00:00:00 So in the next task, before we are ready to run here, it's actually to understand how we train these networks. And this will become the famous backpropagation algorithm. And the backpropagation algorithm is nothing, but a gradient, it's to cast a gradient descent algorithm. But before we can talk about what's about gradient descent, we have to understand what is the cost function. And the basic point here is that the cost function we're working with is kind of connects to normal statistics because it's the...

00:00:30 minus the log likelihood function. And the likelihood function is defined as the probability of the data we have observed. So in this case, we are doing conditional modeling, so we are not concerned about modeling the input features. We only interested in modeling the outputs given the input. So our likelihood function is probability of Y given X. And this probability we model that the deep model, which has some parameters we call those theta. And the kind of philosophy behind the light.

00:01:00 principle is that we say we have seen some data, we want to find the model that maximizes the probability of seeing this data. And of course there's other principles, learning principles, such like Bayesian inference principle, we'll touch a little bit about that throughout the course, but mainly we'll be concerned with maximum likelihood base methods, with regularization, which is like a poor man's version of Bayesian inference. Okay, so how do we actually, for this classification?

00:01:30 problem, the M-ness problem, how do we actually formulate this as a maximum likelihood problem. We can do that by using what is called one-hot encoding. This will lead to the so-called cross entropy likelihood function. And one-hot encoding is this idea that we actually take our class and make that into a sparse vector, which has the same dimensionality as a number of classes. So in this case, with M-ness, we have 10 classes. So we can see if the

00:02:00 output label is 0, then we can represent this with a 10-dimensional vector where we have one at the first position and then 0's of the rest and 1, and 1 label 1 we will represent that with a 0, a 1 and then the rest 0's and so on. And now our output of the network as we already find is this softmax, so that means that we each component of this is the probability for each of these 10 classes. And then we can define in the end, so this is exactly the likelihood of the probability, so we can define the cross into picture.

00:02:30 as minus the log likelihood for such a classification problem. So you can see in the lower part we have the minus sign because we want to make a maximization problem into a minimization problem and we have a sum or the examples because we are taking the logarithm of the likelihood so that we have the log likelihood and then we have the log to the probability and h3 is now our probabilities and our output labels with this passing code.

00:03:00 og måske forde fra Subscripten. I slapen, keskebelel é en fra D forestek этим, så vi sm böyle over largo holds, og vi virker borgere, maknet� og elderken, så har den minus bee BurGe skal piepte satsen som

formede fortægde caresommer Og vi vil også gøre� pode spående eller pensorskøbenrar og med de valgke SH положенде vidre uvolvedale langler.

00:03:30 because the actual implementation will not carry out the sum or K. It's kind of wasteful. We'll only use the one, calculate the one term that we are interested in, to do the classification. Yes, so what I already said is that we use gradient descent based learning. So that means we have some training criteria and for example, the likelihood minus the likelihood and then we compute the gradient. So that means that we take the derivative of our cost function, our scalar cost function,

00:04:00 respect to each of the elements, each of the parameters and models. So if we have end parameters, then we get the gradient as an end dimensional vector. And then our update rule, our basic upgrade rule is to take a small step opposite the gradient. So the gradient, if you think about margin climbing, tells us what is kind of the steepest direction to go. And now we take a step opposite that. So we kind of go down, moves downhill as we can. And then we have a learning rate, ETA, which might depend on which step we have taken, which happens.

00:04:30 we are at so this we index here in this light the iteration number by k and then our learning rate at that iteration is each okay Yes, so let's see how we actually compute these gradients in and deep networks So let's start with the simplest non-trivial networks. So we have a Network which has the has linear outputs in the hidden layer and we have only one hidden layer and we have only one hidden unit and one hidden layer.

00:05:00 put and one up. So it's all scalars and we have only two weights. So we can start by saying, okay, we have a cost function. It would depend on both the weight in the second layer and weight in the first layer. So let's try first to take the derivative with respect to the weight in the second layer. And immediately we see that this, the cost function depends only implicitly on the weights because it depends only explicit, it depends explicitly on the activation.

00:05:30 of the second layer. So we can use the chain rule of differentiation. So that we say we take the first derivative, when we want to take the derivative back to the weight, we take first the derivative, back to the activation in the second layer and then we take the derivative of the weight with respect to this activation. So that gives us this expression here for the derivative. And we can play the same game for the

00:06:00 weight in the first layer and here you can see things get even more complicated because now we actually have to go through two steps of the chain rule. What is very important to notice here is actually that some of the computation we do to get the first derivative with the spectral weights in the single layer and then the first layer are the same. So if we do some bookkeeping we can actually save computation time and this would be more clear if we now go to the general case where we have many hidden units.

00:06:30 in each layer and now we also add this extra layer in our example. So now we want to take derivatives with respect to the weights in all three layers. And each of the layers have now weights that are matrices. So this is just the M-nist example from before. So now for example, if we want to take derivatives with respect to the weights in the last layer, that's the easiest thing because there we know that okay, the output

00:07:00 the cost function only depends on the activations in the last layer and the weights in the third layer depends on this activation through kind of the relo and through the softmax and then this kind of weight times the input from the previous layer. But now if we go to the second layer you can see we have to use chain rule and we get a summation where we have to sum over all the inputs and if we go to the first layer we get both the summation over all the inputs.

00:07:30 and all the units in the second layer. So that means that kind of all the kind of path where the weights in the first layer kind of contribute to the cost in the end. So you can see now that if you put more layers, we'll get more sums and this means that it looks like that actually kind of we cannot go beyond something like this model because we have this kind of exploding complexity where we can't really calculate more and more.

00:08:00 terms kind of the factor of the number of hidden units every time we add a new layer. If you go to next slide where I've kind of highlighted the reuse computation, you can see that actually the same computation, for example, this summation over the outputs is being used in both computing the gradients with spectral weights in the second layer and the first layer. So if we simply store this computation once and for all, then

00:08:30 we can simplify the computation for the gradient in the first layer and we don't have this kind of exploding complexity. So this leads us to define this backpropagation algorithm which is nothing but like a dynamic programming algorithm where we can write it like this and you can see that in general we can write recursion where we say that the derivatives of the activation in of the cost function in one layer is defined in general.

00:09:00 terms of the derivative of the activations in the next layer. So you can see this is why it's called back propagation. So the forward propagation is when we propagate the input to the output, calculate the cost and then the back propagation is through this kind of linear recursion for the gradients. And then the end, when we have done this kind of back propagation in the middle equation, then we can actually compute the actual gradient simply by the last of the rule and the

00:09:30 The important thing here is that we actually have to kind of store a lot of things in the memory and this is usually not a problem if you have something like let's say like here we have three layers of weights but if you think about this for what is called recurrent on network then actually kind of the depth is like the number of time steps and then this can actually be a little bit memory consuming. So we come back to that when we talk about recurrent networks.