Introduction to Large Language Models (LLMs)

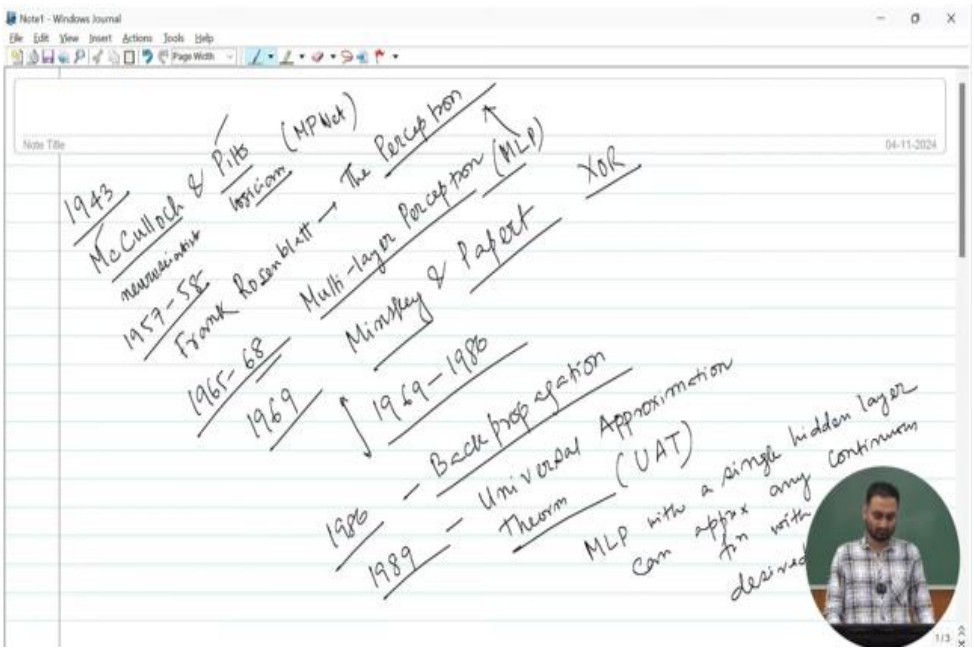
Prof. Tanmoy Chakraborty, Prof. Soumen Chakraborti
Department of Computer Science & Engineering
Indian Institute of Technology, Delhi

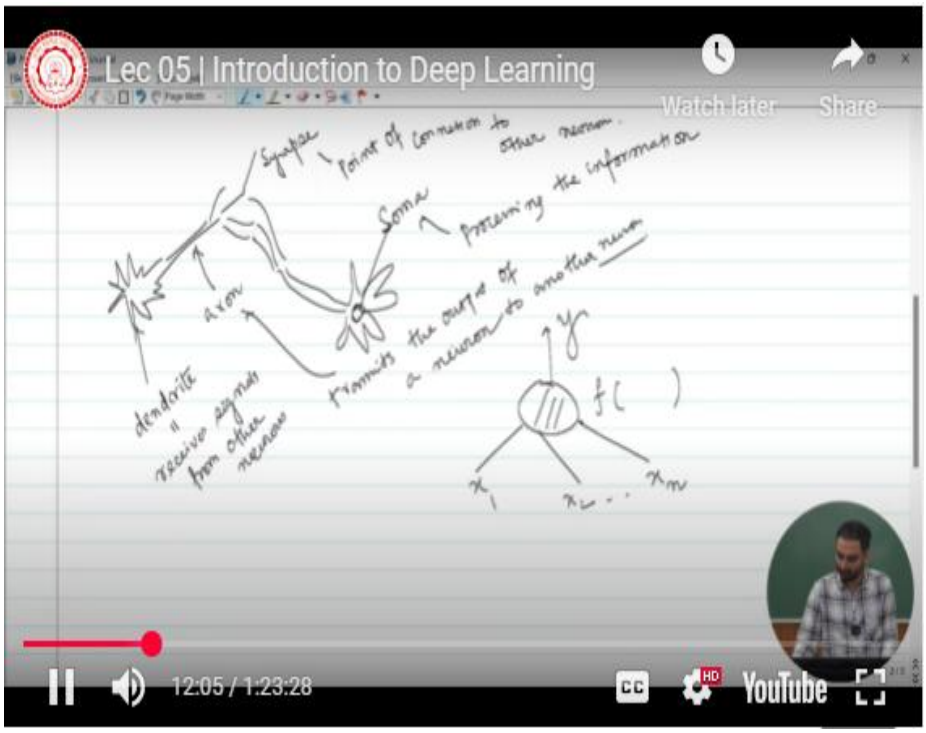
Lecture 5

Introduction to Deep Learning

Hello everyone, welcome to the course on large language models. So in today's lecture we are going to talk about neural networks basics. As usual this is not possible to cover neural network deep learning in a single lecture. So the aim of this part is essentially to give you a very brief overview of neural networks, how these networks operate and what are the ingredients of these networks. Because at the end of the day large language models are also neural networks deep learning methods. So I thought you know I should also give you some ideas about neural networks although there are you know many open source lectures available on YouTube, there are many good lectures available on NPTEL also on deep learning neural networks you can refer to those lectures, but here I will just give you a very brief of the neural network in general.

Okay, so when we talk about neural network, it basically goes back to long time back during World War I, World War II, you know, 1940s, 1943s, those were the time where people started thinking of how to mimic human brain using computational approaches, right.





So if you look at the history of neural networks, it goes back to I think 1943-1944 during that time when Michalok and Pitts, so Michalok was a neuroscientist. and Pitts was a logician. Now interesting fact to note here is there is no computer scientist involved.

Remember the time is 1943 before Alan Turing's invention. So at that time they thought of designing a very simple approach to mimic the way neurons essentially interact. And they came up with a simple method or simple model called MPNet, Mikolov's M and its P, MPNetwork, MPNet. I will discuss MPNet later. 1957-1958 the years may not be correct by the way This is just you know 1957-1958 during that time Frank Rosenbelt.

So he proposed the idea of the perceptron, which is considered to be the first neural network, realistic neural network proposed. Some people often call Frank Roosevelt as the father of neural network, but there are debates. And then later on around I think 1965, 1966 or 68 during that time multi-layer perceptron was proposed. So, this is a single perceptron, now multi-layer perceptron. We will discuss all these models in today's lecture.

Multi-layer perceptron, one proposed. So, this was proposed and Frank Roosevelt, you know, so he said that, so he speculated that the perceptron may eventually be able to learn, to make decision and to translate language. This was kind of a speculation at that time. 1968 MLP was proposed is called MLP multilayer perceptron and later on, in 1969 Minsky and So Minsky, some of you know Minsky is also called the father of computer science. He was the founder of AI lab at MIT.

Minsky and Papert, they identified a serious limitation of perceptron. They showed that a very simple function called the Zohr function, right, A XOR function, you know, cannot be modelled using a simple perceptron, right, and that was a huge setback because right after that, you know, kind of AI research got stalled for almost 20 years, okay, and this time is called the AI winter time, I think 69 to 1986 during that time. Then after 1986, I think around 1986, the famous back propagation method was introduced. All of us know that back propagation is essentially the building block of all the training protocols that we use in today's neural network. and 1989, I guess 1989 1990 during that time it was shown that you know this multilayer perceptron right is capable of approximating any kind of non-linear boundaries that you can think of and In that year, they came up with an idea of universal approximation theorem UAT was proposed and it was said that a multilayer perceptron with a single hidden layer can be used to approximate any continuous function to any desired precision.

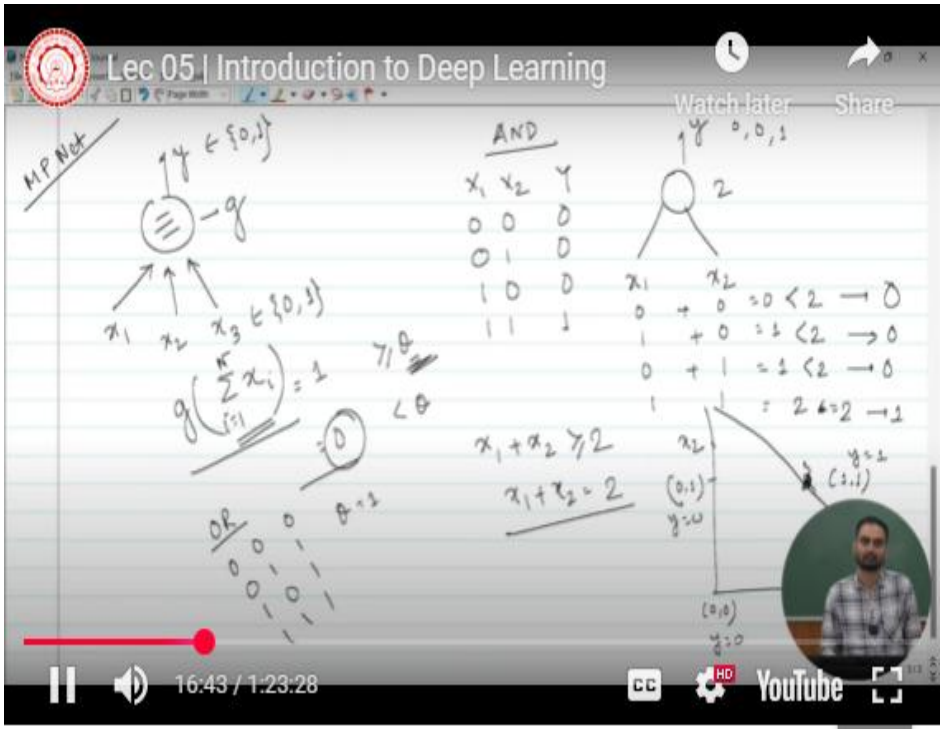
So a multilayer perceptron with a single hidden state, layer, can approximate any continuous function and this was a big claim by the way. This was a big claim because now we can use MLP for any non-linear classification that you can think of. By the way, I am not going to details of this theorem. As I mentioned, this is basically introductory part. So if you are interested in understanding all the details of this, you should basically go through a formal deep learning course.

Nevertheless, this is a very brief introduction of the history of neural network. So when you talk about neural network, the term neural network is there. Therefore, it has some connection with neuron. So you know that time 1960s, 1970s, 1980s, 1990s, even 2000, early 2000, there had been a huge effort to essentially mimic the way human thinks, right, how neurons interact. Now you know it goes back to I think 1860s or 1870s where when people understood that human brain essentially walks through interactions of neurons, right, and neurons collect signals from the environment you know from other neurons and these neurons essentially pass signals do some computation of course computation is a very vague term but neurons basically do some operation and that operation is fed to the other neuron as a signal right. So i think all of you are aware of biology the basic neuron structure right. There are, this, you know, you can think of this neurons which look like this i'm very bad in terms of you know drawing things but you know you can think of these neurons right. So this is one neuron and this is called axon right and you have this dendrite and you know this is called synapsis right. So where two neurons turn through which two neurons essentially interact can think of another neuron here right sorry for the bad drawing but yeah and through basically through synapses you collect signals from other neurons you pass it through axon and then you know you basically do some processing here.

This is the centralized part of the central part of neuron which is called soma. So, essentially the dendrite they receives signals from other neurons. So, these are the point of connection to other neurons, point of introduction. Soma is essentially responsible for processing the information and axon is responsible for transmitting, you have some neurons which are essentially collecting information, you have some centralized system which is the soma in our case which processes this information and it pass this, basically it does some

processing, it outputs something, the output moves to next neuron. So you have some inputs x1, x2, dot, dot, dot, xn.

So in here you do some processing, right. You can think of the processing as a function right which is parameterized which is basically which takes x1, x2, xn as inputs and produces y as an output right. So, this is a very simple you know simple way of describing this complex neuron structure. So, now the MPNet right which was proposed that time as I mentioned earlier this Mikolov-Pitts method.



So, the MPNet was essentially designed to mimic this simple system, okay.

So for an mp net uh what you have, now this is mp net, we call up its network, you have inputs. Now these inputs are binary, okay. These are basically either zero or one okay. This is infinite the simple neuron structure that you can think of right and And then what it does, it essentially processes this here. I'll tell you what they mean by processing things.

And it outputs something called y. This is also a binary variable. So what it essentially does, it basically takes the sum of all these inputs, sum of all xi's. 1 to n, n number of inputs,

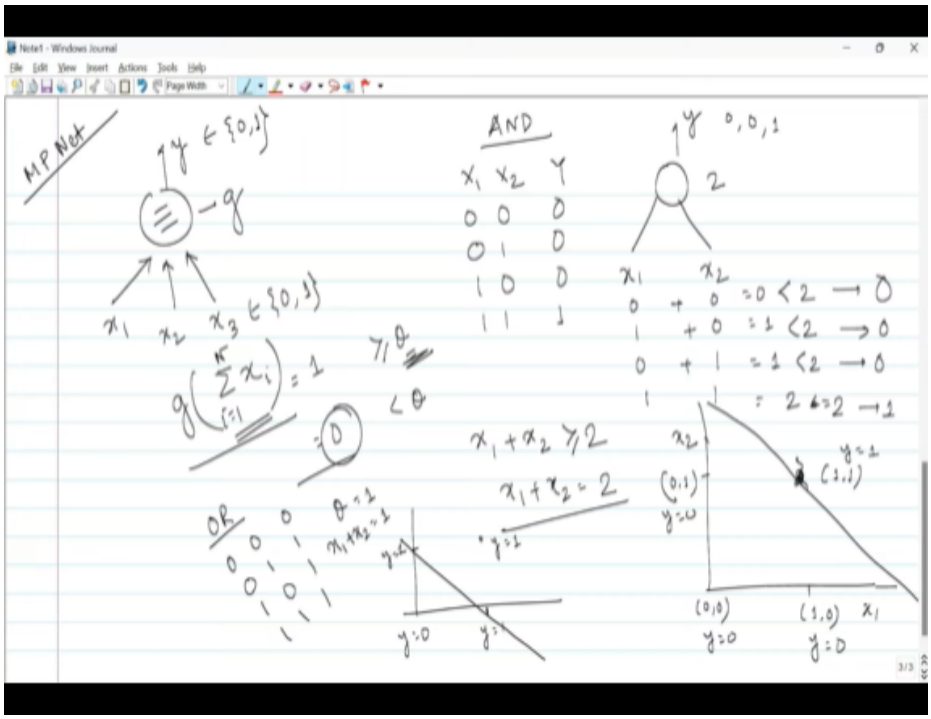
in this case 3 inputs, right. So and let's say you have a function here which is let's say g, right.

You pass it through g, right. So if let's say if the sum is greater than equals to certain threshold, right, you produce 1, otherwise 0, okay, a very simple structure. And with this, now this is a linear function, right? Now with this, you can, it's basically a step function, right? So with this, you can think of modeling different types of operations, right? Boolean operations, Boolean functions. For example, let's say AND. Okay, we know that and this and operation x1, x2 is input, y output 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1.

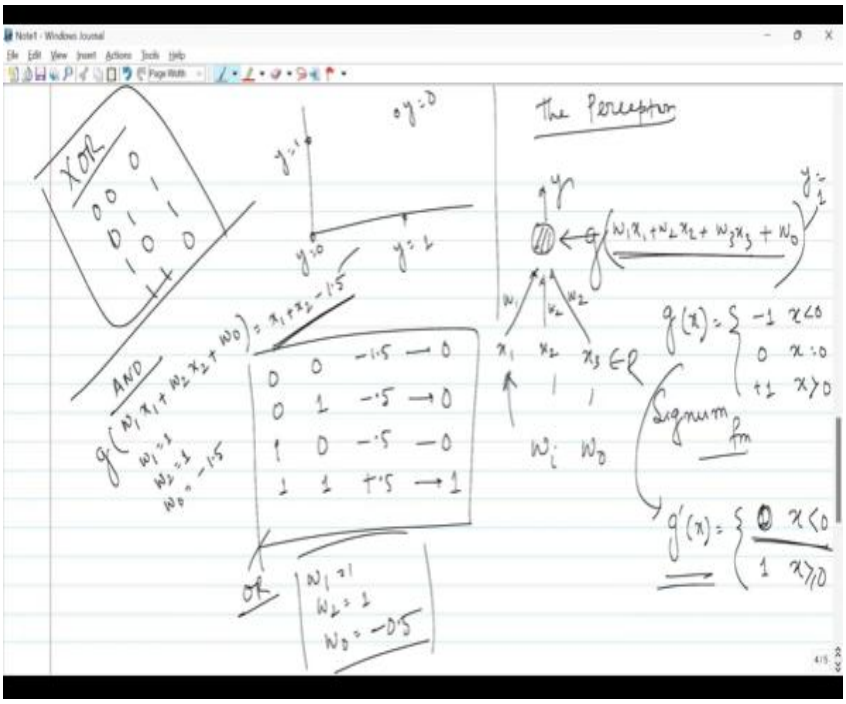
Okay, so how do we use MPNet to essentially mimic and operation? So, if you want to mimic and operation what is going to be theta? Think about it. Your input here, there are two inputs, right? x1, x2, right? And this is your output y, okay? What is going to be my theta? So if both these inputs are 0, it will produce 0. If one of them is 0, it will produce 0, right? If both of them are 1, it will produce 1, right? So the threshold can be 2. Right because 0 plus 0 is 0, this is less than 2, it will produce 0. 1 plus 0 is 1 less than 2 it will produce 0, 0 plus 1 is 1 less than 2 it will produce 0 and 1 plus 1 is 2, this is equal to 2 it will produce 1 okay. So, to mimic and operation the function that is going to be learned is essentially very simple x1 plus x2 should be greater than equals to 2 right. So the line is x1 plus x2 equals to 2 and anything above this would be classified as 1 otherwise okay. Now if you think of the vector space right you have four points essentially is x1 plus x2. So, let us say this is 1,1, this is 1,0, this is 0,0, this is 0,1.

So, this should be for here y equals to 1, y equals to 0, y equals to 0, y equals to 0. And the line that you have drawn would look like this. It will essentially touch this point. x1 plus x2 equals to 2. Now, in the same logic, if you want to mimic, let us say OR, which is (0,0,0), (0,1,1), (1,0,1), (1,1,1), what is going to be the theta? The theta, think about it.

It is very simple. The theta is going to be 1.



If any of them is 1, it basically attains the threshold theta, then it will produce the output 1. So in that case, the line would look like, so for this one, now this is 1, y equals to 1, this point this is y equals to 1, this point y equals to 1, this point y equals to 0, and the line is x1 plus x2 equals to 1. Okay, so it will basically would look like this. So, infinite is great when you have, you know, kind of Boolean inputs and Boolean output, right, and data points are linearly separable. Now why I said that data point is linearly separable, think about it here. So in both AND and OR we have seen that a linear boundary can separate data points.



But think of a complicated function, relatively complicated function, let us say XOR. 000, 011, 101, 110. If you look at this diagram, here y equals to 0, y equals to one, y equals to one, y equals to zero So this case there are four data points you will not be able to classify these four data points with a linear boundary with the line right and this was the setback this was observed by, you know minsky and Pippert that time 1968-1969 and they said that you know perceptron is useless because this is not able to classify a simple XOR or mimic a simple XOR operation, okay. Later on, you know, this MPNet is not very suitable, right? Therefore, the perceptron was proposed in 1957, right? Perceptron is also not able to mimic XOR, right? What is perceptron? How it is different from MPNet? Let's look at it here now. So the perceptron, not very different from MPNet. Inputs here is x1 x2 x3. Inputs are real numbers they may not be boolean okay and now with every input there's a weight w1 w2 w3 okay.

What does this weight indicate? The weight indicates the preference, right. Let's say the decision is whether you want to go to watch a movie or not. This is a decision that you need to take. And what are the inputs? The inputs are let's say the hero, the director, the duration of the movie and so on and so forth.

These are the inputs. These are the features. In case of MPNet, we have seen that all these inputs are equally likely, right? They are basically summed up, right? That means their importance are equal. Now, in this decision process, oftentimes what we have seen that some features are more important. For example, let's say for me, I'm a fan, right? And let's say if I see that there's a movie where Shah Rukh Khan is a lead actor, I will definitely go irrespective of the duration, the director and so on and so forth. So for me, the most important feature is who is the hero because I have a die-hard fan.

Whereas let's say there are other people who actually look at the other aspects, the director, let's say the producer, I don't know, maybe also the cost of the ticket and so on and so forth. For them, the other features are important. So therefore, you know, considering all these features equally may not be a good approach. We should also introduce weights and weights were introduced in perceptron. You know, now your sum here in the central part of the neuron, right, I will still call it a neuron.

Remember this, today's deep network, for example, all these LLMs, they are, I mean if you call them neural network, then people will essentially kill you, okay. So they are just nonlinear functions, right, general deviation, all nonlinear functions, that's all. So there was a time you know 1960s 1970s during that time when people essentially tried a lot to mimic human brain but these days nobody cares about human brain right. The task is essentially to make this models more complex, more effective and so on and so it doesn't matter whether they mimic you know human brain or not. So here the sum is like this w1 x1 plus w2 x2 plus w3 x3 right along with this, there is a bias term w0.

We have seen all of you know right in machine learning we always use this bias term for many purposes right so we have this bias term w0 okay. So now this sum will essentially be fed to some function g, right. We have seen this g function or g or f whatever, right. I think we use g here. So this function will be fed to a g and the output of g will be given to y, right.

Now in case of perceptron, g was essentially a function which would look like this. So g of x is minus 1 if x less than 0, it is 0 if x equals to 0, it is plus 1 if x is greater than 0, right. So, this function is called signum function. So, you take this input and process it, right.

And if the input value is greater than zero it will produce one as an output right, otherwise you know minus one or zero.

So later on when we again reform this perceptron we observe that we don't need minus one zero plus one. There are two levels which are enough so we modified this, we said okay Let us use this modified function which would look like this which will produce one or which will produce zero. If x is less than 0, otherwise 1, if x is greater than 0, greater than equals to 0, right. So, we are happy with two levels, 1 or 0 because remember these methods are proposed for classification, mostly for binary classification, okay. This is perceptron, you have inputs, weights, you add those sum, you basically sum them up multiplied by weights and sum them up and pass it through a signum function and it will produce the output, okay.

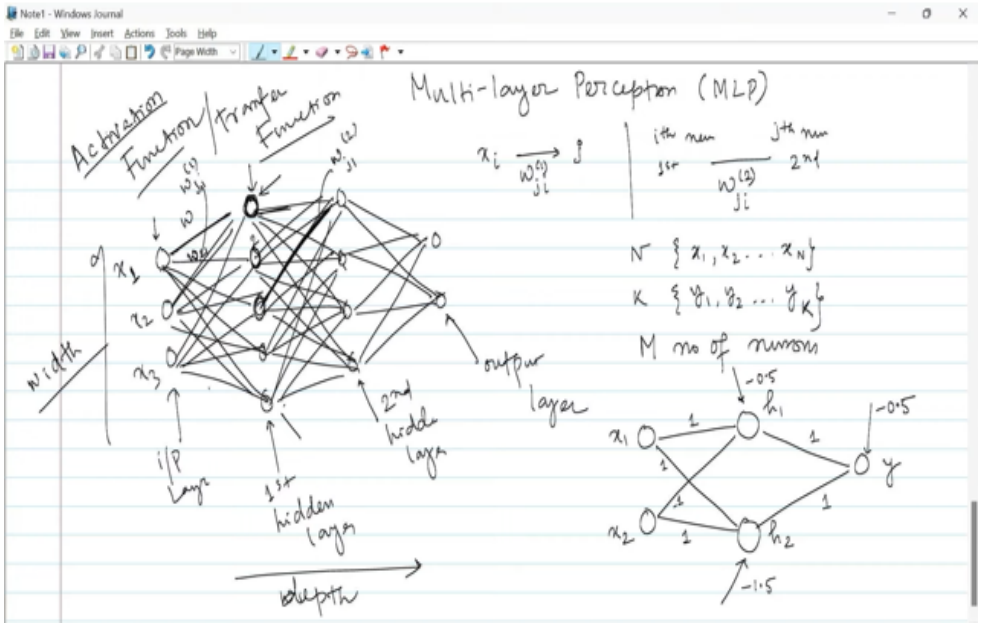
So if you do that, let's see what happens with AND, right, this AND. So what are the parameters that we need to learn here the parameters that we need to learn here in this case all these w's and w0 weights and bias okay. So to learn to mimic and right there are two inputs x0 and x1 whatever x1 and x2 right. So this is w1 x1 plus w2 x2 right plus w0 right you pass it through the signal g and how do we in case of and let's say if w1 is 1, w2 is also 1, w0 is minus 1.5. Okay so it will essentially be a thing about it so this will look like x1 plus x2 minus 1.5. This is my classifier right the line if x1 and x2 are 0 both of them are 0 then this is going to be minus 1.5 which will be this, which will satisfy this, so the output will be 0. If one of them is one then this will be minus point five right this will again be zero.

If both of them are one then this will be plus point five and this will be one okay. So it is able to mimic AND right. Similarly we will see to mimic OR w one should be 1, W2 should be 1 and this bias term W0 will be 0.5 and you will see that it will essentially with this parameters will be able to mimic OR. So whatever functions that MPNet can mimic, perceptron can also mimic, right? Here, the advantage is that we have weights.

You can give advantages to more emphasis to certain weights, certain functions, certain inputs, and less emphasis to certain features. But the problem remains the same, bizarre, right? It is not able to classify XOR because remember there is no non-linearity here. These

are all lines, right? 2D plane, line, if you see 3D plane, a plane, right? 3D coordinates, a plane. There is no non-linearity.

This g, right, is still a step function.



So how to address this XOR function, right? So then you know Mikhailov and other scientists, they came up with the concept of activation function. I intentionally have not used the term activation so far. Although, you know, all these Gs used in perceptron, MPNet, they are all activation functions, but I have not used them so far intentionally because when you talk about activation, we mostly refer to a nonlinear activation. The activation function or it's also called transfer function, right. So and this activation transfer function is mostly used in multilayer perceptron. So far we have talked about basic perceptron. Now we are talking about multilayer perceptron MLP. MLP is used in transformer everywhere. In MLP what is the speciality here? Now here you have concept of layers okay, input layer, hidden layers, right, an output layer okay, x_1, x_2, x_3.

For example there are three inputs right, all of them are connected. So this is also called feed forward network. All of them are connected to each other right. All of them are connected to each other, it's a mess, right so okay. Now so this is input layer, this is the first hidden layer, this is second hidden layer okay and this is output layer.

Now if you notice my drawing i have intentionally made two different hidden layers with different number of neurons. In the first layer there are five neurons, one, two, three, four, five. Second layer there are four neurons. Now each of these neurons itself is a perceptron, isn't it? If you look at this one, inputs are coming here and it will generate some output.

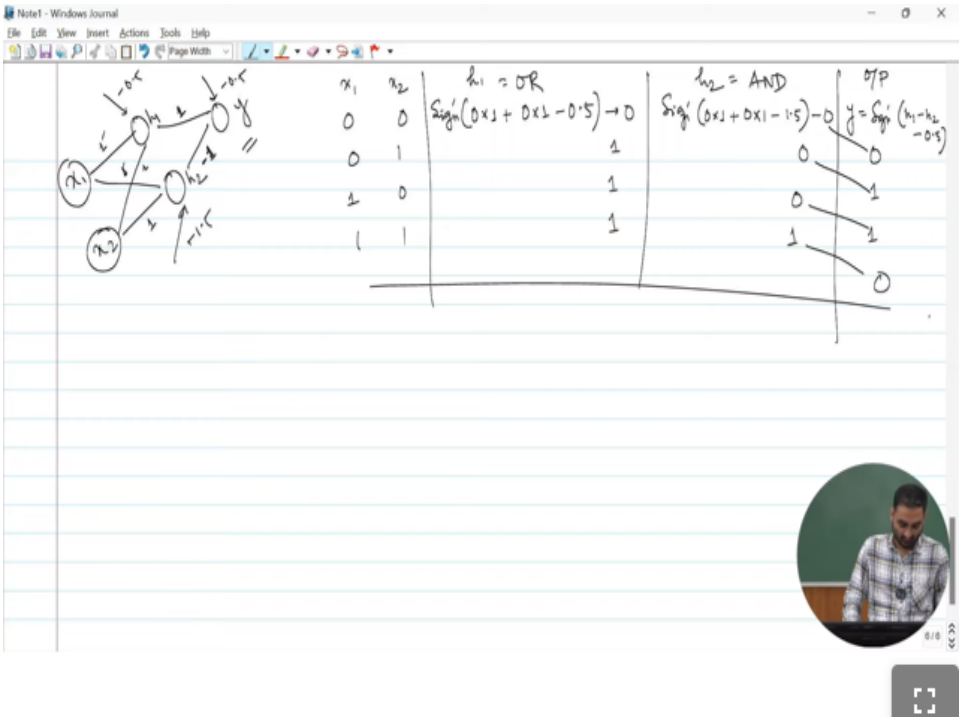
Now this output is fed to all the other neurons in the next layer. With every connection there are weights, these are weights right and here notations are very important by the way. So you can use any convention but here in this lecture we will use a specific convention. Let's say input 1 is connected to input neuron 2 right so the weight will be w21. So if xi is connected to a neuron whose index is j the weight is going to be wji. This is the convention that i follow right and there are different layers since input is in this layer and hidden state is in this layer so the weight will be for layer 1.

So the superscript indicates the layer. For example, let's say this specific weight. Now this is connecting i-th neuron in this layer, first layer to j-th neuron in second layer, the weight is going to be $w_{ji}^{(2)}$. So, this one will be $w_{ji}^{(2)}$ and this one is $w_{ji}^{(1)}$. So, we assume that there are n number of Again, notations are very important.

So, pay attention to the notation. There are N number of inputs $\{x_1, x_2 \dots x_N\}$. There are K number of outputs $\{y_1, y_2 \dots y_K\}$, capital K and there are let us say M number of neurons, M number of neurons in a hidden state. And let us assume just to make things simple, let us assume that there are layers and every layer there are equal number of neurons, same number of neurons, okay. So, the number of layers in a neural network is called the depth of the neural network and number of neurons in every layer is called the width of the neural network.

And these are hyper parameters that you need to fix to start with. Okay. Now we see that how this MLP can mimic the XOR operation. Okay. Two inputs x1 and x2. Let us say there are two neurons in a hidden state and one output y. Okay so now how to calculate the weights and biases right? We'll discuss later when you talk about back propagation but let us assume that the weights are as follows.

This is one, this is also one, it's also one, it's also one okay, and biases here is minus 1.5 this is also minus 0.5 and this is one weight this is one weight and with this neural network, with this perceptron the bias is minus 0.5 okay Now this is the structure right, now this structure is enough to mimic the XOR operation okay.

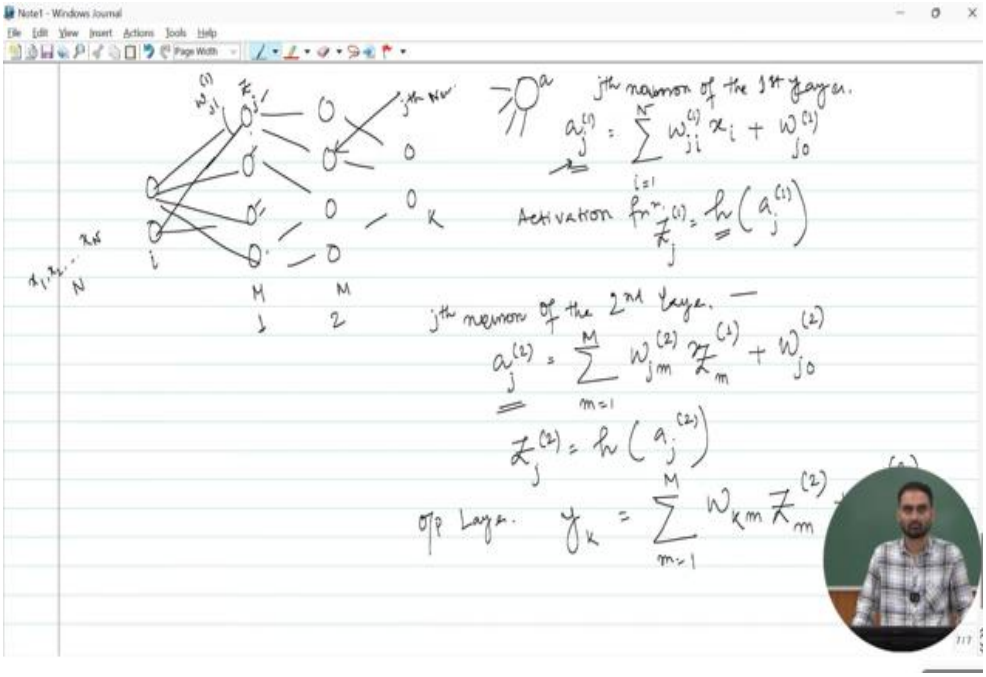


Let me redraw again. So x1, x2, h1, h2, y is 1, this is also 1, this is 1, this is 1, right the bias here minus 1.5, there's a bias here minus 0.5, 1 this is a bias minus 0.5 So think about it x1 x2 right and here also we use the modified signum function as an activation, 0, 1, okay. What is H1 here? So H1 is the signum function, modified signum function, 0 times 1 plus 0 times 1 weights.

These are weights this one and this one right minus 0.5 okay and the output, So this one so this is minus 0.5 so the output will be 0. Similarly here you will see the output will be 1, this is 1, this is 1, this is the output of h1, so h1 essentially acts as a OR H2 is what? H2 is signum 0 plus, you will see is a product of 0, 0, 0, 1.

This will act as an AND. Output here Y is again signum. There is a small mistake here, this would be minus 1. The weight here is 1, this weight is minus 1. Do not worry about how we came up with this weight. Let us assume that the weights are known to us.

Later on, we will see how we came up with this weight. So, this is essentially h1 minus h2 minus 0.5. This is minus 1. And if you do the calculation, you will see that this will produce 0, this is 1, and so this is 0, this is 1, this is 1, this is 0. So now with multilayer perceptron, we will be able to mimic this XOR, which is the beast, the XOR function.



So far we have discussed basic architecture of multilayer perceptron. Now let's focus on activation functions, which is activation function is essentially the core part of neural network. But before that, let us look at the architecture once again, multilayer perceptron. Let's say there are two layers, two hidden layers. So sometimes we don't consider the input layer and the output layer as layers.

So when you say three layers, you basically refer to three hidden layers. So these are inputs. I'm not drawing all the connections. We know all the connections are there. So, let us say there are n number of inputs as I mentioned $\{x_1,\ x_2 \ldots x_N\}$, right.

There are M number of neurons, okay, also connected and there are K number of output. Okay and the notation you know the convention that we followed is that there is a connection from ith input to jth neuron the corresponding weight will be W of Ji layer 1 okay. So at every neuron there are essentially two operations that are happening, that are going on. One is this aggregation operation, right? It aggregates all these inputs, right?

Aggregation is just a sum, a weighted sum, and then we have an activation function, right? So aggregation operation is denoted by A, okay? So let's say at this neuron, at the jth neuron, of the first layer, so aj1 is the aggregation right for that neuron. So this is sum over all the inputs i equals to 1 to n right w j i one x i pay attention to notation again right and we have a bias j zero. Now this bias is fixed for a certain neuron for this neuron for jth neuron the bias is wj0.

$$a_j^{(1)} = \sum_{i=1}^{N} w_{ij}^{(1)} x_i + w_{jo}^{(1)}$$

Let's say for ith neuron the bias is wi 0 and so on and so forth So this is the aggregation. So after the aggregation, what happens? After the aggregation, it will essentially be passed through an activation function, right? So the activation function, activation function is denoted by, let's say h, right? So $h\left(a_j^{(1)}\right)$ is the output of the activation function and let us denote it by Z, $Z_j^{(1)}$. So Z is the output of the activation function of the Jth neuron. So A is called pre-activation output and Z is called post activation or whatever activation output.

$$Z_j^{(1)} = h\left(a_j^{(1)}\right)$$

So now this activation function is very important. So far in case of perceptron, impunate, this was just a linear function, step function, right. We will discuss about this activation functions in today's lecture. But before that, so here all these Zs will be computed here in layer-1, first layer. And the second layer, what will happen? In second layer, there is a neuron which is let's say the jth neuron of the second layer for this neuron what is going to be a j2? This will be sum over right all m, because there are m neurons which will basically produce outputs and all those outputs will be fed to the next layer, to every neuron of the next layer.

So, m ranges from 1 to M. w jm zm1 okay plus the bias wj02.

$$a_j^{(2)} = \sum_{m=1}^{M} w_{jm}^{(2)} Z_m^{(1)}$$

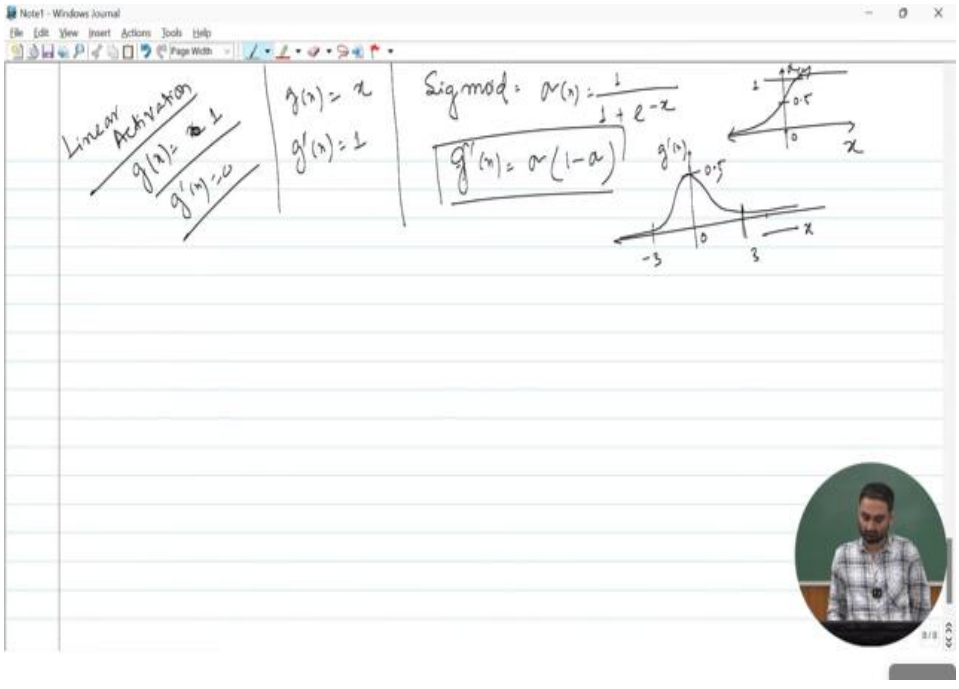
$$Z_j^{(2)} = h\left(a_j^{(2)}\right)$$

Please pay attention to the notation okay. This is very important try to memorize the notation if possible. So this is the pre-activation output for the jth neuron of the second layer right. Let's say this is the guy right jth neuron jth neuron okay.

The z the post activation output will be h right. Now what is going to be? So this is the second layer now the output layer Let's say yK. yK is what? yK is going to be sum over, again let's say there are M number of neurons in the second layer. M equals to 1 to M, right? WKM ZM2 plus WKM. So, this is the output.

$$y_K = \sum_{m=1}^{M} w_{km} Z_m^{(2)} + w_{km}$$

So, if you understood the notation, lets now move on to this activation function H. So, the basic activation function that you can think of is a step function that we discussed.



So, let us say linear activation function. a linear activation function. Let's say gx equals to x, okay.

$$g(x) = x$$

If we take the derivative of this activation function, why do we need to take the derivative of the activation function? Later on when we will talk about back propagation, you will see that these derivatives would essentially contribute to the weight update.

We need to take the derivative of the error function with respect to different weights. Now, when we take the derivative of the error function with respect to the first weight, for example, you need to follow the chain rule. So whatever actually follows between the weight and the error, all those components will be considered while taking the derivative. And one of these components is going to be this activation. Derivative of this activation is very important right. Let's say you have a an activation like gx equals to one, okay a constant activation function, whatever input you give it always produces a constant number.

So problem here is if we take the derivative right this will be zero Okay so d dot derivative doesn't make any sense if we have a linear activation like this cx equals to x so if you take the derivative of this c of x equals to g dash of x is going to be 1 This is again a constant derivative. So constant contribution will be added or subtracted to the existing weight. That is also bad. So therefore what people thought, people thought that this is the right place to introduce nonlinearity. So what are these nonlinear activation functions? The simple non-linear activation function is the sigmoid function which you all know the sigmoid and this is used heavily right 1 by 1 plus e power minus x okay.

$$a = \frac{1}{1 + e^{-x}}$$

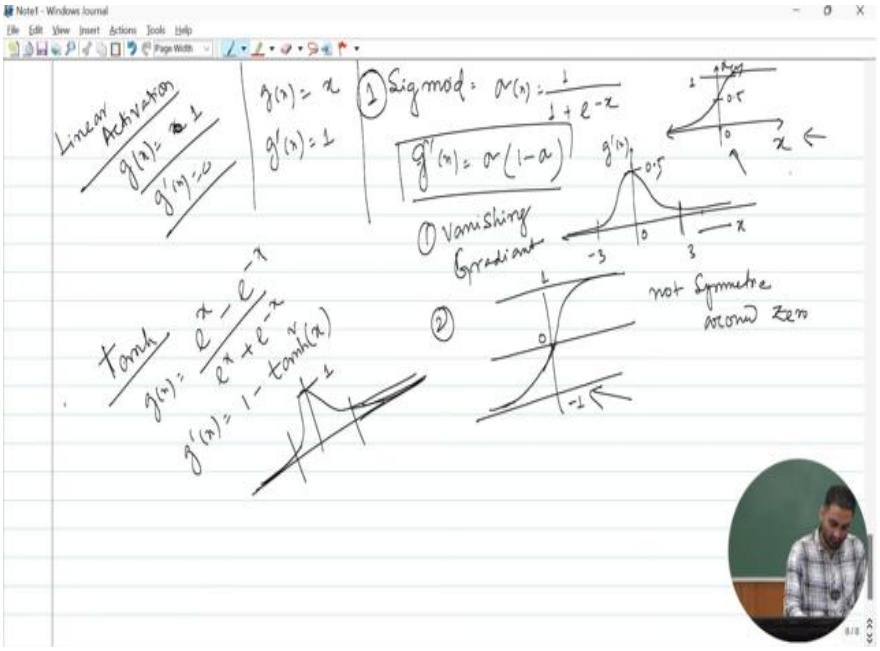
So and we know that the sigmoid essentially squashes everything between 0 to 1. This is 0.5, this is 1, this is 0. So, whatever input you give is x and this is sigmoid x, yx is a sigmoid x. Whatever input you give, it always squashes everything between 0 to 1. The good part about sigmoid is that you take the derivative of this, let us say g dash x is sigmoid into 1 minus sigmoid.

$$g'(x) = a(1 - a)$$

You don't need to compute the derivative explicitly. If you know the sigmoid, you will get the derivative of the sigmoid very easily. So this is a good activation function. The problem here is as follows. The problem here is the derivative, if we take the derivative of sigmoid, if we draw it, you see it would look like this.

At 0, you see that the value is 0.5 and then after certain threshold, so this is x and this is g dash x, the derivative of the sigma. After certain threshold, you see the value of the derivative is very, very small.

This is generally minus 3 and 3. If the input is less than 3 or greater than 3, you see that... the output will be very very small that means it will contribute to the weight of that you know very small and this would further lead to vanishing gradient and all the other problems right. Vanishing gradient problem we will discuss in the RNN part of it right.



So this is the first problem vanishing gradient problem. This is not exactly the vanishing gradient problem, but let us refer to it as a vanishing gradient problem.

The gradient will vanish slowly. The second problem is that the sigmoid is not symmetric around 0. This is not symmetric around 0. What do you mean by symmetric around 0? Let's say instead of this function, if we had a function like this, this is 0. So this is symmetric

around 0. Why is this important? It is important because we will see later that generally, you know, symmetricity is always important, always good, okay.

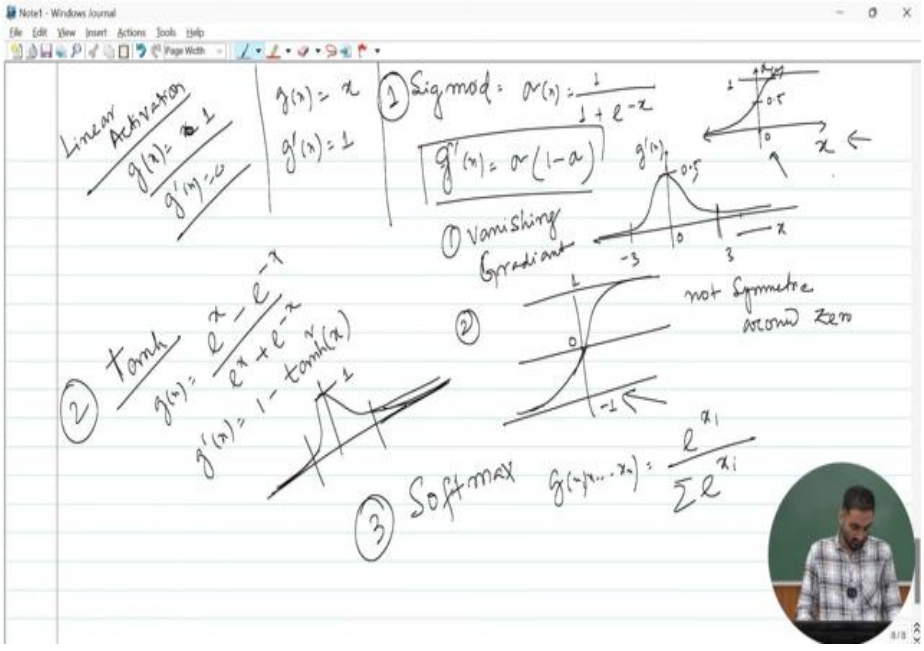
So this is not symmetric around 0. How to address this? To address this, we will come up with another activation which is called the tan h. tan h activation, this is a hyperbolic function. So tan h is g of x e to the power minus x and the derivative of this g of x is essentially 1 minus tan h square x.

$$\tan h = g(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$g'(x) = 1 - \tan h(x)$$

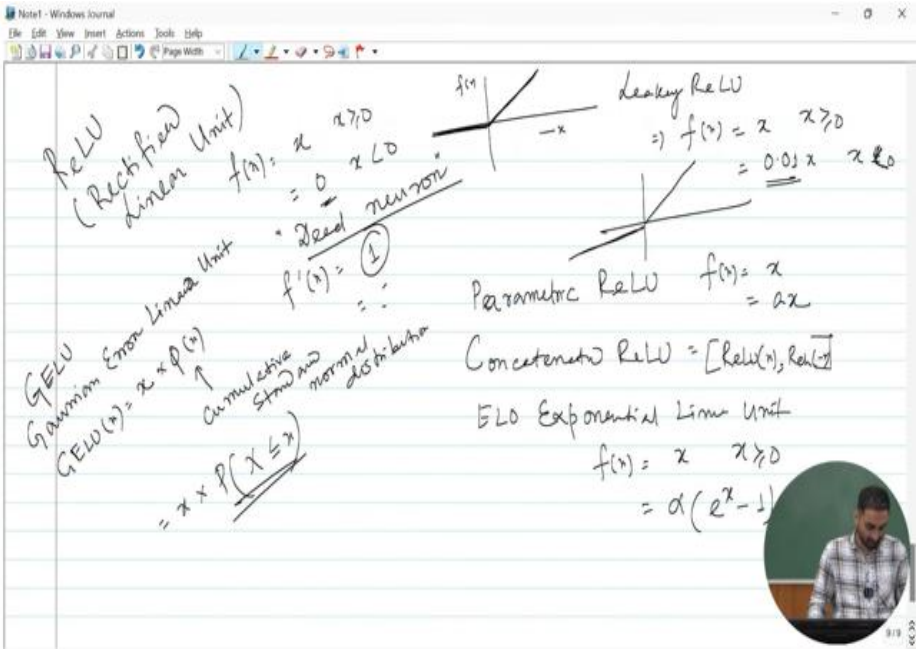
Here also you do not need to explicitly compute the gradient. So tanh, what it does, it squashes everything. So this is the tanh function, right. It squashes everything between minus 1 to 1, you know, unlike sigmoid which reduces everything to 0 to 1, this essentially reduces everything between minus 1 to 1.

This is symmetric. The bad news here is, here also you see the same kind of plot. for the derivative. This is 1. After certain threshold, the value tends to become very small. So the same vanishing gradient problem may occur. Meanwhile, people also started using, so this is 1, this is 2.



The third activation is softmax. We all know what is softmax, right? Softmax is used everywhere. So softmax is essentially g of x, basically x1, x2, dot, dot, dot, xn, e to the power x1 by sum of e to the power xi. It produces a probability.

$$g(x_1, x_2 \ldots x_n) = \frac{e^{x_1}}{\sum e^{x_i}}$$



The other popular activation function is ReLU, rectified linear unit and it is used everywhere. It is very popular.

What is ReLU? So ReLU function is very simple, x equals to x, if x is greater than equal to 0, otherwise 0. That means it chops off all the negative values. It looks like this. Positive values are linear, so this is x, this is f(x) and otherwise 0. Now this is very interesting because this is a very simple activation function that turned out to be very effective.

But here the problem is as follows. So let's say if the input is negative, now remember this activation function is applied on top of A. What is A? A is a pre-activation value. So if the pre-activation value is negative, there will be no activation.

If it is negative, it will be zero. Post activation, this will be zero. It means that that particular neuron will be dead, right? So this is called dead neuron problem. And that's bad because

you expect that, you know, the model should be able to capture the negative part, the positive part suitably, right? If we take the derivative of this, this will produce 1 and this is again 0. So constant addition for positive values but 0 addition for negative values.

This dead neuron is a bad news about ReLU. To address this, many variations of ReLUs have been proposed. Leaky ReLU is one variation which essentially says that f of x is x otherwise 0.01 of x. You give some weightage here.

$$f(x) = x; x \geq 0$$

$$= 0.01x; otherwise$$

So, it basically looks like this and some weightage to the negative part also. So, this addresses the problem of dead neuron. because you have some value for this negative pre-activation neuron. There is something called parametric ReLU where instead of fixing this to 0.01, we make this. where a is a learnable parameter.

$$f(x) = x;$$

$$= ax; otherwise$$

Now sometimes you can see or you can say that you know negative part is bad but we should also capture negative part in some ways. So instead of taking ReLU of x, can we take the ReLU of minus x $[ReLU(x), ReLU(-x)]$. That would solve the problem of the negative part, right? So another variation of ReLU is proposed which is called concatenated ReLU. Now here it will produce a vector of size 2, okay. The first component of the vector is ReLU x, the second component is ReLU minus x, right.

So ReLU minus x will essentially capture, I mean if the pre-activation value is negative it will be positive, right. There is another variation called ELU, exponential linear unit. It is f of x is x, if x is greater than or equal to 0, otherwise alpha e to the power x minus 1. They introduced this exponential component here.

$$ELU =$$

$$f(x) = x; x \geq 0$$

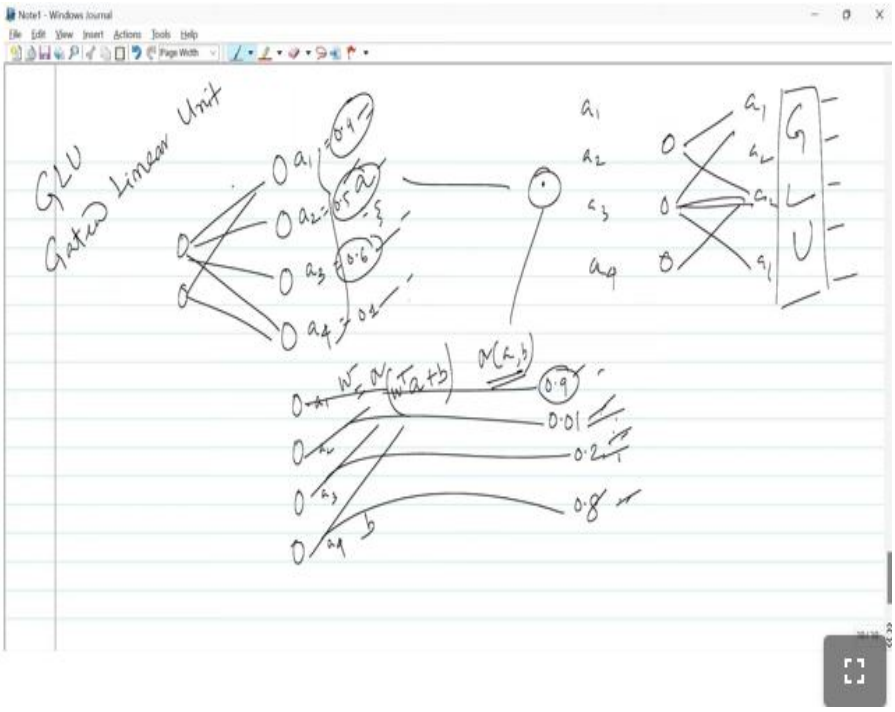
$$= \alpha(e^x - 1)$$

Now these are small variations of ReLU. Let us look at some of the major variations of ReLU. There is another variation called GELU, Gaussian Error Linear Unit. So GELU of x is x times phi of x. what is this phi of x? phi of x is the cumulative standard normal distribution. what is that? So it is x times p of is a standard cumulative function, this is basically captures the quartile the percentile, rather than the signal okay.

$$GELU(x) = x \times \phi(x)$$

$$= x \times P(X \leq x)$$

So this is GELU.



Now let us look at another paradigm of activation which is called GLU right, this is gated linear unit. We'll talk about gating mechanism in the lstm, GLU part okay. Now what is this GLU activation? So let us assume that you have hidden neurons, inputs and hidden neurons. And let's say a1, a2, a3, a4, they are pre-activation values. In GLU, what you do, you just duplicate this one here. So let's say a1 a2 a3 a4 okay, you pass it through some

weight w this weight is different from this weight by the way okay and then what you do and then you have also a bias right.

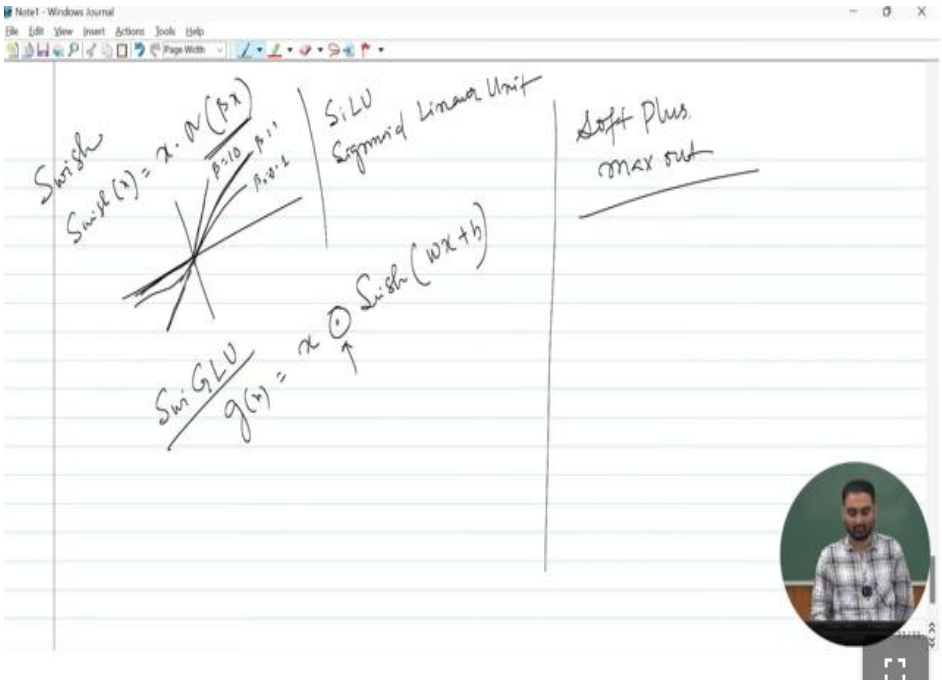
So this will be a so $(w^T a + b)$ okay and you pass it through a sigmoid So you have original A and you have a sigmoid of A, sigmoid of A or B, whatever, sigmoid of A, B, right? And then you do getting. And what is this getting? This getting is an element-wise multiplication.

Remember, A is a vector which consists of a1, a2, a3, a4. The sigmoid is also a vector, right? So let's say this is a1 is 0.4, this is 0.5, this is 0.6, 0.1 right and after sigmoid let's say this value is going to be 0.9. Let's say this is 0 right whatever 0.01, these are all imaginary numbers by the way, okay, these are all imaginary numbers.

So 0.2 or something like that 0.8 okay. So when we do element-wise multiplication, this will basically be 0.4 times 0.9, 0.5 times 0.01, 0.6 times 0.2, 0.1 times 0.8. And this will produce another set of A, A1, A2, A3, A4. Now these weights are learnable. These are trainable parameters. B is also a trainable parameter. All right. Here, this is called gating mechanism because you will essentially train or learn these Ws in such a way that some of these outputs will control the impact of this pre-activation values to the next layer.

Let's say this is 0.9. That means you give more weightage here. This is 0.01, you give less weightage here. This is 0.2, you give slightly less weightage here and so on and so forth. Okay. So you can think of, so these inputs, okay, you think of, so this is A1, A2, A3, A4, you think of a GLU layer here, okay, which will produce numbers.

GLU remember is a parameter w and p that you need to learn okay.



So the next part activation function is called swish activation. There are tons of activation functions which are publicly available and people are still proposing new activation functions every day swish activation is very interesting this is So, swish x is x times sigmoid beta x.

$$Swish(x) = x \cdot a(\beta x)$$

Beta is again a trainable parameter, right. So, essentially you are multiplying a non-linear part with a linear part, right.

So, the beauty of this swish function is as follows. If let's say beta is 10 it would look like this. If beta is, drawing is not good by the way, beta is 0.1 it looks like this. If beta is 1 it looks like this.

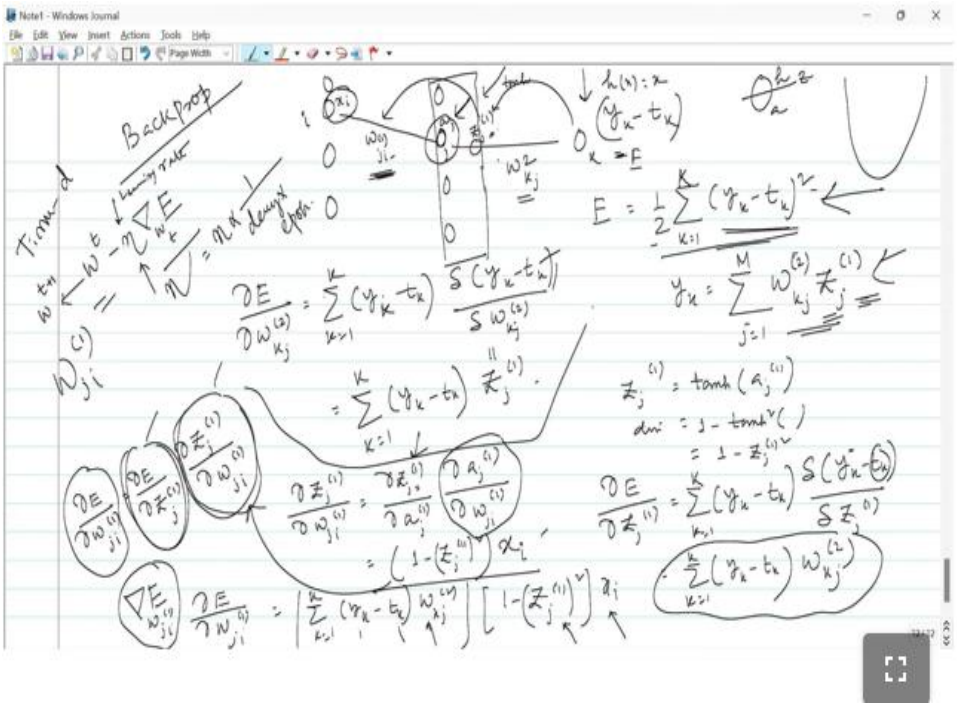
So if you see this when beta is 10 it is almost a value function. If beta is 1, it is quite it is basically a linear okay and you have things in between. So with the tuning of beta you can get linear function and so on and so forth. Sometimes this is also called SILU sigmoid linear unit, okay. And in another paper, people combined SWISH activation and GLU activation and people call it SWIGLU.

Look like this, X, so G of X is X SWIS wx plus b okay.

$$g(x) = x \odot Swish(wx + b)$$

This is element wise dot product that we used here all right. There are many other activations like soft plus right max out and so on and so forth. Some of them are computationally challenging, computationally complex. Remember this gradient? So we need to take the gradient of the activation, right? Therefore, this should be computationally easy, right? So that is another factor.

All right.



So the next part of the lecture, we'll talk about back propagation. Let's draw a diagram again, okay? A simple multi-layer perceptron. And let's assume that there is only one hidden state and only one output.

Let's make our job simple. So, this is i-th input, j-th neuron, k-th output. So, this is $w_{ji}^{(1)}$. and this weight $w_{kj}^{(2)}$ which is input x_i. Now, as I mentioned every neuron has two operations right, A and activation H which will produce this Z right. So, let us bifurcate this up to activation right, to operation. Let us assume that in this component, a is being computed, here z is being computed, $w_{kj}^{(2)}$. So, when we compute this yk, right, we also

know let us assume that you also know the ground truth for this input right and the ground truth is tk.

So what is the loss? The loss is yk minus tk and let's say here the loss or the error is a square, right, square loss. so error e is sum over total error right, So how many outputs are there? There are k outputs yk minus tk square and we take this half because later on we need to take the derivative this two and half will cancel out. So this is my error. Now we will take the derivative of this error with respect to all the weights and we'll update all the weights right. So we'll take the derivative of this error.

This is the derivative of the error, right? Let's say with respect to w, some w, okay? And this will be, let's say this is a convex function. So weight will be updated. This is the learning rate, standard concept, standard machine learning concept, stochastic gradient descent. And this will be my updated error, updated weight.

So let's first focus on this weight update, Wkj2. So to update this, we need to take the partial derivative of this with respect to Wkj2, okay. That means we need to take the derivative of this with respect to Wkj2, right. So, this is x square, so 2x, so 2 and this half will cancel out. So, this will be sum over yk minus tk, k equals to 1 to k. Okay now we also need to take the derivative of this with respect to the partial derivative of yk minus tk with respect to wkj.

$$\frac{\partial E}{\partial w_{kj}^{(2)}} = \sum_{k=1}^{K} (y_k - t_k)$$

$$\frac{\partial E}{\partial w_{kj}^{(2)}} = \sum_{k=1}^{K} (y_k - t_k) \frac{\delta (y_k - t_k)}{\delta w_{kj}^{(2)}}$$

Now when you take the derivative of tk, tk is a constant, it's a ground truth, so this will be zero. What is yk? yk is sum over okay so i forgot to mention another thing let's assume the activation here at the output is a linear activation hs equals to x whatever input you get that's going to the output. So yx is essentially the aggregation right wkj where j goes from 1 to m, there are m number of neurons and zj. Now, when I am, this is 2, when I am taking this derivative with respect to kj2, only that element will contribute, right.

$$y_k = \sum_{j=1}^{M} w_{kj}^{(2)} Z_j$$

So, this is, essentially this is going to be, and this is 1, okay. It doesn't matter if there is only one layer but still it's okay. So the output of this will be this. So the total output is like this. Now remember here. So this is straightforward because there is only one hop.

Now when we update this one, this is difficult. so del e del w j i 1 okay, now from here, this is error this is e, till here. What are the operations that we have performed? We have performed z and a. so we have to use the same rule right so this is essentially del E del z1 del zj1 del wji1 okay Now what is this? Let's look at this one first.

$$\frac{\partial E}{\partial w_{ji}^{(1)}} = \frac{\partial E}{\partial Z_j^{(1)}} \frac{\partial Z_j^{(1)}}{\partial w_{ji}^{(1)}}$$

Now del zj1 del wji1 is what? $\frac{\partial Z_j^{(1)}}{\partial w_{ji}^{(1)}} =$? So we have another intermediate part which is this one because I am now moving from this to this.

There is an intermediate part which is a. We have to take the partial derivative with respect to a also. del A1, del Aj1, del Wji1, okay.

$$\frac{\partial Z_j^{(1)}}{\partial w_{ji}^{(1)}} = \frac{\partial Z_j^{(1)}}{\partial a_j^{(1)}} \frac{\partial a_j^{(1)}}{\partial w_{ji}^{(1)}}$$

What is this? What is this Z? So Zj1 is nothing but this activation function. Let us assume that the activation function here is tan h. So, this is essentially tan h of a j1, what is the derivative of this? This is 1 minus tan h square, this is the derivative.

This is 1 minus square. So, this will be 1 minus zj1 square, this part.

$$Z_j^{(1)} = \tan h\left(a_j^{(1)}\right)$$

$$= 1 - \tan h^2$$

$$= 1 - Z_j^{(1)^2}$$

Now, what is this part? Think about it. So aj here, aj1, aj1 is nothing but the weighted sum, right? Weighted sum of all the inputs, right? And when we are taking the derivative with respect to wji1, only the corresponding input will be responsible. All the other inputs will act as constant, right? So this derivative will produce xi, right?

$$= \left(1 - \left(Z_j^{(1)}\right)^2\right) x_i$$

If it is not clear please go back and check the chain rule okay so this is my result for this component What about the other component del E del zj1 right? This is zj1 and del E. so E is what? E is this one right where we already know that y is this one right so when we take the derivative of this with respect to sorry z is this one when you take the derivative of this with respect to z g what will happen is that first this will act as simple x square 2 x right 2 and 2 will cancel out only the derivative of this and then we need to take the derivative of that component again.

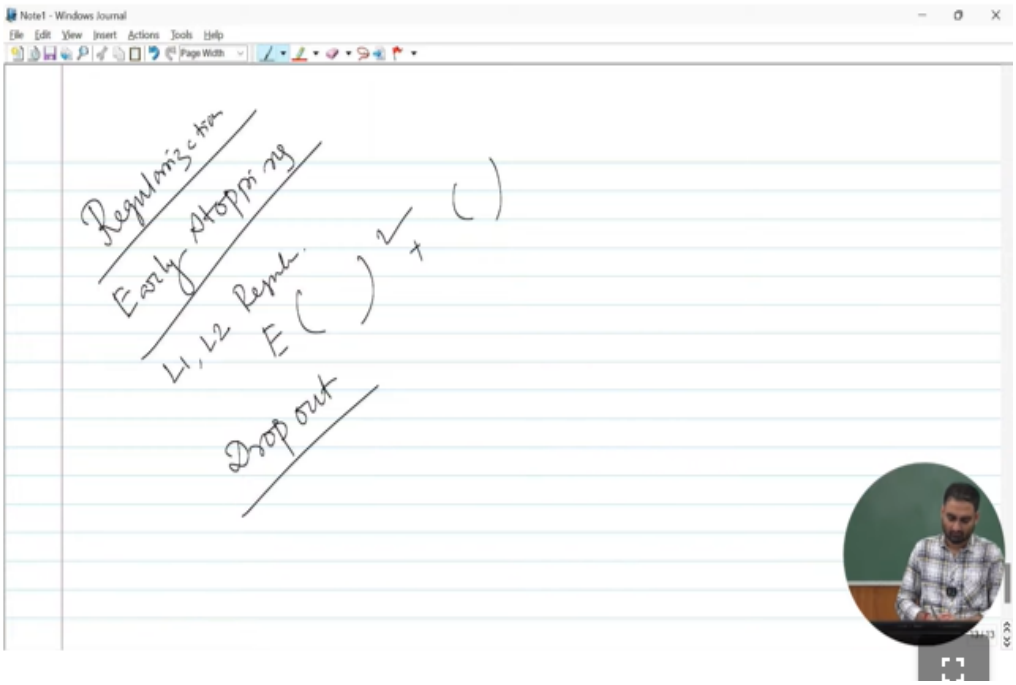
What will happen, only the corresponding z will basically come out right. We are taking the derivative of this with respect to z so only w part will come out okay. so del e let me write here del e del z j 1 is sum over y k minus t k, k is 1 to k ok, then the partial derivative of this, yk minus tk say 1 ok and this is yk minus tk ok tk is constant here so yk is this one. So this will result in Wkj2. The final derivative del E del Wji1 will consist of this, okay, which is sum over k goes to 1 to k, yk minus tk, Wkj2.

So, this part and this part which is 1 minus zj square xi. Now, think about it. So, this is this delta E with respect to wji1. When you do the forward pass this is back propagation backward pass right when you did the forward pass you already stored the value of y the output you also know the ground truth t and what is this w w kj2 is the existing weight right which will be updated right so existing weight is also known to you z j is also known to you this was stored during the forward pass and x i is the input. So all the components are known to you therefore this gradient will also be known to you and we will use this learning rate multiply this with the gradient and that is going to give you the updated w j i with respect to w j i at ith layer. Now think about it as we increase the number of layers

okay You will have to multiply several such gradients, right? Think about it. So this error and there are let's say hundreds of layers, right? And you want to update the weight of the first layer.

So you have to take the gradient of the error with respect to that weight. So all the intermediate components will be considered while taking the derivative. And let's assume that the gradient of the error with respect to these components are very, very small. Right so what will happen the result is a lot of zeros a lot of small numbers will be multiplied and therefore the result in multiply the multiplication output is going to be very small So this is called vanishing gradient in case of multilayer perceptron feedforward networks. As we increase the number of layers, the contribution of the first layer, the contribution of the error in updating the values of the weights and the initial layers will be very small. So initial layers weight will not be updated at all, right? This is vanishing gradient.

Discuss vanishing gradient in rnn recurrent neural network right there are many ways to address the vanishing gradient problem. So that's it, this is back prop what are the parameters here? The parameters are all the weights, bias, right, learning rates, learning is hyper parameter sometimes we fix learning rate and sometimes we also use some sort of dynamic learning rate you know with respect to epoch so sometimes we use something like this one minus dk times epoch time dk learning rate also for regularization right



Because what happens is that neural networks, if you have small amount of data, deep neural network, the models will overfit.

There are many ways to address this. One is early stop. You will not allow the model to basically produce 100% accuracy or zero error in your training set. on your training set. So you stop the model early before convergence. The other option is, of course, L1, L2 regularizer. So what you do, that along with the error, whatever error you have, let's say square error or log likelihood, some sort of cross entropy error, you also add a regularizer there.

The other option is called dropout. where you essentially freeze a fraction of neurons for being updated. So you are basically making the neural network sparse. Again, there are different ways to update this thing, identify the neurons which need to be dropped out. There are many, many concepts, important concepts there in deep learning which we have not covered because this is not a deep learning course.

As I mentioned, there are many deep learning courses available even on NPTEL. You can check them out. Vanishing gradient problems I mentioned, exploding gradient problems are also mentioned. Some of these things are written in the book also, I mean my book. So you can look at Chapter 2 for detailed description of some of the concepts that are discussed

here. With this, I stop here and this was about the introduction of neural network and deep learning. In the next lecture, we will move to the main concept of large language models. Thank you.