

**Deep Learning for Computer Vision**  
**Professor Vineeth N Balasubramanian**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Hyderabad**

**Lecture 23**

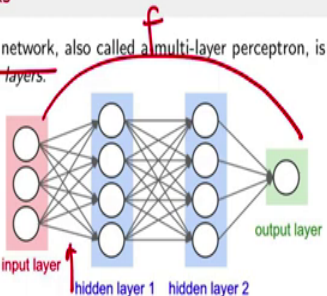
**Feedforward Neural Networks and Backpropagation - Part 1**

Our next lecture will be on continuing the discussion on multilayer perceptrons and knowing how we can train them. So for perceptrons, we talked about the perceptron learning algorithm. Then for multilayer perceptrons how do you train them with the same method work is what we will try to find out now.

(Refer Slide Time: 00:45)

**Feedforward Networks**




- A feedforward neural network, also called a multi-layer perceptron, is a collection of neurons, organized in layers.



input layer    hidden layer 1    hidden layer 2    output layer

- It is used to approximate some function  $f^*$ . For instance,  $f^*$  could be a classifier that maps an input vector  $x$  to a category  $y$ .
- The neurons are arranged in the form of a directed acyclic graph i.e., the information only flows in one direction - input  $x$  to output  $y$ . Hence the term **feedforward**.

Vineeth N B (IIT-H)    14.2 Backpropagation



Multilayer perceptrons are also known as feedforward neural networks because the information is fed forward from the input all the way to the layers to be output. And all neurons are organized in layers. While we saw examples of multilayer perceptrons with one hidden layer in the previous lecture you could have as many hidden layers as you need between the input layer and the output layer. Obviously, at the end of the day a neural network, a feedforward neural network or a multilayer perceptron is approximating a function that takes you from input to output.

So whatever function you are approximating can be better approximated perhaps if you have more hidden layers if that function is complex, this is not a necessity but having more hidden

layers could help you. We will see later how adding too many layers can also cause problems but at this point let us move forward with this discussion of training a feedforward neural network.

So, as I just mentioned a neural network is typically used to approximate some function  $f^*$ .  $f^*$  is the ideal function that in machine learning given an input can assign the correct label so that particular input, so  $f^*$  could just be a classifier also and to the neural network we try to approximate this function  $f^*$ .

The neurons are arranged in the form of a directed acyclic graph, directed because of these edges from one layer to the next layer are directed edges you cannot information does not flow the other way, although for training you use them we will see that in a moment but when you propagate information you will propagate them only in the forward direction that is the directed nature of this graph.



And it is acyclic because there are no cycles in this particular graph. The information flows only in one direction from the input all the way to the output and that is why they are also known as feedforward networks.


(Refer Slide Time: 03:20)

### Feedforward Networks

- The number of layers in the network (excluding the input layer) is known as **depth**

- Each neuron can be seen as a **vector-to-scalar** function which takes a vector of inputs from the previous layer and computes a scalar value.
- Above network can be seen as a composition of functions  $y = f^{(3)}(f^{(2)}(f^{(1)}(x)))$ ,  $f^{(1)}$  being the first hidden layer,  $f^{(2)}$  being the second and  $f^{(3)}$  being the final output layer.

Venketh N B. (IIT-H)
4.2 Backpropagation


The number of layers in the neural network is typically known as depth; each neuron can be seen as a vector to scalar function and which takes a better of inputs from the previous layer and

computes a scalar value. Remember, even a single perceptron is a vector scalar function because it has  $n$  different dimensions of  $x$  as input and gives us one output each neuron is similar to a perceptron in the assets.

And when you have multiple layers stacked one after the other in a neural network you could look at the entire network as a composition of several functions, each layer is a certain function which shakes you.

So let us assume that you have layer 1 which is denoted by function  $f_1$ , this function takes you from  $R^{d_i} \rightarrow R^{h_1}$  this is what the function  $f_1$  does. Similarly, function  $f_2$  is a function that goes  $R^{h_1} \rightarrow R^{h_2}$

Similarly,  $f_3$  goes from  $R^{h_2} \rightarrow R^{d_o}$ . So each layer is a function by itself and the overall neural network can be envisioned as a composition of functions that achieves the purpose you are looking for.

(Refer Slide Time: 05:21)

The slide is titled "Feedforward Networks" in red text. It contains four bullet points: 1. "To approximate some function  $f^*$ , we are generally given noisy estimates of  $f^*(x)$  at different points, in the form of a dataset  $\{x_i, y_i\}_{i=1}^M$ ". 2. "Our neural network defines a function  $y = f(x; \theta)$ . Our goal is to learn the parameters (weights and biases)  $\theta$  such that  $f$  best approximates  $f^*$ ". 3. "How to find the values of the parameters i.e., train the network?". 4. "In this lecture, we introduce **Gradient Descent**, the go-to method to train neural networks". On the right side of the slide, there are two logos: the NPTEL logo (National Programme on Technology Enhanced Learning) and the IIT Bombay logo. At the bottom left, it says "Vineth N.B. (IIT-H)" and "4.2 Backpropagation". At the bottom right, there is a small video feed of a man speaking.

So in machine learning we all know that we are given training data to train an algorithm and if you consider the supervised learning settings, then you have data as well as labels provided to you in your training data.

Now to approximate some function  $f^*$ , we are generally given noisy estimates of  $f^*$  at different points in the form of a data set,  $(x_i, y_i)$ ,  $x_i$  is the vector that could have certain dimension,  $y_i$  is the output label corresponding to that particular input data point  $x_i$  and there are a total of  $m$  such examples in your data set, you could assume that these examples in your training data set are instances of the overall function  $f^*$  that takes you from  $x$  to  $y$ .

So our neural network defines a function  $y$  which is  $f(x, \theta)$ ,  $x$  is the input, by  $\theta$ , we are going to refer to all the weights and biases that we have in the neural network. We are going to henceforth call  $\theta$  the weights and biases of the neural network. These are the  $\theta$  other values that parameterize the neural network that is what defines the function output that you are going to get when you propagate a value through our neural network.

Our goal is to somehow find a way that  $f$  can best approximate  $f^*$ , once again let me clarify what  $f^*$  is.  $f^*$  is a hypothetical function which takes you from input to networks, so each of your training data points are instances of that function and our goal now is and we do not know what  $f^*$  actually is in machine learning typically.

If you knew  $f^*$ , you actually do not need machine learning. When new data points come in you simply apply  $f^*$  on it and you will get the label that you need. In machine learning we are only given those noisy instances of  $f^*$  but we do not know what  $f^*$  is and that is what defines the field of machine learning itself.

And our goal is to use the neural network to best approximate  $f^*$ . Now the question that we have is how do you find the values of the parameter  $\theta$ , the weights and biases to train this network? So you are given a training data set, let us assume you are given a neural network of a certain architecture, you are given say two hidden layers, the first hidden layer has 10 neurons, the second hidden layer has 100 neurons whatever that be that is user defined.

That is given to you and the training data set has given to you your goal is to find what should be your weights of the neural network which will do well on the training data. So that is the process of training neural networks and to do that we introduce a very well-known method optimization

known as Gradient Descent. A gradient descent is very simple but a very well-studied method in optimization which is used to minimize objective functions in general. And that is what we are going to use to train feedforward neural networks.

(Refer Slide Time: 09:06)

### Gradient Descent: A 1D Example

- Neural networks are usually trained by minimizing a loss function, such as mean square error.

$$Loss_{MSE} = \frac{1}{M} \sum_{m=1}^M f^*(x) - f(x; \theta)^2$$

*(Handwritten notes:  $\theta \rightarrow$  random)*

- Let us consider a simple 1D example, where we try to minimize the function  $f(x) = x^2$ . Specifically, we find out the value  $x^*$  gives the smallest value for  $f(x)$  i.e.,  $f(x^*)$ .

$$x^* = \underset{x}{\operatorname{argmin}} f(x)$$

Vineth N B (IIT-H)

4.2 Backpropagation



Let us take a simple example and then take it forward neural networks. Neural networks are usually trained by minimizing a loss function such as mean squared error. The neural network when you give an input to a neural network it uses a certain output. What is that output?  $f(x; \theta)$  Once again if you take the patient example if you gave a set of patient attributes to a neural network assuming that you initialize this  $\theta$  to some random values, some random values to start the neural network with.

If you now pass one patient's information such as blood group A is well person smokes or not so and so forth, all of that information lets us say you passed it to a neural network you get a certain output on the output label based on the weights that you have initialized. That value that you get is defined by  $f(x; \theta)$  but in your training data you already know what  $x$  should give you because that is why it is called training data.

You have the correct labels given to you so you have at least  $f^*(x)$  define for that particular value of  $x$  you may not know what  $f^*$  is at other places but you know what  $f^*$  is at that particular location. I knew some of this error up over all your input data points  $m$  is the total number of data points and your average error across all of your data points.

This is typically known as the mean squared error is the mean of the square errors for all your training data samples. Let us take a simple one day example and then try to study this subject

further. Let us say we like to minimize the function  $f(x) = x^2$  specifically let us assume that we know the value  $f^*$  which gives you the smallest value for  $f(x)$ .

Let us assume that  $f^*(x)$ . Of course, we know that  $x^* = \operatorname{argmin}_x f(x)$ ,  $\operatorname{argmin}_x$  gives you the minimum value of  $f(x)$  across all the possible values which you can choose for  $x$ .  $\operatorname{argmin}_x$  is the value of  $x$  that gives you the minimum  $f(x)$ . That is the difference between  $\min$  and  $\operatorname{argmin}$

So  $x^*$  is that  $x$  that gives you the minimum value of  $f(x)$ . So to summarize this slide so you have mean squared error which we typically use to train neural networks but before we deal with that mean squared error loss let us just consider any function  $f(x)$  which we want to minimize, let us try to see for the 1D case.

(Refer Slide Time: 12:14)

### Gradient Descent: A 1D Example

- We can obtain the slope of the function  $f(x)$  at  $x$  by taking its derivative i.e.,  $f'(x)$ .
- This means, if we give a very small push to  $x$  in the direction (sign) of the slope, we're sure that the function will increase.

$$f(x + p \cdot \operatorname{sign}(f'(x))) > f(x) \text{ for an infinitesimally small } p$$



NPTEL



## Gradient Descent: A 1D Example

- We can obtain the slope of the function  $f(x)$  at  $x$  by taking its derivative i.e.,  $f'(x)$ .
- This means, if we give a very small *push* to  $x$  in the direction (sign) of the slope, we're sure that the function will increase.

$$f(x + p \cdot \text{sign}(f'(x))) > f(x) \text{ for an infinitesimally small } p$$

- The reverse is also true i.e.,

$$f(x - p \cdot \text{sign}(f'(x))) < f(x) \text{ for an infinitesimally small } p$$

- This forms the basis for gradient descent - we start off at a random  $x$ , and take small steps in the direction of the **negative** gradient.



Vineth N B (IIT-H) 14.2 Backpropagation



So what gradient descent suggests to us is that given the function  $f(x)$  we can obtain the slope of the function  $f(x)$  at  $x$  by taking its derivative, let  $f'(x)$  denote that slope of  $f$  at  $x$ . Now if you give a very small push to  $x$  in the direction of the slope the function will increase, what we mean is you have  $x$ , you have  $p \cdot \text{sign}(f'(x))$  which means we take the sign of the gradient and whatever the direction be, in that direction you take a small  $p$  step forward.

So that is going to give you a function value which will be greater than  $f(x)$ . You could now ask me the question: what if the sign itself was negative but  $f(x + p \cdot \text{sign}(f'(x))) > f(x)$  of course yes, because that is how the gradient is defined. So if your sign is negative the reverse is also true which means if you take one step in the negative direction of the gradient your function value will become lesser than the function value that you had at  $x$ .

That should give us a clue which means if I take the gradient at a particular point  $x$  and I go one step in the negative direction of the gradient I am going to find another point  $x$  where the function value will be lower which means if I keep repeating this over and over and over again I will finally reach a point where the function value is released. This is the basic idea of gradient descent. We start off at a random  $x$  and keep taking small steps in the direction of the negative gradient and we iteratively reach a point at which  $f(x)$  is optimum.



(Refer Slide Time: 14:31)

### Gradient Descent: A 1D Example

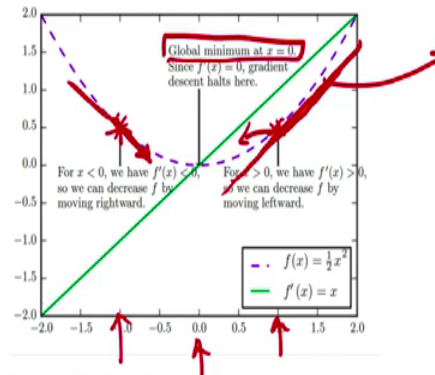


Figure Credit: Goodfellow et al, DL Book Ch 4

Vineth N B (IIT-H)

14.2 Backpropagation



NPTEL



With support from the Ministry of Education, Government of India

Here is an illustration of what we just spoke about. Let us assume that  $f$  was indeed a convex function that is your  $f$  and the minimum is attained at  $x$  is equal to 0. Since the minimum is attained here we all know that which means that particular point is a critical point which means  $f' = 0$  at that particular point.

So let us consider a certain  $x$ , say its minus 1 which is this particular point, we can see that at that particular point the gradient is going to be negative. If the gradient is negative we take one step in the direction of the negative gradient which means we grow one step towards the positive side or rightward and you will take one step in the rightward direction which will definitely take you to a function value which is lowered.

On the other hand if your current  $x$  was here at 1 let us say at this particular point your gradient is, your gradient of the tangent is positive, you are still going to take a step in the negative direction on the gradient which means you will go left when you increment this algorithm which again will take you to a point which is where the function value is lesser.

As you can see it is a fairly simple method but given a function this is a simple method that can help you reach the minimum of that function and find the  $x$  at which you can reach that min. How is this connected to neural networks? Will come to that in a moment.

(Refer Slide Time: 16:12)

### Why Negative Gradient?

- Consider the multivariate case, since while training neural networks, the loss function we minimize is parametrized by multiple weights,  $\theta$
- For simplicity, we denote our loss function as  $L(\theta)$ . Our aim is to find the weight vector  $\theta$  which minimizes  $L(\theta)$



NPTEL



NPTEL

Vineth N B (IIT-H)

4.2 Backpropagation



Let us consider now a slightly more complex setting, a multivariate setting as shown in the previous slides we consider  $f(x)$  which is univariate. Let us consider a multivariate setting where while training neural networks the loss function we minimize is parameterized by many weights of the neural network. Let us subsume all of them into a quantity known as  $\theta$  or the weights and the biases.

You have to subsume into one quantity known as  $\theta$ . Let us now denote this loss function as  $L$  of  $\theta$  and our aim is to find the weight vector  $\theta$  which minimizes  $L(\theta)$ , very similar to what we saw on the earlier slide.

(Refer Slide Time: 16:56)

### Why Negative Gradient?

- Consider the multivariate case, since while training neural networks, the loss function we minimize is parametrized by multiple weights,  $\theta$
- For simplicity, we denote our loss function as  $L(\theta)$ . Our aim is to find the weight vector  $\theta$  which minimizes  $L(\theta)$
- Let  $u$ , a unit vector, be the direction that takes us to the minimum, i.e.:

$$\min_{u, u^T u = 1} u^T \nabla_{\theta} L(\theta)$$



Vineth N B. (IIT-H)

4.2 Backpropagation



Now, let  $u$  be a unit vector which is the direction that takes us to the minimum of this loss function, rather we are saying now that we have the gradient  $\nabla_{\theta} L(\theta)$  and we know that some component of the gradient may help us go to the minimum of this loss function that we want to minimize.

Mean squared error was one such loss function there could be at this point we are saying that mean squared error may not be the only loss function for a neural network there could be other loss functions that you minimize the primary test with respect to let us just call that  $L(\theta)$  at the gradient be  $\nabla_{\theta} L(\theta)$

And let  $u$  be the vector which is concurrent to the gradient that will take you to the minimum and for simplicity you are going to assume that  $u$  is a unit vector. So we want to minimize over all

possible unit vectors  $\min_{u, u^T u = 1} u^T \nabla_{\theta} L(\theta)$

(Refer Slide Time: 18:03)

### Why Negative Gradient?

- Consider the multivariate case, since while training neural networks, the loss function we minimize is parametrized by multiple weights,  $\theta$
- For simplicity, we denote our loss function as  $L(\theta)$ . Our aim is to find the weight vector  $\theta$  which minimizes  $L(\theta)$
- Let  $u$ , a unit vector, be the direction that takes us to the minimum, i.e.:

$$\min_{\|u\|_2=1} u^T \nabla_{\theta} L(\theta)$$

$$= \min_{\|u\|_2=1} \|u\|_2 \|\nabla_{\theta} L(\theta)\|_2 \cos \beta$$

+ve  $\sqrt{L_1(\theta)^2 + L_2(\theta)^2 + \dots + L_n(\theta)^2}$

- Since  $\|u\|_2 = 1$ , we can minimize the above function when  $\beta = 180^\circ$ , i.e. when  $u$  is the direction of negative gradient

Vineth N B. (IIT-H)

14.2 Backpropagation



Now this can be written as min be simple this is simply a dot product, so it can be written as  $\|u\|_2 \|\nabla_{\theta} L(\theta)\|_2 \cos \beta$  where  $\beta$  is the angle between the  $u$  vector and the gradient vector, gradient of the loss with respect to theta by 2. Clearly we know the 2-norm is a positive product, this quantity here is a positive quantity. It is number to a norm in standard definition is simply square root of L1 theta square plus L2 theta square so on and so forth depending on how many components it has whatever components it has.

This is your standard true norm definition and that quantity 2-norm is positive a non-negative quantity. So which means we ideally want to minimize this dot product because  $u$  is a unit vector, the only way we can minimize this quantity is to make  $\cos \beta = -1$ . You remember 2-norm of  $u$ , the value is 1 because  $u$  is a unit vector, we already know that.

So the only way to minimize this entire quantity is to make a  $\cos \beta$  to be as low as possible and the least value of the  $\cos \beta$  is minus 1 which means since  $u$  is 1,  $u$  has to be the direction of the negative gradient. So that is the vector that will actually minimize this particular quantity. Remember for  $\beta$  to be 180 degrees  $u$  has to be because  $\beta$  is simply the angle between  $u$  and the gradient.

$u$  simply has to be in the opposite direction of the gradient which will minimize this quantity rather if we want to use the gradient in some way to reach the minimum of the function of a

function we have to go the opposite direction of the gradient to reach that minimum. Keep in mind that this is the gradient descent algorithm. Keep in mind that it has a complement known as gradient ascent where if you go in the positive direction of the gradient you will reach a maximum. That algorithm also exists but in this context we are interested in minimizing a loss function and hence we are going to focus on gradient descent.

(Refer Slide Time: 20:34)



### How to use Gradient Descent


We can use Gradient Descent to train neural networks as follows:

- Start with a random weight vector  $\theta$ .
- Compute the loss function over the dataset, i.e.,  $L(\theta)$  with the current network, using a suitable loss function such as mean-squared error
- Compute the gradients of the loss function with respect to each weight value  $\frac{\delta L}{\delta \theta_i}$ .
- Update the weights as follows, where  $\eta$  is the learning rate i.e., the amount by which the weight is changed in each step:

$$\theta_i^{next} = \theta_i^{curr} - \eta \frac{\delta L}{\delta \theta_i}$$

We can repeat the above steps until the gradient is zero.

Vineth N B (IIT-H)
4.2 Backpropagation


Let us now try to see how you actually use gradient descent in practice to train neural networks or train any other function for that matter. For neural networks in particular we start with a random weight vector  $\theta$ , we compute the loss function over the data set which you have a training data set, you take a  $L(\theta)$  with the current network using a loss, suitable loss function such as mean squared error.

You could have other loss functions and we will see many of them over this course but at this point let us take one of the simplest which is the mean squared error. We compute the gradient of the loss function with respect to each parameter in the network, it could be the way, it could be the bias, could be any other parameter.

Every parameter that is important in getting the output of the neural network you compute the gradient of the loss function every parameter that you need to learn ofcourse. You compute the gradient of the loss function with respect to each of those weight values and we denote that as

$\partial L/\partial \theta$  right. So now based on gradient descent which is an iterative procedure you define this actual step to be given your which is one of your weights current value.

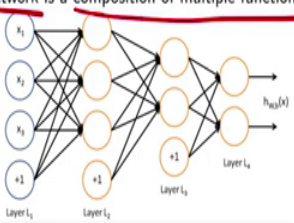
You take  $\partial L/\partial \theta$  which is the gradient of the partial derivative of the loss with respect to that particular parameter  $\theta^i$ , if there are many weights and biases on the neural network you have to compute the derivative of the loss function with respect to every weight or every bias neural network. And  $\theta_i^{next} = \theta_i^{current} - \eta \partial L/\partial \theta_i^{current}$  where  $\eta$  is simply known as the learning rate or a step size.

It tells us how large a step you want to take the direction we will talk about ways in which this can be chosen a little later. I am repeating this process until the gradient is 0, why? Because when the gradient is 0 we have reached a critical point which is perhaps the minimum of that function.



(Refer Slide Time: 22:45)

### Gradient Descent

- A feedforward neural network is a composition of multiple functions, organized as layers

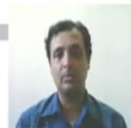


- What do we need to implement gradient descent? Compute gradient of loss function w.r.t. each weight in the network. How to do this?

Vineth N.B. (IIT-H)

4.2 Backpropagation



Let us look at a neural network again as we just said sometime ago a feedforward neural network is a composition of multiple functions organized as layers. Now how do we implement gradient descent in this kind of neural network? We know we have to compute the gradient of the loss function. Let us assume the loss function is given to us, let us assume it is mean squared error. We have to compute the gradient of the loss function with respect to every weight in the neural network. How do we do this?

(Refer Slide Time: 23:17)

**Gradient Descent**

- A feedforward neural network is a composition of multiple functions, organized as layers

$L(\theta)$   
 $\frac{\partial L(\theta)}{\partial w_i}$   
 $\theta: \{w, b\}$

- What do we need to implement gradient descent? Compute gradient of loss function w.r.t. each weight in the network. How to do this?
- Using the **chain rule in calculus**
  - Computing gradient w.r.t. to a weight  $w_i$  requires computation of gradients with respect to outputs which involve that weight i.e., all activations from layer  $i + 1$  to last layer,  $n_l$

Vineth N B. (IIT-H) 14.2 Backpropagation

We do this by taking advantage of the chain rule in calculus. Computing gradients with respect to any weight in a layer  $i$  requires the computation of the gradient with respect to outputs which involve the weight in every layer from that layer to the output.

So for example if you had a certain weight  $w_i$  here remember the loss function is defined at this particular point loss of let us say  $\theta$  where  $\theta$  is nothing but a set with all the weights and biases. So we want to find  $\partial L(\theta)/\partial w_i$  where  $i$  is one of the weights in the neural network.

So to do that remember we need to apply chain rule which means we have to now find out  $\partial L/\partial \theta$  by computing partial derivatives of all the steps in which  $w_i$  equal to which  $w_i$  contributed across all of the weights, for example  $w_i$  probably contributes to every value because this neuron contributes to every other neuron in that next layer and so on so forth. So we now have to sum all of those contributions and then find the gradient of the loss function with respect to this  $w_i$ . That is the overall intuition but let us say let us see how we actually do it as an algorithm.