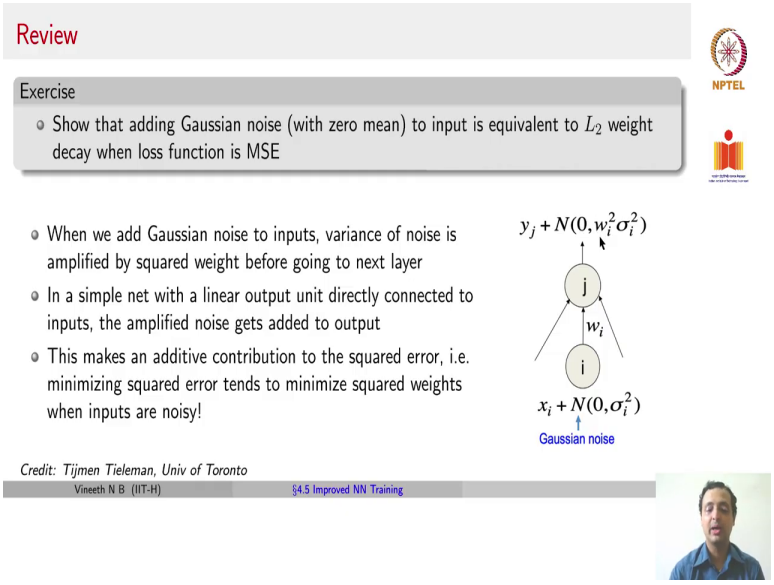
Deep Learning for Computer Vision
Professor Vineeth N Balasubramanian
Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad
Lecture 29
Improving Training of Neural Networks

In the last lecture we spoke about how regularization is an important component to improve the generalization performance of Neural Networks, we also spoke about a few different ways of performing regularization. In this lecture we will talk about other components that can help improve the training of Neural Networks.

(Refer Slide Time: 0:46)

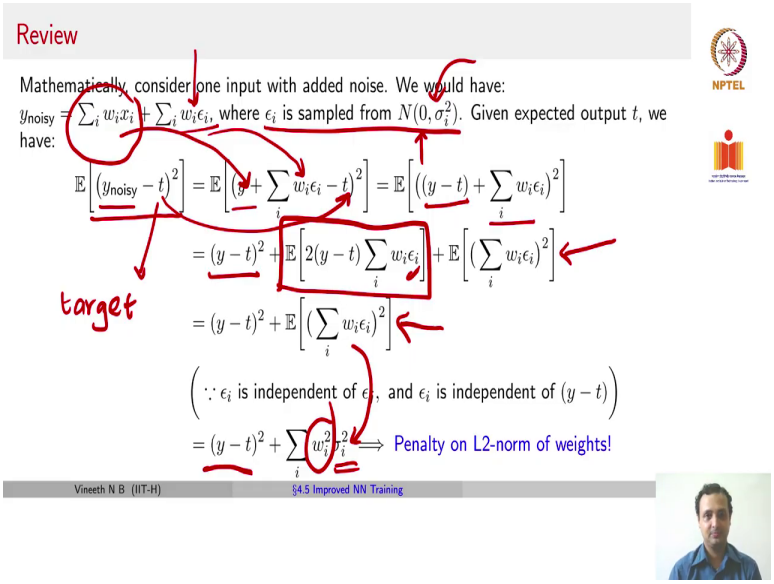


Before we go forward, we left one question behind, your exercise was to show that adding Gaussian noise with zero mean to input is equivalent to L2 weight decay when your loss function is mean squared error. Hope you had a chance to watch it out. Let us try to understand why this is the case.

So, when you add Gaussian noise to inputs the variance of the noise is amplified by the squared weight as you go to the next layer of the neural network. So, which means if you simply had a simple neural network where the output layer was directly connected to inputs, the amplified noise would directly get added to the output. And that added output makes a contribution to the loss function term, the squared error term and that is what makes it look like an L2 weight decay in the loss function.

So, you have Gaussian noise, say, sampled from a distribution with zero mean and variance (σ^2) and when you send it through a layer it is amplified by w and it goes through on the output layer with a new variance which is amplified by w_i^2. That is the intuition, let us work it out mathematically also.

(Refer Slide Time: 2:22)



So, mathematically if you consider one input with added noise, which means you would have your output $y_{noisy} = \sum_i w_i x_i + \sum_i w_i \epsilon_i$. Remember here that originally your output would have been only $\sum_i w_i x_i$. So, this is a component that got added to the output because you added noise to the input. And what is the noise? ϵ_i which is sampled from a Gaussian with mean 0 and variance σ_i^2.

Now, let us try to look at what happens to the mean squared error. So, your expected mean square error now, so your new mean squared error is going to be $E[(y_{noisy} - t)^2]$, where t is your ground truth or target. So, that is the correct value. So, your expected mean squared error

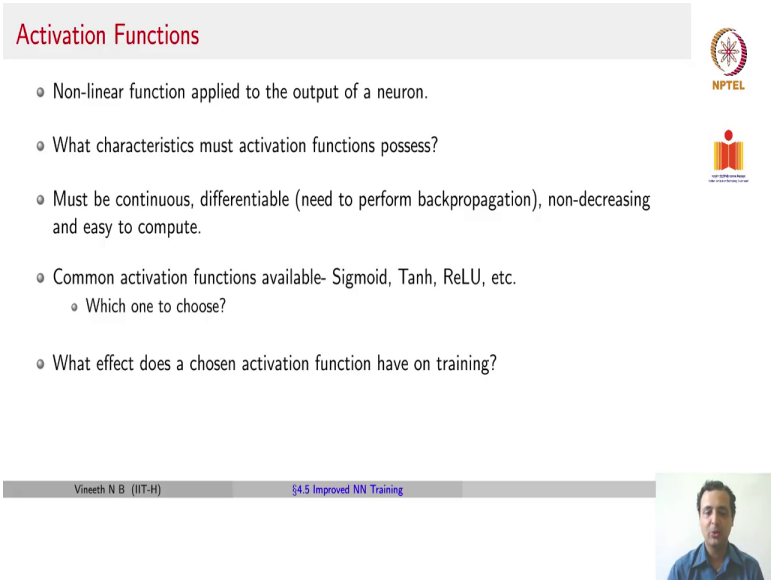
will be given this way, let us expand it out, we know that $y_{noisy} = \sum_i w_i x_i + \sum_i w_i \epsilon_i$ term that comes here and then you have the $- t^2$ that comes this way.

Now, by grouping terms this can be written as $((y - t) + \sum_i w_i \epsilon_i)^2$, now we use your standard expansion of $(a + b)^2$ and you get the terms on the next equation. Keep in mind that the first term here does not have an expectation because there is no random variable there, there is no ϵ_i in that, so you can remove the expectation in that first term. And the expansion follows the linearity of the expectation.

So, now if you see the middle term here, we assume now that all weights have a mean 1 and all ϵ_i are going to be mean 0 by the very definition of how ϵ_i is sampled. Which means the $E[\sum_i w_i \epsilon_i] = 0$, so you are left only with the first term and the third term.

Now, this tells you that this third term can now be written as the expectation can be pulled in and this can be written as $\sum_i w_i \sigma_i^2$, because that is the variance of ϵ_i. This as you can see clearly looks like a penalty on your L2 norm of the weights where the coefficient of your penalty is given by σ_i^2. Rather when we add noise to the input your expected mean squared error, the new loss function rather is going to look like your old loss function plus an L2 term.

(Refer Slide Time: 5:55)



Let us move on now to this lecture's focus which as I said is going to talk about a few different components on how you can improve the training of neural networks. The first component that we are going to talk about is activation functions. In the very first lecture of this week, we gave an introduction to activation functions. We are going to spend some more time now to understand each activation function and its pros and cons.

We know now that an activation function is a non-linear function that is applied to the output of every neuron in your neural network. Traditionally people use the same activation function for the entire neural network or at least a layer. Why do you do this? Why can't you change the activation function for every neuron in the neural network?

The simple reason for this is to keep things less complex, if you have the same activation for all neurons in a layer, you can use matrix vector operations directly to make your computations rather than have to compute values for each neuron separately. And matrix vector operations are already optimized using linear algebra subroutines and we would like to take advantage of those fast subroutines to make computations in a neural network.

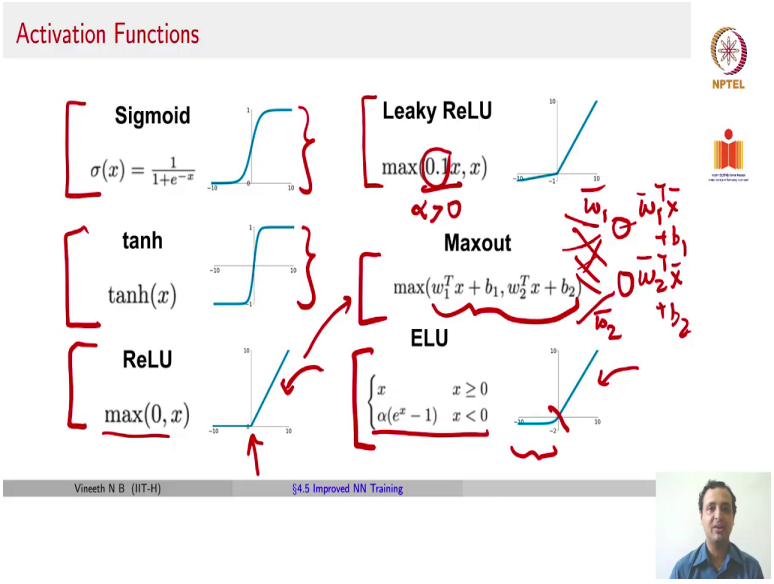
Let us now try to ask what characteristics must activation functions possess? What do you think? By now you must know that they must be differentiable, because otherwise you cannot back propagate gradients. But what else should they possess? What kind of functions can become activation functions?

We have seen a few so far, we have seen the sigmoid, we have seen the tan hedge, we have seen ReLU, so on and so forth. Do you see any commonality in the shape of these functions, in the nature of these functions? There are a few important aspects, the functions must be continuous so that you do not have discontinuities where the gradient is not defined.

It must be differentiable because we need to perform back propagation. It must be non-decreasing too. You do not want an activation function which is something like this, at least ideally speaking because as the input increases you do not want the output of the activation to be fluctuating between high and low values, so you ideally want it to be monotonic or non-decreasing.

And obviously you also want it to be easy to compute. Remember anything hard to compute is going to make your neural network slower. Common activation functions that we have seen are sigmoid, hyperbolic tangent, rectified linear unit and so on. Which one do you choose, does any of these activation functions have a specific effect on training? We are going to see that over the next few slides.

(Refer Slide Time: 9:18)



Here are some of the activation functions that we visited earlier too. Let us quickly review them before we look at each of them in more detail. The sigmoid activation function was one of the earliest activation functions we used to overcome the threshold problem in perceptrons and it is a logistic function given by this shape.

The hyperbolic tangent is also a logistic function that looks very similar in shape to sigmoid, but for its range being between minus one and one, whereas sigmoids range is between 0 and 1. The rectified linear unit developed in 2012 as part of the AlexNet contributions has a form given by this.

The equation for this graph is $max(0, x)$. If the input is negative the output is going to be 0, if the input is positive the output remains as it is, the input itself. A variant of ReLU is known as Leaky ReLU where when the input is negative your output is going to be $0.1x$, 0.1 is just an example, this could be any value α which is reasonably close to 0, it is not 0, but it is somewhat close to 0.

There is a particular reason why Leaky ReLU was proposed and we will see that in some time. Let us see ELU before coming to max out. ELU is Exponential Linear Unit and ELU was developed because the ReLU activation function actually seems non-continuous, it has a discontinuity at 0 where it is not differentiable.

In practice people get around this by assuming that the derivative at that particular location is 0, which is valid because for those of you who are aware of the concept of sub gradients, 0 is a valid sub gradient for ReLU at that particular point, so 0 is a valid choice, but there are other gradients possible at that particular location.

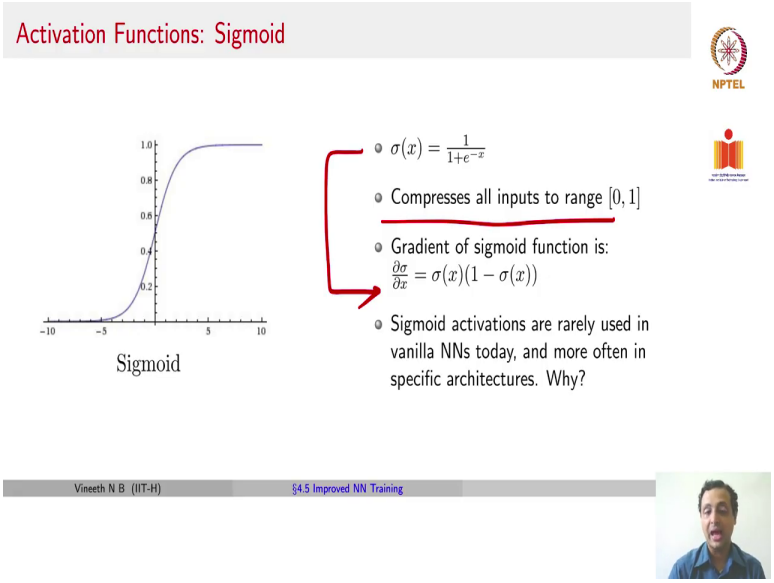
Nevertheless, it is a point of discontinuity and people get around this by assuming the gradient of ReLU at 0 to be 0. If you did not want to make that assumption ELU is an option which is a smooth continuous function, which is given by for any value greater than 0 it is x just like what we saw for ReLU, but for values the input values less than 0 we assume it to be $\alpha(e^{x} - 1)$ which has a shape as you can see here. As you can see here this now becomes a smooth continuous function with no discontinuities.

Another activation function that was proposed is known as Maxout, it is used occasionally these days but not as popular. Maxout is you can look at maxout as a generalization of ReLU because of the max function there. But it does not take a max on a single activation on a single neuron output, it takes a group of neurons and takes the max value of that group of neurons as the output of all of those neurons.

In this mathematical definition, you see it written in terms of two neurons, remember here, that this would be the output of two of these neurons, let us assume that these were two neurons in some network, some layer of the network. So, assuming that the weights coming in to this neuron was w_1 vector, the weights coming into the second neuron was w_2 vector, so the output of this neuron would be $w_1^T x + b_1$ but the output of this neuron will be $w_2^T x + b_2$.

So, now the activation of both of these neurons is given by the max of these two values, it is not a very traditional activation function, the activation function is not defined on a single neuron but on a group of neurons. Now, let us try to visit some of these at least in more detail and try to understand how they can affect the training of neural networks.

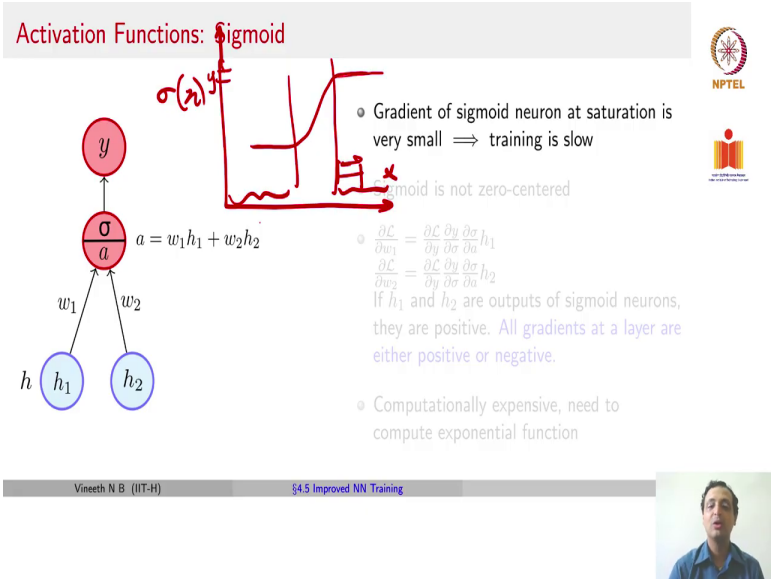
(Refer Slide Time: 14:10)



Let us consider the earliest one sigmoid which is given by $1/(1 + e^{-x})$, clearly it compresses all inputs to range (0, 1). The gradient of the sigmoid function hope you have work this out, this was a homework which is $\partial\sigma/\partial x = \sigma(x)(1 - \sigma(x))$, it should follow very easily from this definition of sigmoid.

But while sigmoid activation functions were very popular in the 80s, 90s, they are not used as often today. Do you know why? Of course, one reason is today you have other activation functions such as ReLU which was not available in the 80s, 90s and 2000s but there are specific reasons why sigmoid neurons can create certain problems.

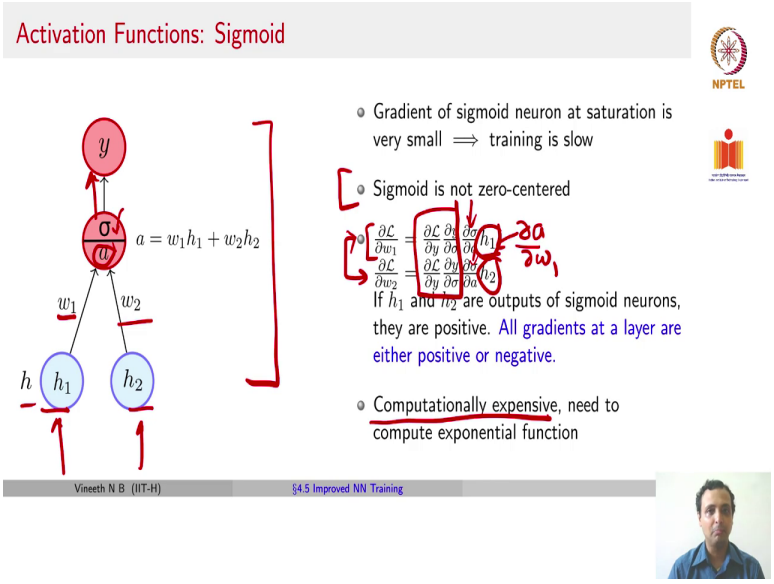
(Refer Slide Time: 15:10)



The one important problem is if you look at the shape of the sigmoid which is somewhat like this, you see here that when the input is very small or even very large, remember that your input is along the x axis here, the y axis is your $\sigma(x)$. When your input is small or large, even a very large change in the input may result only in a very small change because this function is almost flat after a certain point.

This is the case whether the input is small or whether the input is large, this could create some problems for us while training, because even as you keep increasing an input you would find that it has no impact on the gradient for certain weights in the neural network where you have a sigmoid, this can slow down training.

(Refer Slide Time: 16:17)



Sigmoid is not 0 centered because its output is in the range between 0 and 1 which means if you want your activation to consider negative values and positive values differently, sigmoid will not be able to do that for you, let us also see now another reason why sigmoid could have certain problems, let us look at this neural network here, a very simple neural network where you have your inputs or rather a part of a neural network, your inputs in a particular layer for a particular neuron are h_1 and h_2, the weights are w_1 and w_2, the weights are combined to become a, you apply a sigmoid and then get a y, one simple perceptron or one simple part of a neural network.

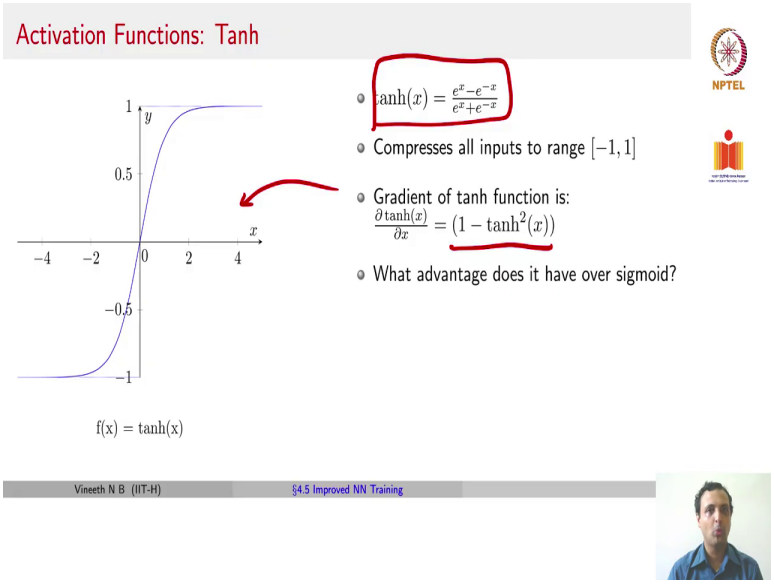
Let us analyze how the gradients flow in this part of the network using back propagation and chain rule, you perhaps know now that if I had to find out the gradient of the loss function L whatever the loss function was for this neural network $\partial L/\partial w_1$ is going to be given by $\partial L/\partial y * \partial y/\partial \sigma * \partial \sigma/\partial a$ where a was the pre-activation of that neuron before applying the sigmoid activation and then into h_1.

Why h_1? Because h_1 is going to be $\partial a/\partial w_1$. Similarly, for w_2, if h_1 and h_2 were also outputs of some sigmoid neurons from a previous layer, they would both be positive, which means you can see now that h_1 and h_2 would be positive, $\partial \sigma/\partial a$by the definition of the gradient of the sigmoid function would also be positive, which means both these gradients will assume the sign of this quantity which is common for both the gradients.

$\partial L/\partial y * \partial y/\partial \sigma$ is common for the gradient of w_1 and w_2, which means if that gradient turned out to be negative both w_1 and w_2 will have negative gradients, if that turned out to be positive both w_1 and w_2 will end up having positive gradients. Rather, this may not just be for one specific neuron, it could hold for an entire layer where you use a sigmoid activation function because all of those activations because they are outputs of sigmoid will be positive. This may slow down learning too.

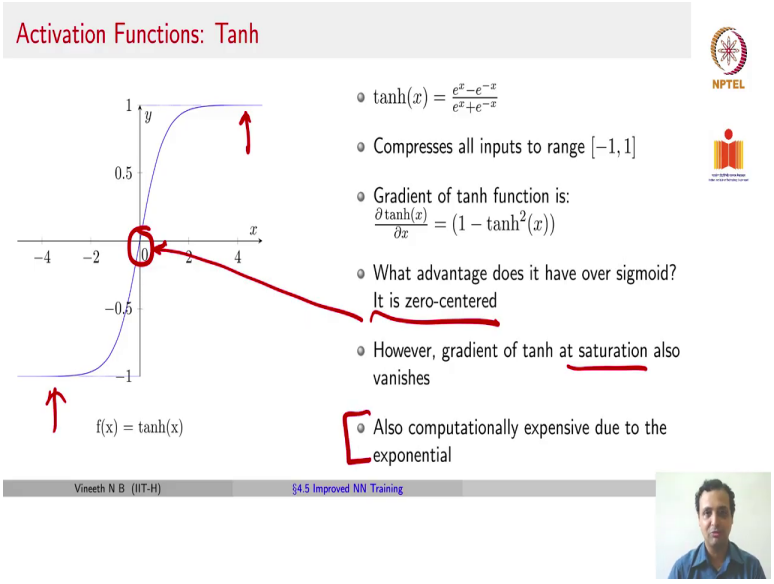
Also sigmoid is inherently computationally expensive. Why? Because you have to compute an exponential function, remember an exponential function always takes time to compute, an exponential function is an infinite expansion in terms of a polynomial series and it always takes time to compute at the most basic level.

(Refer Slide Time: 19:42)



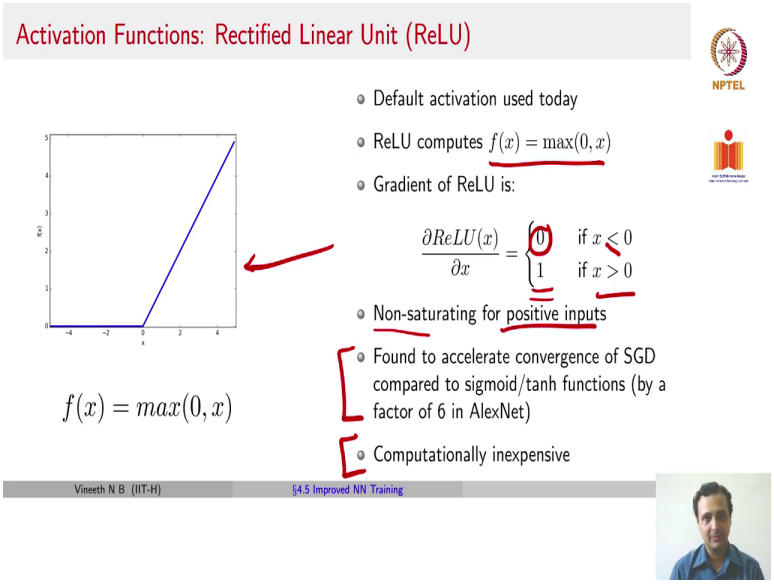
What about hyperbolic tangent? Hyperbolic tangent is given by this equation and this visual form, it compresses all inputs to a range (-1, 1) and the gradient of the $tanh$ function is given by $1 - tanh^2(x)$ which you may have worked out as homework again. What advantage do you think this has over a sigmoid activation function?

(Refer Slide Time: 20:10)



It is zero centered, which means the outputs of a $tanh$ activation function can be both negative and positive which could help in learning better. However, very similar to sigmoid it also suffers from some disadvantages, it also has saturation at lower inputs and higher inputs as you can see from the shape here, any logistic activation function may suffer from the same issues. It is also computationally expensive because of the presence of the exponential functions.

(Refer Slide Time: 20:52)

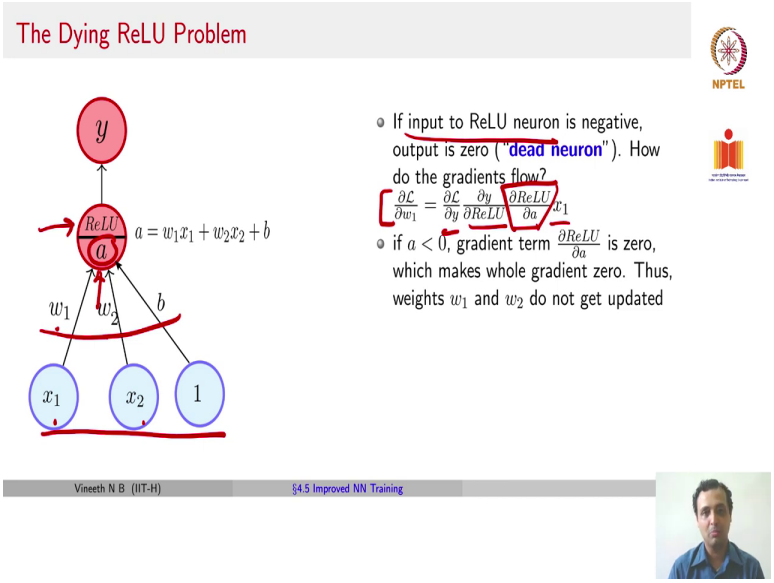


Moving on to the rectified linear unit which as I mentioned was proposed as part of the AlexNet in 2012 is the most often used activation function today, its equation is $max(0, x)$ and here is the graph form of the activation function, the gradient is defined by 0 if x is less than or equal to 0 to be honest because at 0 you just define it to be 0 and if x is greater than 0 the gradient is 1 because $f(x)$ is x itself.

For positive inputs the ReLU activation function is non-saturated which means if your input keeps increasing or if you give a higher input to a particular neuron, the ReLU will ensure that that input also gets a higher gradient based on how it contributed to the final loss, which is considered desirable, you do want a weight to get its reasonable share of the gradient depending upon how it contributed to the output.

In the AlexNet work in 2012 this was found to accelerate convergence of SGD compared to sigmoid or $tanh$ functions by a factor of six. In other words convergence happened six times faster using a ReLU activation function as compared to using a sigmoid or a $tanh$ activation function. It is computationally very cheap, all that you need to do in an implementation is threshold and just move on, if it is greater than 0 just take the value and move on, if it is less than 0 set it to 0 and move on.

(Refer Slide Time: 22:50)

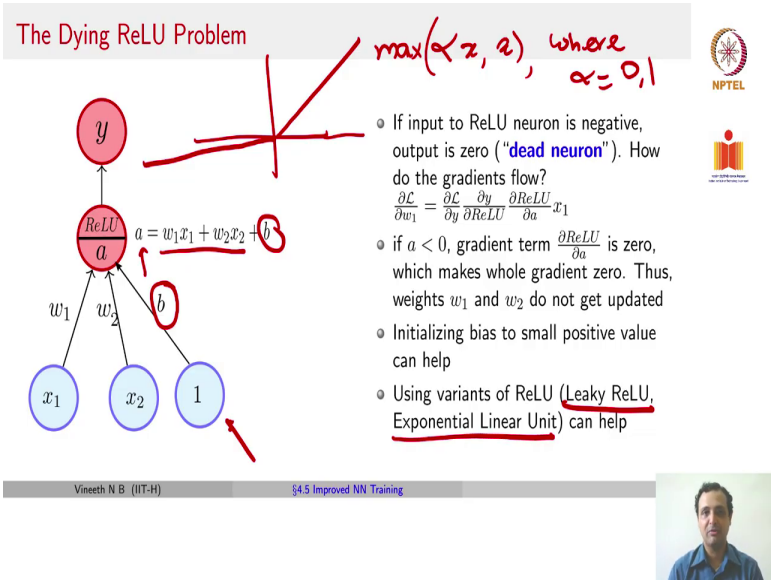


But ReLU has one problem which is known as the dying neuron or a dead neuron problem, if the input to a ReLU neuron is negative, the output would be 0. Now, let us try to understand how the gradients would flow in this kind of a situation. Let us assume now that a previous layer's inputs were given this way with inputs x1, x2, weights w1, w2, a is the pre-activation of that neuron, you then apply ReLU activation and you get the output y.

If you want to now compute $\partial L/\partial w_1$ by chain rule, you have $\partial L/\partial y * \partial y/\partial ReLU * \partial ReLU/\partial a * x_1$. Because the input a is negative $\partial ReLU/\partial a$ would straight away become 0 and none of these weights would get any contribution through the ReLU activation, which means those weights w_1, w_2 may never get updated, they stay the same and this cycle could continue.

For example, if w_1 and w_2 were negative which means in every cycle it is very likely that a will keep getting a negative value and because of ReLU there will be no gradient that comes in, w_1 and w_2 will stay negative and this cycle keeps on going and the neuron never participates in the neural networks output, this is known as the dead neuron problem. And it has been observed through empirical studies that for a neural network that has a ReLU activation in all its layers, many neurons end up suffering from the dying neuron problem.

(Refer Slide Time: 24:52)

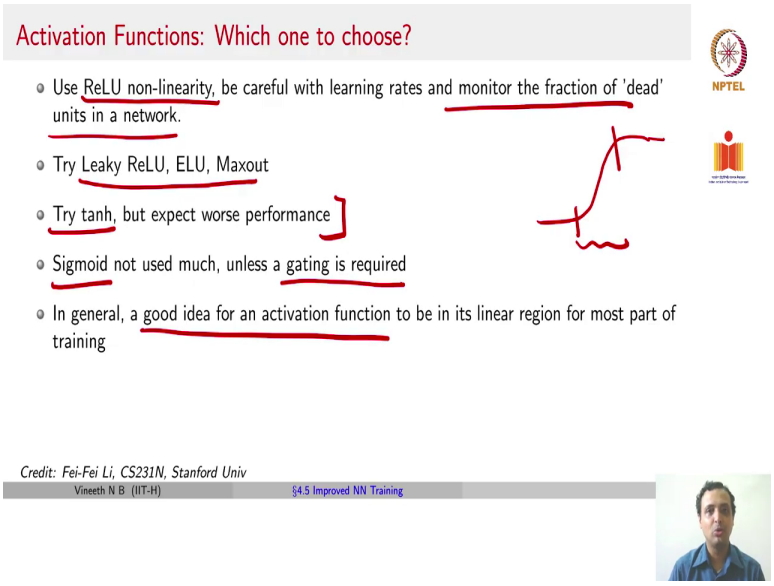


How do you overcome this? A couple of ways, one is you could initialize your bias weight to a positive value. How does this help? In case $w_1x_1 + w_2x_2$ turns out to be negative, having a positive bias b here may push a towards the non-negative side and this could help ReLU get activated and the gradients start flowing through this particular neuron.

Other options are use Leaky ReLU, now you should be able to appreciate why Leaky ReLU was useful, if you go back and see the shape of Leaky ReLU we said that it is very similar to ReLU on the positive side but the negative side instead of making it 0 just let a small value go through and that is why it was max of αx, where α could be a small value such as 0.1.

This lets the output of the ReLU not become a 0 which may kill the gradient but let a small gradient at least pass through the neuron, so it does not stop contributing to the neural network at least contributes marginally. Exponential Linear Unit or ELU also has the same effect.

(Refer Slide Time: 26:20)



Now, you could ask, having seen these, which one do I choose when I design a neural network. In practice most often people use ReLU, it just seems to work very well, very cheap, very fast, so people often use this, but it may be handy to keep monitoring the fraction of dead units in a neural network, neurons that are not contributing and see if that requires you to change something in your design.

In case ReLu does not work, consider using Leaky ReLU or ELU or Maxout. $tanh$ can be tried but you may have to expect a performance which is not as good as ReLU depends on a certain circumstance. Sigmoid is not used much today unless a gating kind of a response is required, we will see such networks later in this course where you want a neuron to have a gating effect, rather have an output line between 0 and 1, in those cases including an output neuron for example in a binary classification problem.

In those scenarios sigmoid neuron is the default choice, $tanh$ is also used in certain such settings where you want the output of the neuron to be bound in a certain range. In general, a takeaway here is you would want an activation function to be in its linear region for as much part of the training as possible.

What does that mean? So, if you take the sigmoid activation function, you would want to ensure the activation function is most often used when the inputs are lying in this range. So, you could modify your sigmoid activation function, you could modify certain coefficients, this is sigmoid activation function to make this linear region a little wider instead of lying between specific values you could make it lie between a larger range if you knew that is how the inputs are going to be in your neural network.

You can do such things too to improve the performance. In fact, some of these suggestions here are given in a very nice reference which I will point you to at the end of this lecture.