \end{bmatrix}$ Elementwise $\begin{bmatrix} z_1 \\ z_2 \end{bmatrix}$

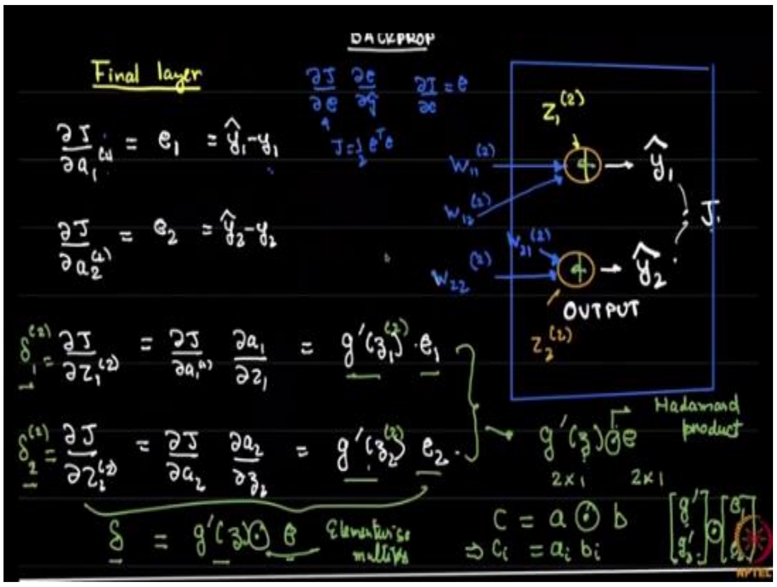
$$\boxed{\underset{2\times1}{Z^{(1)}} = \underset{2\times2}{W^{(1)}} \cdot \underset{2\times1}{x}}$$

$z^{(1)} = W^{(1)} a_0$ | $z^{(2)} = W^{(2)} a^{(1)}$

$a^{(1)} = g(z^{(1)})$ | $a^{(2)} = g(z^{(2)})$

So, I have written the same thing here. You do a forward prop as z 1 s w 1 times x, and after that a_1 is g (z_1). And similarly, z_2 is w_2 times a_1, and a_2 is g times g (z_2) or sigmoid of z_2.

(Refer Slide Time: 05:30)



BACKPROP

Final layer

$\frac{\partial J}{\partial e} \frac{\partial e}{\partial \hat{y}}$ $\quad \frac{\partial J}{\partial e} = e$ $\quad J = \frac{1}{2} e^T e$

$\frac{\partial J}{\partial a_1^{(2)}} = e_1 = \hat{y}_1 - y_1$

$\frac{\partial J}{\partial a_2^{(2)}} = e_2 = \hat{y}_2 - y_2$

$\delta_1^{(2)} = \frac{\partial J}{\partial z_1^{(2)}} = \frac{\partial J}{\partial a_1^{(2)}} \frac{\partial a_1}{\partial z_1} = g'(z_1^{(2)}) \cdot e_1$

$\delta_2^{(2)} = \frac{\partial J}{\partial z_2^{(2)}} = \frac{\partial J}{\partial a_2} \frac{\partial a_2}{\partial z_2} = g'(z_2^{(2)}) \, e_2$

$\underset{2\times1}{g'(z)} \odot \underset{2\times1}{e}$ Hadamard product

$\delta = g'(z) \odot e$ Elementwise multiply

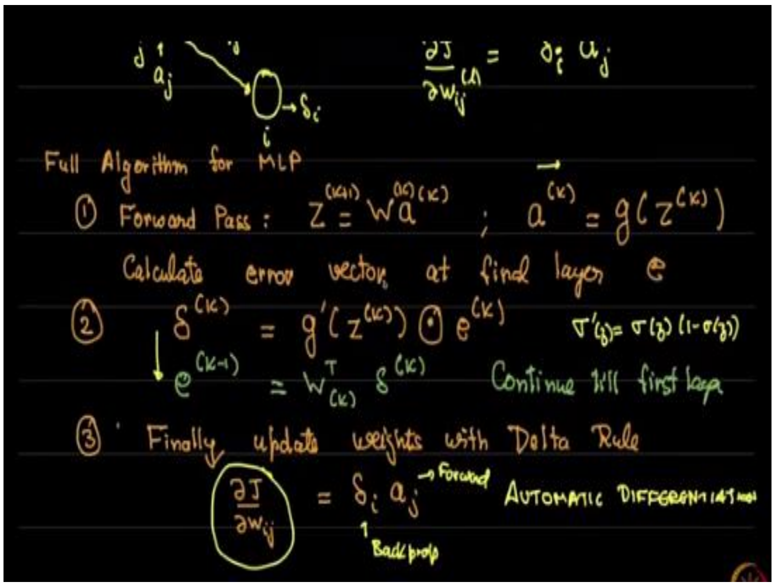
$c = a \odot b$

$\Rightarrow c_i = a_i b_i$

$\begin{bmatrix} g'_1 \\ g'_2 \end{bmatrix} \odot \begin{bmatrix} e_1 \\ e_2 \end{bmatrix}$

The backprop here, I had shown you in quite some detail in the previous week's videos. The important thing is the final summation that I had here.

(Refer Slide Time: 05:41)


j a_j δ_i i $\frac{\partial J}{\partial w_{ij}^{(l)}} = \delta_i a_j$

Full Algorithm for MLP

① Forward Pass : $z^{(k+1)} = W^{(k)} \vec{a}^{(k)}$; $a^{(k)} = g(z^{(k)})$

Calculate error vector at final layer e

② $\delta^{(k)} = g'(z^{(k)}) \odot e^{(k)}$ $\sigma'(z) = \sigma(z)(1-\sigma(z))$

$e^{(k-1)} = W_{(k)}^{T} \delta^{(k)}$ Continue till first layer

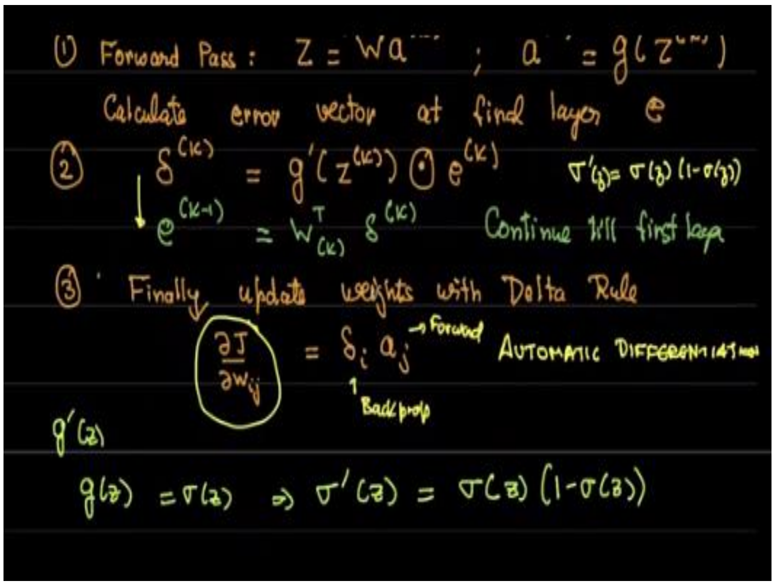
③ Finally update weights with Delta Rule

$$\boxed{\frac{\partial J}{\partial w_{ij}}} = \underset{\text{Backprop}}{\delta_i} \overset{\text{Forward}}{a_j}$$ AUTOMATIC DIFFERENTIATION

That the forward prop versus z is w times a in all sequential layers. So, z here will be w times this. Similarly, z here will be w times this, but there is also a nonlinear step in between g (z) gives you a. And similarly, g of this z will give you the a. Of course, I showed a different example, we only have one output neuron here.

Finally, when we want $\frac{\partial J}{\partial w_{ij}}$, we have to take error of the output, error of the z of the output multiplied by the activation of the input, okay? Remember this entire dance of taking e is w transpose delta, when z is w times the same a. So, you will see that e k - 1 is w transpose times delta k. And this basically shifts around. I had explained this in week 10. So, notice this w transpose wherever there is w. And finally notice that delta is g prime times e.

(Refer Slide Time: 06:54)



① Forward Pass : $Z = Wa^{\cdots}$; $a^{\cdot} = g(z^{(\cdots)})$

Calculate error vector at final layer e

② $\delta^{(k)} = g'(z^{(k)}) \odot e^{(k)}$ $\quad \sigma'(z) = \sigma(z)(1-\sigma(z))$

$e^{(k-1)} = W^T_{(k)} \, \delta^{(k)}$ Continue till first layer

③ Finally update weights with Delta Rule

$\boxed{\frac{\partial J}{\partial w_{ij}}} = \delta_i \, a_j$ → Forward ↑ Backprop AUTOMATIC DIFFERENTIATION

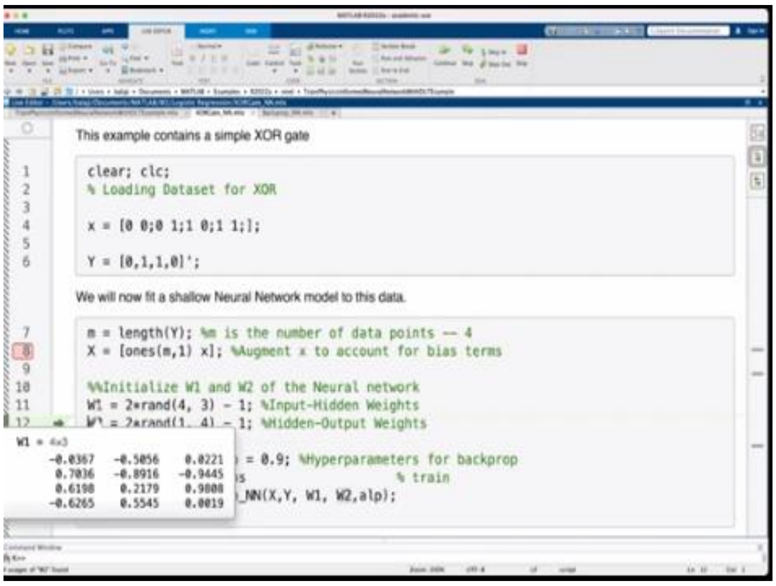
$g'(z)$

$g(z) = \sigma(z) \Rightarrow \sigma'(z) = \sigma(z)(1-\sigma(z))$

Recall that when I look at g prime of z, for g (z) being sigmoid means that sigmoid of z is basically, sigmoid prime of z is sigmoid of z minus 1 minus sigmoid of z. Once again, I had discussed this in our earlier videos. So, what we are going to do is do a code walkthrough of exactly this network that I am showing here, which is two neurons in the input, then four neurons in the middle layer and one neuron in the output.

So, I will show a hard coded case of XOR gate and we can see whether it fares better than the logistic regression code. So let us go ahead and take a look at the code.

(Refer Slide Time: 07:38)



This example contains a simple XOR gate

```
clear; clc;
% Loading Dataset for XOR

x = [0 0;0 1;1 0;1 1;];

Y = [0,1,1,0]';
```

We will now fit a shallow Neural Network model to this data.

```
m = length(Y); %m is the number of data points -- 4
X = [ones(m,1) x]; %Augment x to account for bias terms

%%Initialize W1 and W2 of the Neural network
W1 = 2*rand(4, 3) - 1; %Input-Hidden Weights
W2 = 2*rand(1, 4) - 1; %Hidden-Output Weights

) = 0.9; %Hyperparameters for backprop
s                         % train
_NN(X,Y, W1, W2,alp);
```

```
W1 = 4x3
   -0.0367   -0.5056    0.0221
    0.7036   -0.8916   -0.9445
    0.6198    0.2179    0.9808
   -0.6265    0.5545    0.0019
```

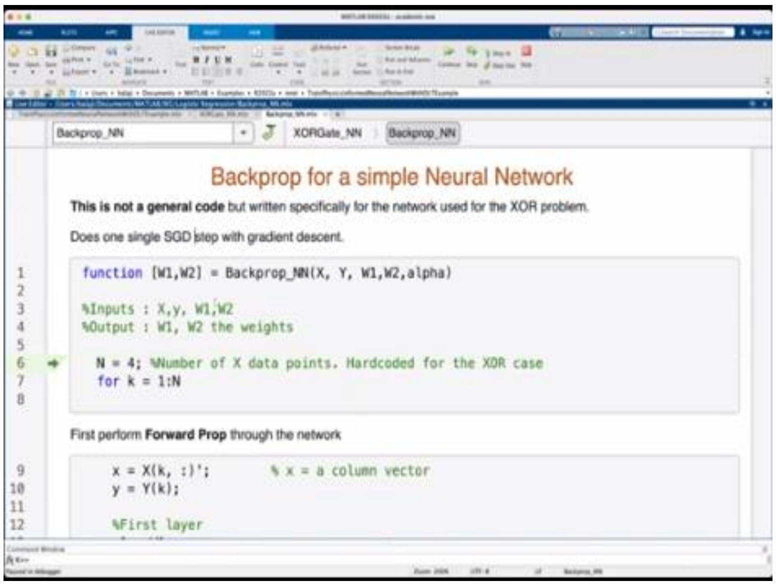
So here is a simple neural network. This is once again the XOR gate. But the change here is I have sort of hard coded this for the XOR gate, the logistic regression code was written far more elegantly. This one is written a little bit more inelegantly, because of these intermediate layers. I have written this in some detail. You should be able to run this and look through this and see that it makes sense for you.

I would highly encourage you for you to those of you who have taken the course for credit, you will have access to the code through NPTEL/Swayam. So here is simple data 0 0, 0 1, 1 0 same as before, okay, I have written, this looks like the OR gate right now. So, this is here, let us change it to the XOR gate.

So, once we run this, since we have included the bias units at the beginning, we can now look at x same as before, except now your Ground Truth is 0 1 1 0. Now what this function does, is it gives you a random number between 0 and 1. So I am going to just step and now you can see w1 is a 4 cross 3 matrix, sorry -1 and 1. It is not 0 and 1. You get -1 and 1.

So, you create a 4 cross 3 random matrix. You can see those standard PINNs, some random values are given initially. Again w2 the sizes we looked at, in just at the beginning of this. So, this is just some poor random numbers total of 16 unknowns, okay? Now I have chosen 10,000 epochs a large alpha. But the key portion of the code here is the backprop code.

(Refer Slide Time: 09:37)



Backprop for a simple Neural Network

This is not a general code but written specifically for the network used for the XOR problem.

Does one single SGD step with gradient descent.

```
function [W1,W2] = Backprop_NN(X, Y, W1,W2,alpha)

%Inputs : X,y, W1,W2
%Output : W1, W2 the weights

  N = 4; %Number of X data points. Hardcoded for the XOR case
  for k = 1:N

```

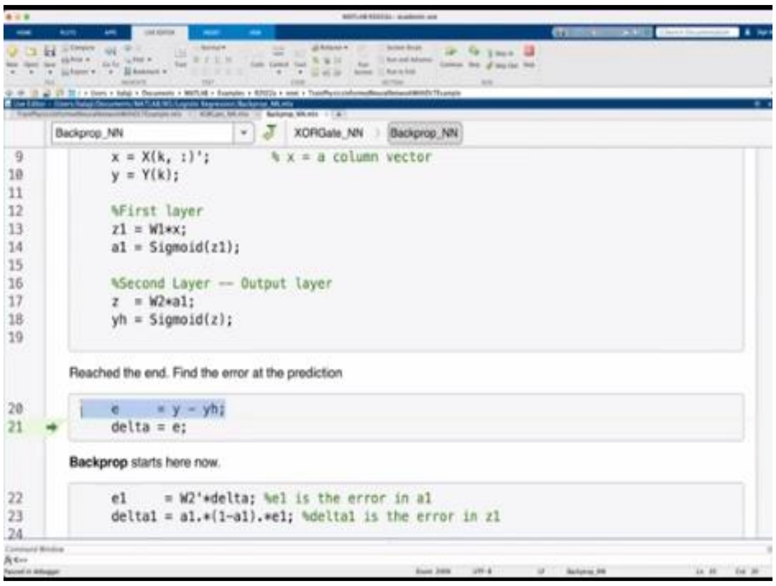
First perform **Forward Prop** through the network

```
    x = X(k, :)';        % x = a column vector
    y = Y(k);

    %First layer
```

So, what we will do is step in slowly into the backprop code. Now notice this is not a general code. This code was written specifically for the network written used for the XOR problem. I simply do in the backprop code I am simply doing just one single stochastic gradient descent with a stochastic gradient step with the backprop algorithm sitting there. Now notice this N equal to 4, this is the number of x data points.

Obviously, this is hard coded for the XOR gate. If you want some other case, then you may have to change this code. What I am doing now is sequentially stepping through every single data point. The first data point we will step through is the 100 data point, which simply says that the bias unit is 1, but x1 and x2 are simply 0. Next data point will be 1 0 1, which is bias when it is as usual 1. Then you are doing 0 1, then 1 0, then 1 1, okay. So, we run through those four. For each 1, let us see what happens.

(Refer Slide Time: 10:38)



```
x = X(k, :)';        % x = a column vector
y = Y(k);

%First layer
z1 = W1*x;
a1 = Sigmoid(z1);

%Second Layer -- Output layer
z  = W2*a1;
yh = Sigmoid(z);
```

Reached the end. Find the error at the prediction

```
e     = y - yh;
delta = e;
```

Backprop starts here now.

```
e1     = W2'*delta; %e1 is the error in a1
delta1 = a1.*(1-a1).*e1; %delta1 is the error in z1
```

Okay, so I now select the data from that particular row. So, what you should see for x now is 1 0 0. So, this is the data set. And of course, the Ground Truth here is if you have 0 0, the Ground Truth here is 0. Now notice the first layer. First layer is whatever arbitrary value of w1 we have chosen; it will just propagate that and see how well that works. Okay, so you run through that, you see z 1.

Now why are there four values here, because there are four intermediate neurons. So, for this one single data point, we now have all four neurons being updated. So, -0.03, 0.70, 0.61 and -0.62. What does these mean? Nothing. This is simply what the intermediate calculations are. We can now calculate the nonlinearities at the specific neurons.

We have simply taken sigmoid of the corresponding neurons and we are getting 0.49, 0.66, 0.65 and 0.34. Now this of course is just the end of the first layer. What we want to do is what happens in the second layer, which is the output layer. So, the output layer is now you take w2 and multiply that by a1.

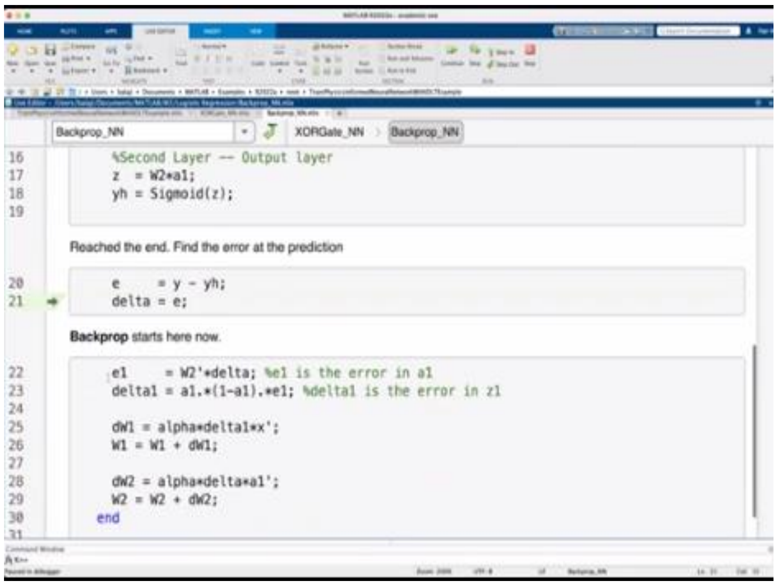
Remember, a1 is the output of the hidden layer. I would request you to maybe draw the figure by hand and just check this out. I am not able to show both the figure as well as the code simultaneously. But you can draw a figure just so that you can understand a little bit better. All these four neurons in the intermediate layer are now three multiplied by w2, and you should get the output of the last layer.

So, you should get just a single neuron. Let us see if the sizes match. W2 is 1 cross 4, a1 is 4 cross 1. So, 1 cross 4 multiplied by 4 cross 1 gives you a single 1 cross 1 and that is what you should get at this point. So, z is -0.3855 and y hat is 0.4048. So, once you see that, you see that y here was 0, but y hat your prediction is 0.4048. So here is the error. The error is what you calculate and say okay error is y minus y hat.

What happened to J our cost function? Remember that the cost function is not what we back propagate. We back propagate $\frac{\partial \mathrm{J}}{\partial \mathrm{w}}$, okay. So therefore, remember $\frac{\partial \mathrm{J}}{\partial \mathrm{w}}$ for a sigmoid or $\frac{\partial \mathrm{J}}{\partial \mathrm{z}}$ for a sigmoid with binary cross entropy simply gives you $y - \hat{y}$, okay? So that is why we typically never really have the cost function written here.

You can draw the cost function, you can plot the cost function as I will show you one in the pin case, later on. But all we are bothered about is simply this value e and then we start our backprop.

(Refer Slide Time: 13:36)



Backprop_NN

XORGate_NN › Backprop_NN

```
    %Second Layer -- Output layer
    z  = W2*a1;
    yh = Sigmoid(z);

```

Reached the end. Find the error at the prediction

```
    e     = y - yh;
    delta = e;
```

Backprop starts here now.

```
    e1     = W2'*delta; %e1 is the error in a1
    delta1 = a1.*(1-a1).*e1; %delta1 is the error in z1

    dW1 = alpha*delta1*x';
    W1 = W1 + dW1;

    dW2 = alpha*delta*a1';
    W2 = W2 + dW2;
  end
```

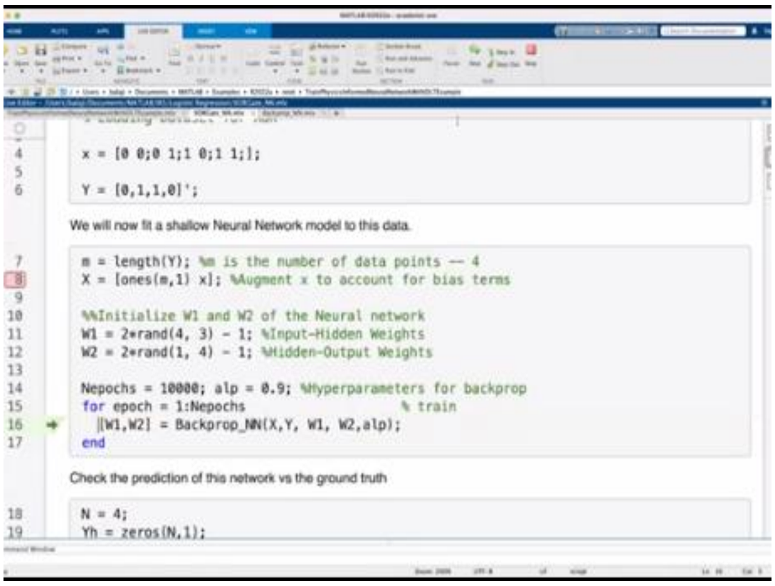
So, remember backprop starts as e goes as w transpose or w2 transpose delta. This corresponds to the backprop corresponding to this equation. This here is forward. Z is w 2 times a 1. So, the delta a 1 basically becomes w 2 transpose multiplied by delta z. So that is what you see here as e 1 is w 2 transpose delta. And then of course, we start the next step, the error in a becomes this term is just g prime.

Remember a1 is sigmoid of z. So, this is sigmoid multiplied by 1 minus sigmoid multiplied by e1, okay? Now here is the update. I have multiplied by alpha also and just run the full update, but basically the $\frac{\partial J}{\partial w}$ term is this, which is error in the output multiplied by the input. Similarly, error in the output multiplied by input, and you multiply that by alpha for learning rate and you simply update.

If this should be, there is a small trick here, this should be y hat minus y. But there is a negative here which gets cancelled and this looks like w1 is w1 plus delta w1, okay? So, we can go and do these steps. So let us take each step. So, you can see e 1 now is a 4 cross 1 matrix, because that is the error in z1 and this was error in sorry z1 is delta 1 and e1 is the error in a 1, okay.

So, you can now see d w1 is a 4 cross 3 matrix. And we are updating 4 cross 3 matrix. And we just move forward. So, you can also see what delta w2 is. It is the same size as w2 and we can update it.

(Refer Slide Time: 15:28)



```
x = [0 0;0 1;1 0;1 1;];

Y = [0,1,1,0]';
```

We will now fit a shallow Neural Network model to this data.

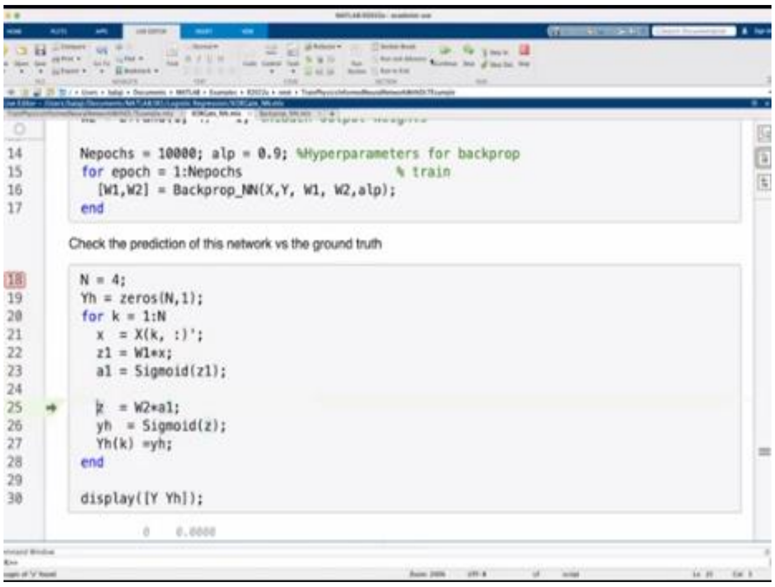
```
m = length(Y); %m is the number of data points -- 4
X = [ones(m,1) x]; %Augment x to account for bias terms

%%Initialize W1 and W2 of the Neural network
W1 = 2*rand(4, 3) - 1; %Input-Hidden Weights
W2 = 2*rand(1, 4) - 1; %Hidden-Output Weights

Nepochs = 10000; alp = 0.9; %Hyperparameters for backprop
for epoch = 1:Nepochs                          % train
  [W1,W2] = Backprop_NN(X,Y, W1, W2,alp);
end
```

Check the prediction of this network vs the ground truth

```
N = 4;
Yh = zeros(N,1);
```

So, if we come out, this is one step of backprop. Of course, if we continue, you do 10,000 steps of backprop, you get an entirely new set of w's and entirely new set of w 2's. Remember original set of w's were between -1 and 1. And so were w2. You can see that clearly these values have been updated. They have been updated constantly by feedback to ensure that you are getting close to the actual y.

(Refer Slide Time: 15:59)



```
Nepochs = 10000; alp = 0.9; %Hyperparameters for backprop
for epoch = 1:Nepochs                          % train
  [W1,W2] = Backprop_NN(X,Y, W1, W2,alp);
end
```

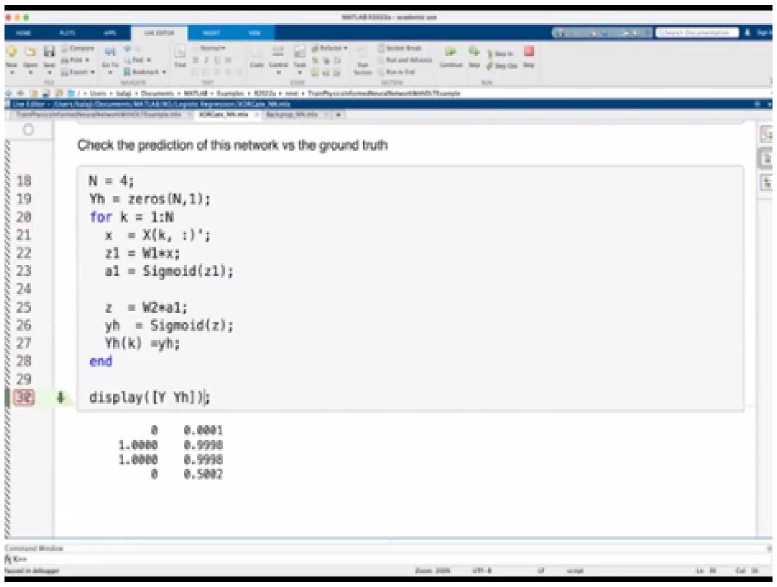
Check the prediction of this network vs the ground truth

```
N = 4;
Yh = zeros(N,1);
for k = 1:N
  x  = X(k, :)';
  z1 = W1*x;
  a1 = Sigmoid(z1);

  z  = W2*a1;
  yh  = Sigmoid(z);
  Yh(k) =yh;
end

display([Y Yh]);
```

```
      0    0.0000
```

So, the final w's we can now run them and see how well they do against the actual data. So, for each one of them, so for example, I take 1 0 0, the original z, so z for this arbitrary w looks like some random values here. You look at a, that also looks as some random values here. You calculate z, this is really negative, which luckily gives us y hat, which is really small. So, y was 0 there? Sorry, why I have it here. y was 0 there but y hat is also 0.

(Refer Slide Time: 16:45)



Check the prediction of this network vs the ground truth

```
N = 4;
Yh = zeros(N,1);
for k = 1:N
  x  = X(k, :)';
  z1 = W1*x;
  a1 = Sigmoid(z1);

  z  = W2*a1;
  yh  = Sigmoid(z);
  Yh(k) =yh;
end

display([Y Yh]);
```

```
         0    0.0001
    1.0000    0.9998
    1.0000    0.9998
         0    0.5002
```

So, we can actually come here and see what the output is. So, you can see, it is given 0, approximately 1 and it is given 0.5, which is not so great. So, we can run this whole code again. Let me just run this again, just to see what the update is. And now it has given 0 1 1 0. So, we saw, I mean incidentally, this was not planned, that for some bad initialization, you might not always get a great match.

I can always increase the number of epochs here and it will match well. But you can see that in general, if I even run this again, for a completely different set of weights. So, each time these weights will be different. It is a nonlinear regression problem; you need not obtain the same weights each time. But you get the XOR gate at least repeated. So, the first one is the Ground Truth 0 1 1 0.

y hat is coming approximately equal to 0 1 1 0. This was not the case when we use logistic regression. Now will this work only for XOR gate and we have to use logistic for let us say OR gate? That is not true. We can even basically run an OR gate classification and that will work completely fine. Of course, it is going to be a different set of weights now. But even OR gate works. What about AND gate?

So, AND gate would be 0 0 0 1. And if we run that you can see 0 0 0 1. This matches that well too. Now the problem of course, is whether it is AND gate, OR gate, XOR gate, or indeed a much more complicated classification problem. Here is basically just a four data point classification problem. Regardless of what problem you use, the neural network will tend to do a decent job provided you give enough hidden neurons.

The problem of course is we do not know what these w 1 and w 2 mean. So that is the problem. That is the standard problem of interpretability. So, the neural network works, but it is not interpretable. The key here in this code was doing this backprop that I showed you. So, we wanted to do the backprop explicitly.

In more complicated examples, like the ones that I will show you next, when we do physics informed neural network case, when we do a physics informed neural network case, you will see that that is a little bit more complicated. In fact, we will have to abstract out the details into the programming language that we are using. We cannot write these. This code will be sitting somewhere inside.

That in fact TensorFlow, PyTorch, or MATLAB as we are using. All these sorts of remove these away from the users so that you do not have to do this painful back propagation code. Nonetheless, it is important for you to see what sits here. Notice in particular this term, a 1 times 1 – a 1 when you want large change. As I had told you earlier, we tend to use ReLU rather than use sigmoid or tan h because this term tends to get small as you use large d networks.

So better to use the value wherever possible. I will explain why we do not tend to use ReLU for things like physics informed neural networks in many cases in the next video. So, I hope this was a little bit insightful for you to go through a backprop for a simple neural network for the XOR gate. I will see you in the next video with an actual example for physics informed neural networks. Thank you.