Distributed Optimization and Machine Learning

Prof. Mayank Baranwal

Computer Science & Engineering, Electrical Engineering, Mathematics

Indian Institute of Technology Bombay

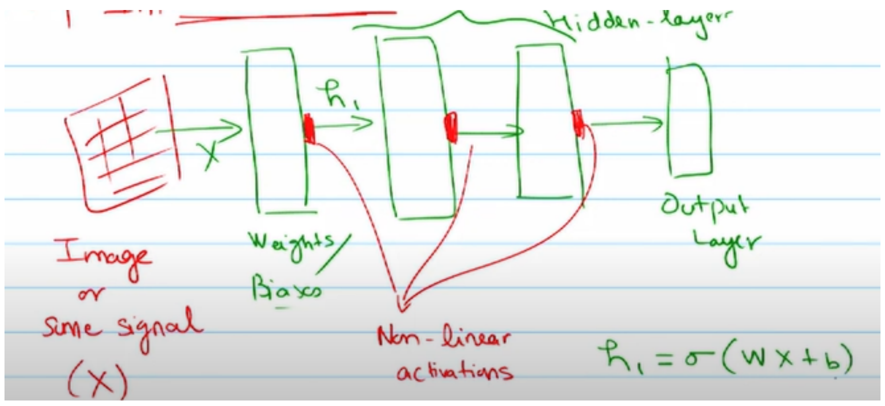
Week-10

Lecture 38: Introduction to Neural Networks

So the focus for today's lecture is going to be on large scale machine learning. Essentially when we perform gradient descent on the weights of the neural networks let's say. So what kind of considerations that we need to keep in mind, be it in terms of communication and let's say you have large, I mean abundance of data and there are multiple agents in the network and they have their own private data. So what kind of algorithms can be employed to ensure a decentralized learning, essentially, or a distributed learning, but at the same time, how do we sort of alleviate the communication bandwidth constraints that we may have? Eventually, let's say when we are trying to, even to our neighbors, when we are trying to relay certain information about the weights, or maybe the gradients, right? if the if the number of parameters in the network are in the order of a few millions or even nowadays even like with larger i mean large language models they're in the order of billions right so if you're going to be extending that many parameters with your neighbors I mean that entails a huge communication bandwidth requirement uh like for inter-agent communication right and in that case you would want to ensure that I mean if you have a centralized server that is communicating with all the likely multiple other agents They cannot expect to, I mean, let's say there are 10 agents, so it will be 10 times of that information, right? So that's a huge amount of communication bandwidth requirement. So how do we design communication architectures and algorithms around those communication architectures? The underlying algorithm is still being the gradient descent, but how do we sort of decentralize it is going to be the focus for today's lecture.

And to start with, we are briefly going to review what neural networks are and why, I mean, just to motivate the problem of large scale machine learning. And then we would look at the algorithmic aspect of it. So a very brief introduction to neural networks. So let's say you have some input data it can be an image or some other signal right it can be a time series signal or some other features right.

So the goal, so what do neural networks do? So for some input x what does it do? Let's say we are looking at supervised supervised learning problem right. So, we are going to predict some label y right and let us say if it is an image then we are trying to predict maybe the object in that image if it is an image of a cat or a dog or a tiger and so on right. So, what does a simple feed-forward neural network architecture looks like? So, you have this input x going in then what do you have? Weights. So, there will be some weights also biases Yeah, so some non-linear activation right. So, why do we need non-linear activation? In bearing will be redundant right, because if there are no non-linear activation then you can multiply all the weights together and then you can represent output as a function of input.



I mean you don't need multiple stacks of weights. So the reason we introduce multiple weight stacks or these layers is because after each layer we want to add little bit of non-linearity. So that you are making this entire model more expressive. Otherwise it would just be a linear model no matter how deep you make your neural network right. So there is let's say a non-linear activation.

So, think of these as non-inner activations and then there is an output layer right So, these are called hidden layers. and then you have an output layer. In certain text they do not specifically mention biases. So, let us say the input here is x right and the output of this layer is let us call it h 1. So, how is h 1 related to x? h 1 is.

Yes, some sigma of. W x plus b right and w what is the size let us say h is in R m and x and x is in R n, then what is the size of w? m cross n. m cross n right. So, w would be in m cross n and what about the biased term? It is in R m dimensional vector right. So, how many parameters are there like for just simple this thing? Yeah, m into n plus 1 parameter.

Thus in certain text you do not see biases coming in and why? Like we will just be talking about weights and not specifically about biases because you can also define a new input x tilde as x n 1 and then if you multiply this by this new weight vector which is m cross n plus 1, the last entry is like a bias term right. So, I mean that is why some in certain text if they do not specifically talk about biases they will just talk about the weights of the neural network ok. So, you the idea is you stack these multiple layers together and then there will be an output layer. So, if it is a classification problem what kind of output layer functions that you can think of? Softmax. Softmax right.

for the output layer if it's a let's say regression problem. So, for a given x you would want to predict y a scalar y then the size of the output layer like the output the dimension of the output layer would be just like a whatever input is coming in cross one right. So, output layer like usually let's say this dimension is the input dimension here h 3 is in R p. So, usually of size p cross 1 for regression problems and let us say if you have a k fold classification problem. So, if you have k fold classification problem that means are k possible classes and you are trying to predict the correct class out of those k classes.

Then you have p cross k as the size of the output layer right ok. p is this particular dimension the hidden feature the dimension of the hidden feature that comes to the output layer. just like this or k cross p is what you want to call it maybe I think that is that may have been the confusion. So, k is like when I talk about this weight matrix right output is a layer is supposed to be k dimensional in this case if it is a k fold classes likewise you can write it as 1 cross v that is confusing. ok because for a simple scalar regression problem, you will have just one-dimensional output.

Now for this, so the values of this particular thing, so ideally for an input image you are going to be predicting the probabilities of like that a particular image belongs to a particular class right. So you are going to be predicting k fold probabilities. So probability of it being a cat, it being a dog and so on right. So let us say there are k such classes they have k classes and the output you expect the output to be of this form right. So, it will be let us say 0.

1, 0.4, 0, 0.3, maybe 0.1 and so on right. So, that first of all the sum of I mean it is a valid probability vector. So, the sum should be 1 and the correct class is supposed to be the one with I mean or at least the predicted class is supposed to be the one with the largest entry. So, in this case let us say 0.4. ok. But how do you convert this output let us say I mean right now there are if I look at the output layer let us say let us call it h 4. So, h 4 is simply W times h 3 let us say bias I mean we have some assumed bias as well W times h 3 right which W is k dimensional. So, you get a k-dimensional vector, but it need not be a probability vector right. So, you convert this into a valid probability vector using using. So, how like right now it is just a some some matrix-vector product of with W and h 3 right.

So, it needs to be converted to a valid probability vector using softmax activation. This h 4 or this particular before softmax activation this particular output this particular quantity is called logits or logits depending ok. And you convert these logits into softmax into valid probabilities using softmax activation. So, the probability of the ith class would h 4 i here right and this converts it into a valid probability vector. So, now, summation y i is going to be 1 and that is what you want right you want summation y i is to be equal to 1.

So, when computing this by the way just an side thing when computing this softmax activation let us say h 4 turns out to be a number maybe 100. Now, e raise to the something 100 is going to be enormously huge. So, how do we compute this? Like if you want to make it faster this particular calculation. Yeah, but then just see, but maybe there may be some other entry with which is like 200. So, even if you divided by e raise to the 100 then there will be an entry with I mean for this particular class it will be not necessarily right it is w times h 3 right and it can give you a very large positive number as well from smaller numbers I mean you are fine in fact it is almost going to be 0 right.

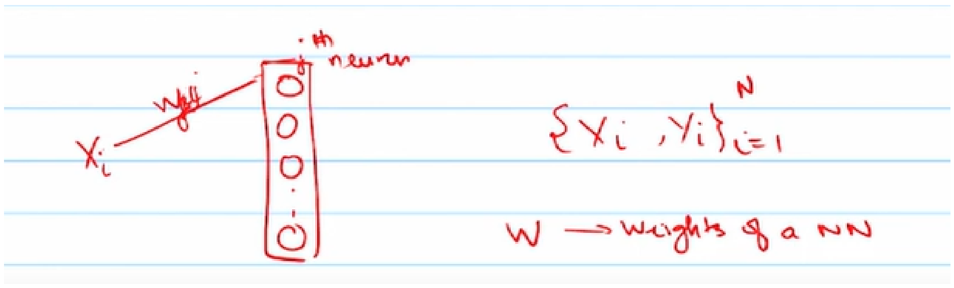
So, you do not care about computing exponential of large highly negative numbers, but exponential like something e raise to the something positive number a large positive number that is very difficult to compute. especially I mean if it is if number is large enough right. So, it is I mean both in terms of storing the number as well as computing it is going to be challenging. So, the trick here is that you find the max of h 4 i's overall i right and then you basically subtract this from all the logits and then exponentiate it right. So, then in that case the largest entry is going to be 1 and everything else is going to be negative right and that is that is easier that is easiest to compute.

$$\left(h_i - \max_{i\in[k]}\{h_4(i)\}\right) \longrightarrow \text{then take softmax}$$

So, this is what is typically done when actually for I mean in your pytorch or tensorflow instead of directly evaluating these exponents or this softmax they first subtract this and then basically take the softmax. ok and this basically saves a lot in terms of computational effort ok all right. So, you can really think of neural network as some function some non-linear function that takes an input x and gives an output y right. So, y is a complicated function of x and usually denote this by like f subscript theta, the theta can be considered as the weights of the neural network right. So, why are neural networks so powerful? Approximate any function.

Yeah. So, there is this universal approximation result any piecewise continuous function can be approximated arbitrarily close using a 4-layer neural network with ReLU activations. So, this is by Shevinko there is this famous result. So, any piecewise non-linear function continuous function can be approximated arbitrarily closed. So, give me an epsilon. So, I will find a four-layer neural network with non-linear active with basically with ReLU activations and number of neurons that are going to be there in each layer right.

So, and what do you mean by number of neurons? So, when you have an input x coming in. So, you can think of each unit as a neuron here right and essentially it is a strength, it is a connection between this input and this particular neuron which is represented by this particular weight matrix or this particular element w i j kind of thing right. So, if you have an input i like x i. So, this w i j kind of or w j i represents the strength of the connection between this ith input and the jth neuron.



right. So, how many neurons are going to be there? So, that determines the width of the neural network. How deep the neural network is going to be? Basically it is related to the number of layers that we are going to have in the neural network and it turns out again. So, the reason that deep learning is so popular is like if you want to approximate the same function to the same degree of approximation like if you want to approximate the same function to the same epsilon value. if you want to design a neural network with wider with let's say fewer number of layers then you would need exponentially more neurons then maybe having fewer neurons per layer but have making it deep. So that's why making the neural networks deep in fact we can show is useful in terms of the number of parameters that you want to have in your network.

So it would try to in some sense you would have fewer number of parameters to work with or at least Yeah, and if you, let's say if I try to reduce the number of layers, then I would have to have exponentially many neurons in each layer. So that is one of the results. Anyway, so the idea is that, I mean, given an input, you are going to be learning a certain output of it. And how do we do that? So if, let's say I have a bunch of data points, right? I have xi, yi, i ranging from one through n.

I have n such data points. And I want to learn the, again, I mean, you can think of weights and biases as just the weights, right? I want to learn the weights of neural network. Weights of a neural network. So how do we learn that? How do we train a neural network? So, we need something called a loss function right. So, what kind of loss functions can we work with or like some of the common loss function that you may be aware of.

Cross entropy. So, again it this question has like. So, it depends on the kind of problem right like let us say if it is a regression problem. Yeah. So, if then a mean square loss would satisfy would make sense and a mean square loss would be of the form. So, you have a prediction for a given sample let us say x i, you have a prediction y i hat and you have the true value y i and this square is going to be your mean square loss right or the square loss and then you can take the mean over the data samples and that gives you the mean square loss.

$$-\sum_{i=1}^{k} t_i \log p_i$$

What about classification problems? What kind of loss functions Do we work within the context of classification problems? Cross entropy loss, right. And what is the functional form of cross entropy loss? Let us say t i is your target or y i like what whatever. again think of like 1. So, t i's are going to be 1 0 and so on right? If it is going to be 1 if let us say it belongs to let us say it is an image of a cat.

So, if it is a probability vector it is going to have 1 only for the cat element cat true and everywhere else it is going to be 0 minus t i log t i right. So, that is the cross entropy loss i ranging from 1 through for a for particular sample this is the cross entropy loss and then again you take the mean of it. So, you get mean cross entropy loss and. So, you have a loss function let us call it L ok and the loss function depends on what? Your current weights right it is a function of the weights of the neural network and what else does it depend on? Thus the data points on which you are going to be working with right.

So, on which you are trying to optimize. Let us represent this by zeta. So, your loss function L depends on the weights as well as the data points that you are going to be optimizing your or training your neural network on right. So, if I if I were to write. So, how do we then train the neural network? So, we want to minimize this loss function.

So, minimize this loss function. So what are the algorithms that we have looked at for minimization problems? Simple minimization problems what like what is the simplest

$$\min_{x\in\mathbb{R}^n} f(x)$$

$$\boxed{x(k+1) = x(k) - \eta_k \nabla f(x_k)} \longrightarrow \text{Gradient descent}$$

algorithm that comes to your mind? Gradient descent right? So what was gradient descent? Suppose I want to minimize a function f. So again just to briefly recap suppose this is what I want minimize this function f of x. So what does the ith iteration or kth iteration of this looks like? You have xk plus 1 is xk minus some step size eta k times this is your simple gradient descent algorithm. ok.

$$W(k+1) = W(k) - \eta_k \nabla L(W(k), \xi_k)$$

↳ Data points sampled in k^{th} iteration

What is stochastic gradient descent? Yeah, one point or let us say a few set of point here small batch of. I mean technically yeah. So, we are going to be updating the weight of the neural network. So, wk plus 1 is going to be wk minus step size times gradient of loss function defined using the current weights wk and the data points that you are going to be sampling in the kth iteration right zeta k. So, very similar to gradient descent, but the stochasticity comes from the I mean fact that we are choosing the point and depending on what point we are trying to optim like compute the gradient on, we only have a stochastic estimate of the gradient, we do not have the full gradient estimate right and therefore, it says it is called stochastic gradient descent algorithm ok.

Full-batch Gradient descent (GD):

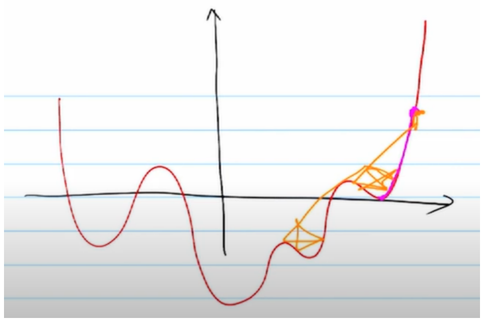
$$\checkmark \quad W(k+1) = W(k) - \eta_k \frac{1}{N} \sum_{i=1}^{N} \nabla l_w(W(k), x_i)$$

I mean in theory would actually I mean the if you look at certain text in particularly in deep learning or machine learning stochastic gradient descent would also correspond to having just one considering one data point at a time and there is another notion called mini-batch gradient descent where you consider a few set of points I mean not the entire data set, but a few set of points on which you try and optimize this right. So, what is the advantages or disadvantages of stochastic gradient descent? So, there is also a full batch gradient descent which is similar to is full batch gradient descent. You can also call it a simple gradient descent algorithm which is fine and the way this works is Wk. Now, here you are going to maybe take the let us say there you have n data points in this in your example nf.

So, this is simply going to be gradient of ok. So, you compute this gradient with respect to let me also include wk so, that is also So, this is you compute the average of the gradient of all the like evaluated on all the data points and then you update the estimates wk plus 1. So, which is better stochastic gradient descent or full batch gradient descent or maybe somewhere in between mini-batch gradient descent, why? So, full batch gradient descent is too slow ok, because you want to because of gradient computation right, because of gradient computation on all data points. What else? So, then why is stochastic gradient descent not good? Okay, so stochastic gradient descent is too stochastic I mean if you just computing gradient with respect to one point on one point right? So, it is going to be too stochastic somewhere in between is going to give you a nice sort of it is going somewhere in midway it is basically it will try and balance out the two aspects. Is there anything else any other advantage of working with like full batch gradient descent of maybe working with mini-batch gradient descent then a full batch gradient descent something else that you can think of in terms of convergence let us say which one mini-batch will converge faster why so then it will converge slowly right.

say this is how a loss landscape looks like. First of all in neural networks because we introduce non-linearities it becomes a non-convex function. Like I mean the loss function is a non-convex function in. So is the square thing, is square loss a convex function or a non-convex function? It is convex, but when we talk about convexity, we also always talk about convexity with respect to what? It is convex in y i hat right, but it is not convex in the parameters of a neural network. So, it is a non-linear function of your parameter like this, this particular loss function is a non-linear function of the parameters of the neural network. So, this is while this is convex in y i hat, it is not convex in the parameters of the neural network and that is why you see the loss landscape to be highly non-convex at something that you must have heard of same with cross-entropy loss.

This is also non-convex in the parameters of the neural network, but otherwise, I mean this is a convex function. So, if you have a non-convex landscape like this right and if I use a full batch gradient descent we expect to be to act smoothly right. So, let us say I start somewhere over here and if I run a full batch gradient descent and it will slowly because gradient computations would be good I mean would be smooth so it will sort of slowly converge towards this that's one thing but it's also smoothly converges to the local optimum. When we add stochastic gradient descent they are going to be lot of so let's say I start here right and I use stochastic gradient descent so at this point I took a sample which was somewhere over here like which basically gave me the gradient value over here here I am going to be bouncing around a lot but because of this randomness there is a possibility that I may get converge to a better minima. So which is not the case with full with full batch gradient descent the chances that you converge like I mean converge to a very good local minima are also going to be somewhat smaller because your convergence is somewhat smoother.



So you are not going to be bouncing around a lot and if you want, I mean the only reason that you can avoid shallow local minima and converge to a better local minima is by basically bouncing around, maybe use a larger step size or keep changing the step size so that you sort of escape that shallow local minima and then converge to a better local minima, right? The chances of this happening with the full batch gradient descent is going to be smaller. With stochastic gradient descent, it's possible that you may converge. to a but then there's too much randomness with stochastic gradient descent so therefore we use a middle path where we actually use some set of data points and that is called a mini-batch gradient descent. So instead of using just one data point we use a few set of data points and then compute the gradient so that it's not as jittery as stochastic gradient descent but it's enough to actually avoid shallow local minima and then obviously there's another thing that you don't need the entire batch be stored and the gradient to be computed. So, that is another computational advantage of mini-batch or these stochastic gradient descent over a full batch gradient descent. Is this clear? Thank you.