Applied Accelerated Artificial Intelligence
Prof. Bharatkumar Sharma
Department of Computer Science and Engineering
Indian Institute of Technology, Palakkad

Lecture - 34
Fundamentals of Distributed AI Computing Session 1 Part 1

(Refer Slide Time: 00:14)

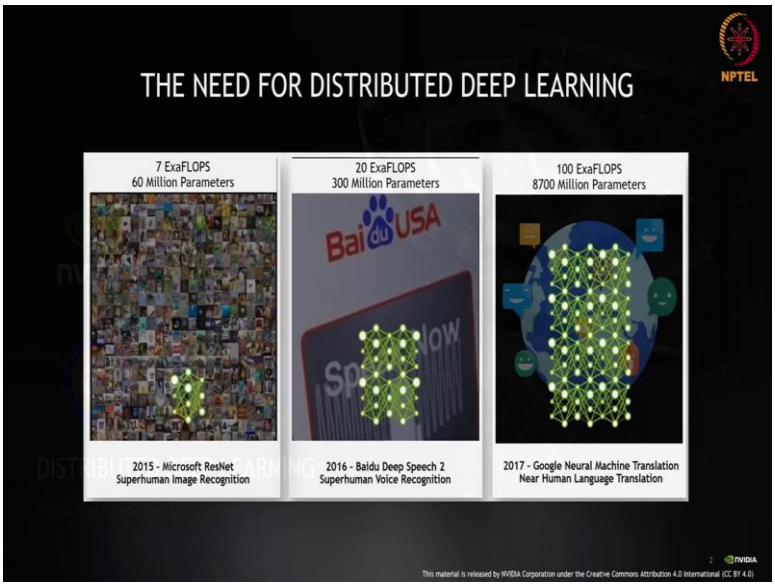
Welcome everyone to the next lecture in the series. My name is Bharat Kumar and I will be the speaker for the today's session. Actually, the next couple of sessions on distributed deep learning, if you look at the previous set of lectures they have been primarily targeted towards a particular approach. In the session 1 you went through how to use frameworks like PyTorch.

TensorFlow and the other ones also you also saw how PyTorch TensorFlow and the other frameworks they basically utilize the GPU. They how do they utilize the GPU and how do they accelerate the computation and make it much more easier for the researchers to do some of their jobs, but the advancement in AI it has become almost impossible to do latest work or the state of the art work without having to distribute or adding one more level of parallelism, which is not limited to only say one GPU or thousands of cores within one GPU.

And we are going to look at the motivation for this in this particular lecture, couple of lectures and then we will also look at how to utilize multiple GPUs. There are different

kinds of strategies how each and every framework utilizes it. So, the next couple of lectures would be dedicated towards advancing you from using a smaller set of problems to a larger set of or solving the grand challenges as we say in this field. So, welcome to this particular lecture.

(Refer Slide Time: 02:21)



And the need for distributed deep learning is there, we have witnessed an AI revolution in the past decade and AI has made its impact in a variety of domains over time. A deep learning workload has also grown in size, which needs a huge amount of compute power to train them. We can see here some of the architectures and the compute power needed to train them, as you know we had talked about some time back how do you measure the performance of a processor.

The performance of a processor is measured by something called as FLOPS, Floating point Operations per Second, when we talk about AI based on whether it is training or whether it is inferencing or with the training itself you could use 16-bit-FLOPS or you could use 32-bit-FLOPS or you could use even 8-bit-FLOPS or have a hybrid approach where you have a combination of 16 plus 32-bit. No matter what FLOPS I talk about if I talk about 2015 itself, when the ImageNet challenge where Microsoft ResNet was one of the winner model there.

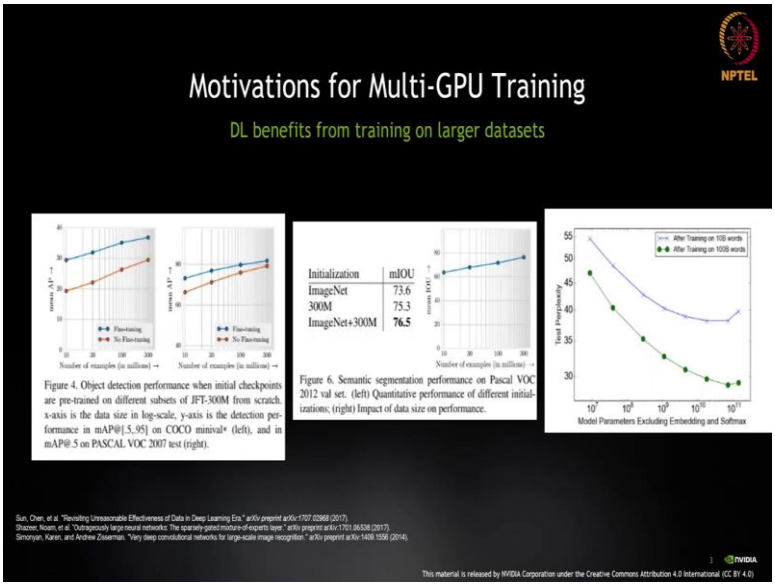
The amount of parameters used by the Microsoft ResNet model was 60 million parameters that were supposed to be trained. In order to train them the amount of

computation which was spent was almost 7 ExaFLOP. When we moved towards 2016 we went from a computer vision problem to a different set of a problem in the space of NLP and all.

In 2016 Baidu Deep Speech basically release the model which had 300 million parameters. Resulting into almost the amount of performance which was required to train this model was in the range of 20 ExaFLOPS. Same 2017 we talk about Google Neural Machine Translation right and we almost the parameters reached a few billion, we are not talking about million, but now billion parameters that needs to be trained.

And we are talking about few examples of the training that was supposed to be done to make all of this work right. Now, look at the amount of parameters that needs to be trained for realistic or real world examples that we want to solve and the amount of computation which is required to do that right.

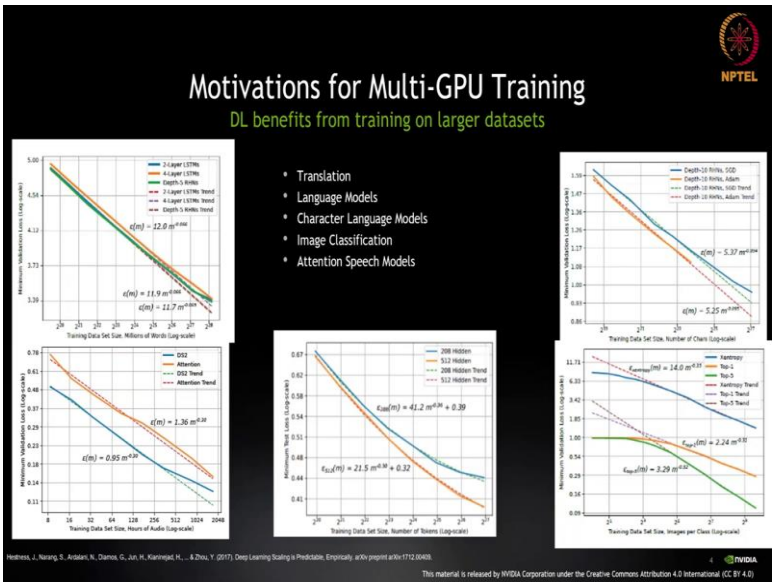
(Refer Slide Time: 05:08)



So, what you see here is the motivation for multi-GPU programming from slightly different. So, if you see here, this started to change in 2017 particularly and this relationship was investigated further by number of research groups. What you are seeing is a different set of papers, the first paper that you see on the left is basically presented here is a result of Google training a set of image classification in neural networks, on the 300 million image data set that we talked about.

And as a result using a recurrent neural network, those initial results suggested that the relationship between the data set size and the accuracy is linear in a logarithmic space. So, what you are seeing here is the relationship of the data set size, which means when the data set size is increasing at the same level the accuracy also kept on increasing. And that is what is kind of highlighted in the paper which is kind of listed here, this paper is from Baidu Research.

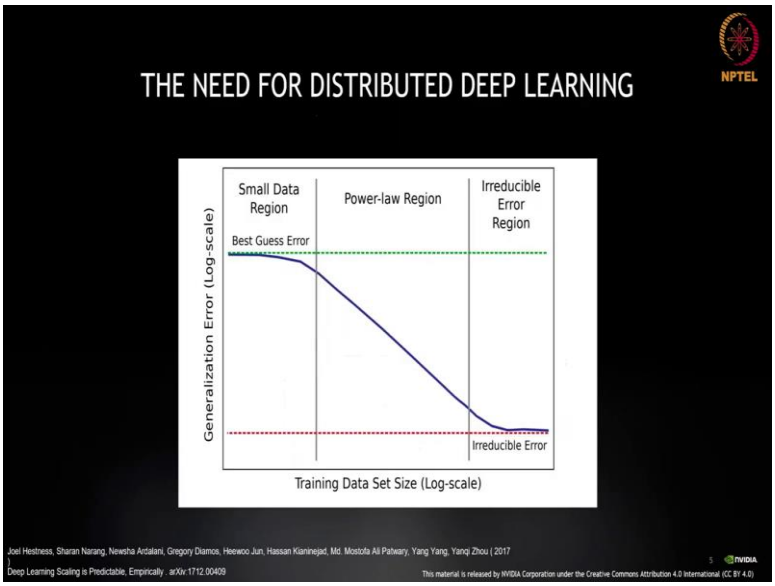
(Refer Slide Time: 06:22)



And what it covers is some of the benefits of training, but it takes this the previous work which we showed to you to the next level, this further and validate this assumption not only on much wider set of models. So, earlier we were talking about a limited set, it takes into consideration more different types of models, but across wide set of problems, not just for one particular type of a problem.

The link is kind of already there, you could anytime go ahead and start reading this paper. It kind of provides further empirical evidence for this relationship between data and accuracy. With the increase in the data and you are seeing different sets of problems translation, language models, character language models, image classification, attention speech models no matter which study which was done empirically it was evident, that with the increase in the data set size there was a logarithmic relation, linear relationship to also increase in the accuracy levels and there have been lot of study which has been done over time.

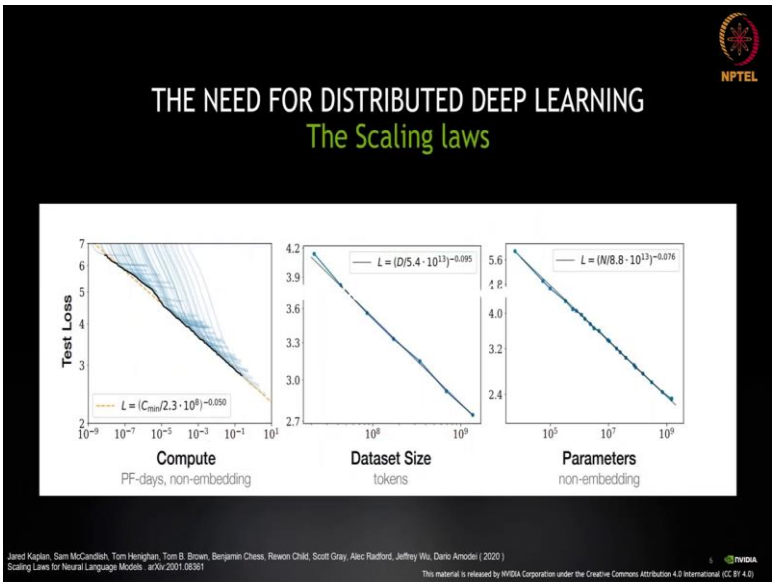
(Refer Slide Time: 07:40)



So, it is estimated that we are currently what you are seeing here is a another representation of the same thing that we are saying, while your x axis is at the logarithmic scale the training data set size, your y axis is the generalization error at the logarithmic scale again. So, on the extreme left what you see is the small data region and you can see the error is not getting better and better, but it reduces with the increase in the data set size, but after some I would say in certain level that error becomes irreducible as well.

So, it is estimated that we are currently in the power law region and we can notice how the generalization error reduces with the increase in the training data set. This increase in the dataset size also comes with increase in compute power required for convergence and that is what is also showed in the next couple of slides.

(Refer Slide Time: 08:48)

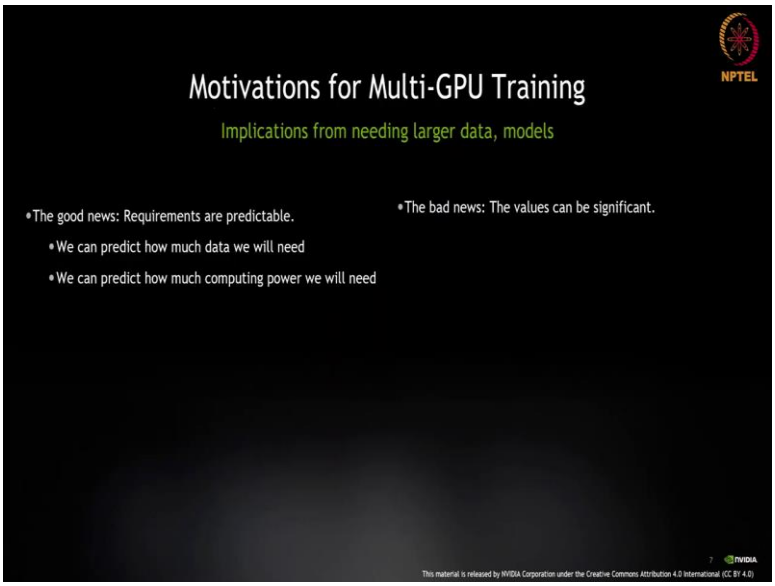


So, what you are seeing in this particular slide is the scaling laws, in this paper which was published the scaling laws of neural language models. There is a linear relationship on the compute requirements with increase in the model parameters and size of data set.

In the first set of slides, we did tell you the amount of flops which were required when there was increase in the data set and also the size or the number of parameters which were supposed to be trained. This gap between the compute requirements for the larger network can be solved by utilizing multiple GPUs and distributing our deep learning workloads.

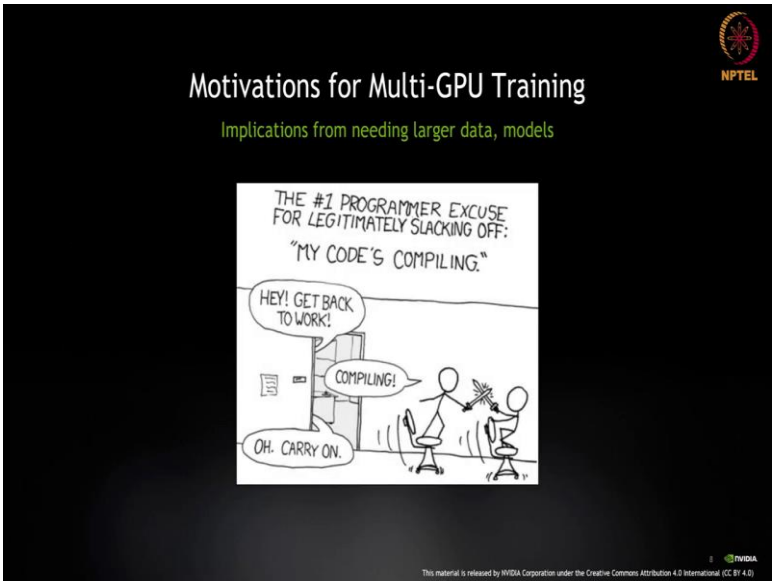
This allows us to add further parallelism giving us faster convergence. So, that is what is one of the critical things which is, so we do not talk about anymore about using for a larger problem statements a single GPU anymore.

(Refer Slide Time: 09:55)



So, if I look at the motivation for the GPU training, the good news is the fact that now we can predict how much data we will need to meet a performance goals right. So, there are different relationship of papers which kind of define the amount of data that is required. The bad news is; obviously, that because of the nature of this relationship for a lot of problem the data volumes can be very very high. So, we are talking about this relationship and the amount of time which it will also take.

(Refer Slide Time: 10:30)



This is a very very interesting slide which we thought, I will just show to you. And this particular slide is kind of representing the life of a data scientist or the excuse given by a lot of the researchers who are working, because like we are saying that like in the earlier days the compilers which we used to use for compiling the code. So, if you look at that in the earlier days, the compiler used to take a lot of time to compile your code as well.

So, a programmer whenever they used to write code and I am talking about codes which are not like few lines of code, like thousands of lines of codes, but I am talking about few tens of thousands or millions lines of code in the open source projects and all like Linux Kernel's and all that are pretty huge. So, it used to take a lot of time to compile those codes. So, if a programmer was going and writing a code, they will basically give it to the compiler for compilation and then used to go for a coffee or used to go for a break.

And then when he used to come back the same problem, maybe there was a error in writing the code and again you go back and correct it and there is a lot of a time which was spent. And the same thing exists for even deep learning or even for the data science world, where when you give a particular if you choose a particular model and if you choose a particular data set you give it for training and most of the times you will just submit a job and come back after a day or a week or sometimes more than that.

Realizing the fact that maybe the model that you tried out is not even good or the changes that you made to the layers was not even working and you are not getting into the convergence zone which you would have expected for you. And a lot of time is actually not spent on research, but most of the time is actually spent in waiting for the results to come back to you and this is evident and evident with the increase in the data sets and the models.

(Refer Slide Time: 12:51)

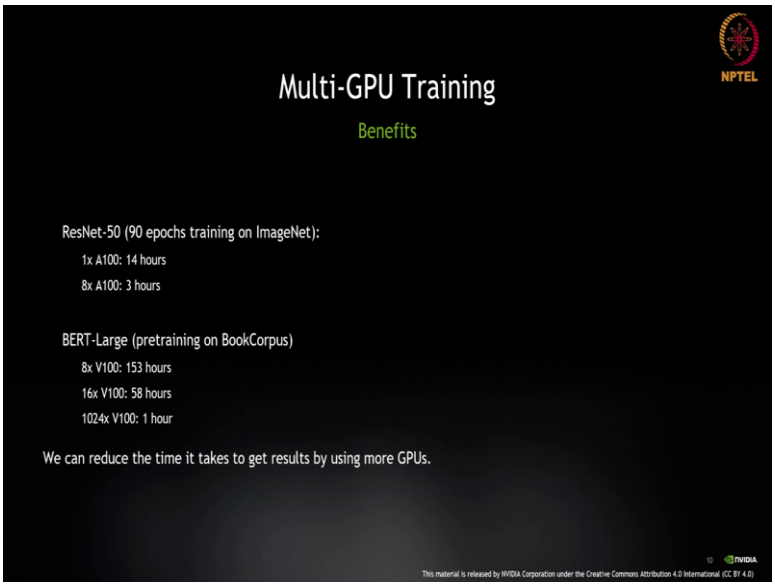
So, the benefits of using multiple GPUs is kind of already I would say established to certain extent, where you can see from a different perspective. So, suppose if you had a particular time duration attached to see your thesis or attached to some work that you wanted to submit or whatever it is.

So, you could do multiple things, you could basically say that I want to train on more data. So, I have a fixed duration I want to do this training in 2 days, now but what I can do is I can provide it with more data within that 2 days. I can do more iterations on it to see if it is converging or I can in fact, rather than a smaller model I can try train on a larger model.

So, we are talking about getting better results or get the same results faster. So, there are two ways of looking at it, one is that I can solve the same problem faster right. So, that I can save that time and I can try out more different kinds of permutations and combinations or you can get better results by trying out something which is larger, like trying out more data, more amount of augmentation to the data that you might have done or trying out much more larger models which otherwise would take you forever.

So, it can be seen from a perspective of both the cases of multi-GPU training.

(Refer Slide Time: 14:24)



So, to give you a snapshot of what kind of results that you can get, even going I am not even comparing it with CPU, I am comparing it from going from one GPU to multiple GPUs, on a CPU it will be manyfold folds slower. Like if you see a number here if I were to train ResNet-50 for 90 epochs on ImageNet data set. So, for a ImageNet data set for ResNet-50 running it for 90 epochs 1 A100 which is our latest generation architecture called as ampere.

Ampere is one of our latest generation architecture it takes around fourteen hours to train on the overall data set. And if I combine and take a note say a DGX machine which has 8 A100 and surprisingly you will see in the tomorrows lecture most of your frameworks also support multi-GPU programming. And if you provide it with 8 A100 the time reduces from 14 hours to 3 hours. This is computer vision, what about say another BERT is one of the largest models out there.

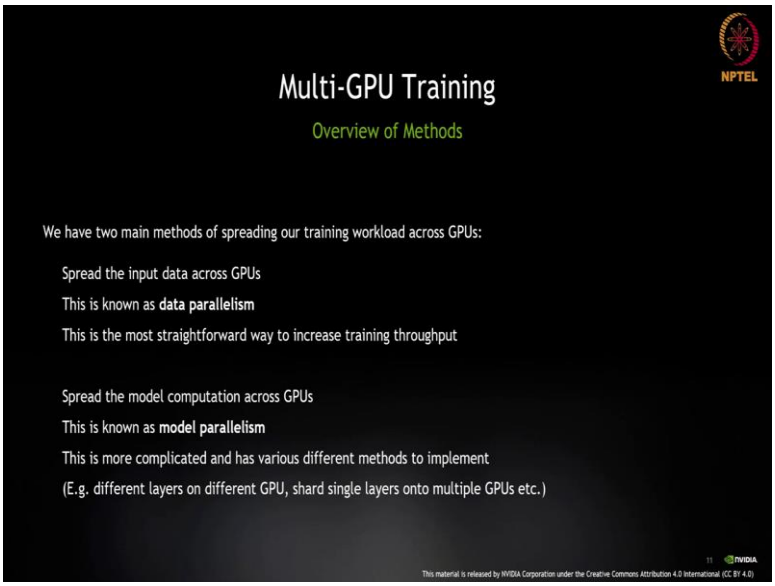
And this one is being pre trained on book corpus right. And if I were to train BERT for this book corpus data set on 8 V100s itself forget about running it on V100 on 8 V 100s it takes 153 hours to train. And hopefully you would have also gone through the previous lecture on transfer learning and why transfer learning also becomes more important.

Because generally all of these trained data sets are kind of retrained and provided to you and ideally you are supposed to work on your additional data set and do the transfer learning. But coming back to our topic of multi-GPU, if you are training from scratch for

this data set if you went from 18 to 16; obviously, the time reduces, but you can see at the level of scale that you can run it on we are talking about running this BERT model which is very much a scalable in nature.

It was designed to be scalable model and you can see it runs on 1024 V100s and it gets finished in 1 hour. Now, think of it like this that if you were to work in this phase of national language processing machine translation and all, and if you are working on the real life data set on a larger model, how much time it would have taken on a single V100 or 8 V100s versus if you had access to a cluster which had say 1024 V 100 cards. So, we can reduce the time it takes to get results by using more number of GPUs.

(Refer Slide Time: 17:26)



So, let us start getting into some more amount of details about the multi-GPU training or the overview of methods themselves. So, today I will quickly introduce you to some of those methods and tomorrow we will get into the details of how to use these methods inside TensorFlow PyTorch and Horovod.

And in the next set of lecture we will also cover the part of the convergence, which means when you start using distributed computing the way your strategy of back propagation or how are you averaging the gradients and all is also going to change and you get into those other algorithmic problem.

Where even though you are using multiple GPUs actually, if you were converging on say within 10 iterations maybe by using distributed computing, you might not be even converging on 20 or 40 iterations and its taking more time to converge. Because the back propagation algorithm or the methods which were developed in the earlier days where primarily not meant for parallel computation or not meant for distributed computing.

So, you may end up actually fast doing it faster, but when it comes to the overall accuracy, you might not be even near. So, you will have to run it for a longer iteration kind of taking away all the benefit that you got. So, we are going to cover that part also in the subsequent lecture. So, there are two main methods when it comes to multi-GPU training, the first method is spread the input data across GPU.

So, generally if you see you might have like a batches of images, like millions of images or 10000 of images, that you have. And you would like to basically spread that input data across the GPUs. So, and this type of multi-GPU training is called as data parallelism, because what you are doing is you are distributing the batch of data across different GPUs.

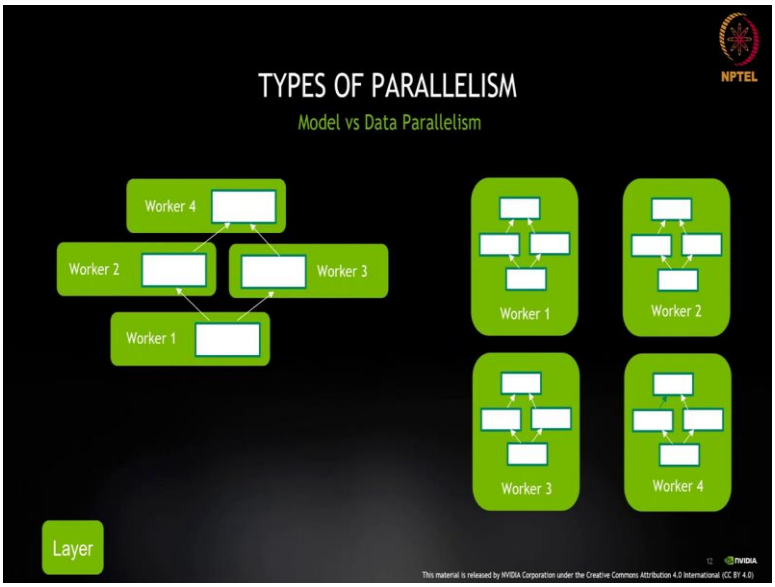
And this is the most common or straightforward way to increase the training throughput because if you see it is kind of embarrassingly parallel. Which means every image is kind of independent of each other and you are just distributing it across different GPUs. So, what I was saying was that we have this two types of methods for the multi-GPU screening.

The first method is referred to as data parallelism, where you are basically distributing the batches of images like generally you will decide the batch of images that you like to give to every GPU and they get spread out and it is the most straightforward way and the embarrassingly parallel way to just distribute the images right.

The second approach is basically the model computation across the GPUs, this is also referred to as model parallelism and we will get into this is more complicated way and there are various and different methods to implement it. For example, you may put different layers of your network onto different GPU or sometimes you may also share the single layer onto multiple GPUs.

It is much more complicated way to make it work. In this particular series we are going to concentrate primarily on the data parallelism part which is the most straightforward way of using the multi-GPU training.

(Refer Slide Time: 21:32)



So, as I told you deep learning takes a complicated problem and simply makes it expensive and hence to reduce our training time to quickly test different networks and study, how they converge we make use of multi-GPU or multi node ecosystem available to us.

Here we can see two types of parallel approaches used in deep learning. The data parallelism and the model parallelism, part of it. Data parallelism would run on the same net would run the same network in different workers or GPUs. When I say worker I am referring to different GPUs, but with a different input data for each worker and then the outputs are aggregated.

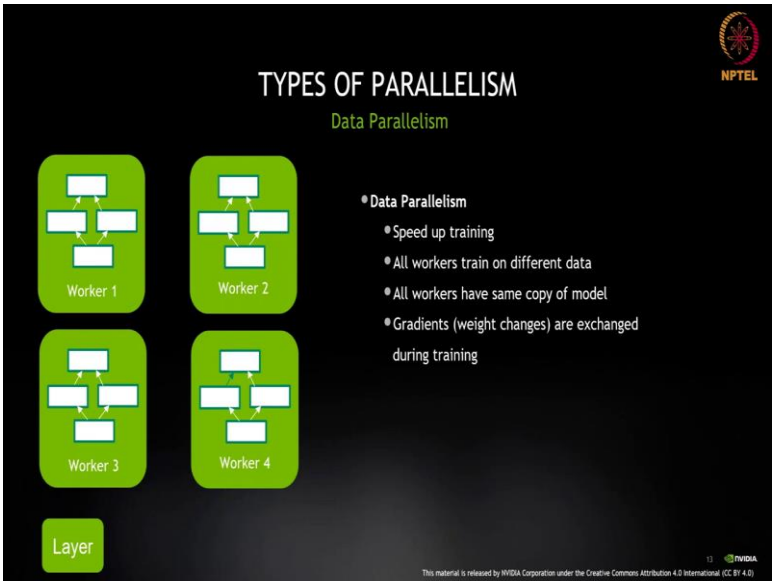
While the mode of parallelism is used when the network is big enough to not fit in a single GPU memory. Like I was talking about the newer set of networks which have few billion parameters. If you see and if you remember the GPU memory is kind of limited in nature, the latest GPUs will give you around 80 GB of GPU memory.

If your network is that large which does not even fit on a single GPU memory, I think model parallelism is the one which is necessary. A combination of these can be used and

they would be also referred to as hybrid parallelism. Same if you want to train a huge network in a multi node setup you can span the network across entire node containing 2 or more GPUs and then each node can then process a different input and aggregate that. This uses both a model parallelism as well as the data parallelism.

So, let us look at both these kinds of parallelism in somewhat details.

(Refer Slide Time: 23:32)



So, the first one that I said is the data parallel part, you can see here the part that which is very much clear is that you see these different workers. Each and every worker is using one GPU. So, like here in this particular case you can assume that there are basically 4 workers and in this 4 workers all are using four different GPUs. And if this was your network or the DL network you can see the same network is replicated across different GPUs.

So, is the same network which gets copied to all different GPUs. So, in a data parallelism, the data set is divided into n parts, where n is the number of GPUs. In the figure above you can see its actually n equal to 4, then the copy of the model is placed on each GPU and trained on the corresponding data chunk.

After that what happens is that the gradients are calculated. If you remember the process of forward propagation and then followed by the backward propagation. After the

forward propagation you get the loss and then what you are doing is basically you will calculate the gradient with respect to the loss function.

So, they are calculated for each copy of the model. So, you can see every worker is basically going to create that gradient and then all the models basically exchange the gradients which is subsequently averaged. So, all the models will independently basically work and they are going to calculate their own gradients and then after the forward propagation after a particular level, like you can decide at which iteration and all.

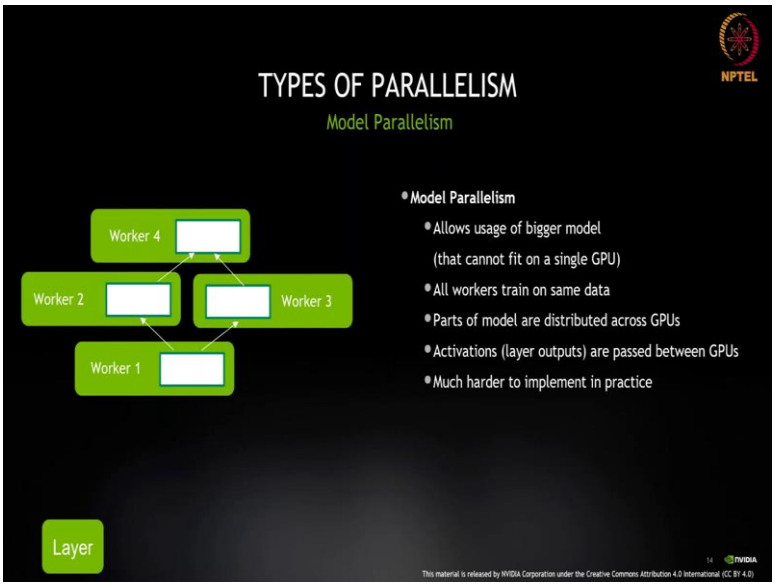
You basically then each and every worker exchanges these gradients across with each and every other worker and you can basically finally, do an and some kind of a averaging, right. So, if you see you are doing is why is the data parallelism you are increasing the speed you are speeding up the training by distributing the data, all workers are basically training on different data sets.

Now, all workers also have the same copy of model, all of these things are very important the gradients which are the weight changes are exchanged during the training. Now, if you have gone into the maths part of the deep learning, here itself you will start getting certain amount of clues in terms of why the convergence will differ, when you run on a single node versus multiple nodes or when you run on a single GPU versus multi GPU.

So, when you start in every model you are first of all giving different data set, every model here would have been initialized to different weights. So, you would be in a different landscape of your loss function right. And that, so you are basically starting at a different location and then finally, the idea here is that when you start doing this averaging you may not reach the same destination within the same set of iteration, within your lost landscape to reach the global minimum.

So, here itself you can start in case if you have studied the maths, you start getting a feeling of what can actually go wrong when you start distributing the work across. So, it is all part of the randomness which is brought in by the deep learning framework itself right.

(Refer Slide Time: 27:16)



So, the second part is the model parallelism. As you can see here in this model there are different ways, here what I am showing you is like worker 1 is actually working on one particular layer inside your network, while worker 2 is basically on another layer worker 3 another layer. So, basically you are creating a graph and each and every worker or GPU is working on different layer, while all of them are using the same data.

So, in the model parallelism one model is divided into n parts, where n is equal to the number of GPUs in the figure here in this case 4 and each part of the model is placed on its separate GPU. The model is placed on separate GPU or basically separated out on separate GPU, then the batch is sequentially calculated on GPU 0, GPU 1 and GPU n. This is where the forward propagation ends, the backward propagation is done in reverse order starting from GPU n and ending with GPU 0.

So, the forward propagation the way it will work is it goes from GPU 0 to GPU 1, 2 and 3 and it goes towards GPU n in the forward propagation. While the backward propagation is done in the reverse order starting from GPU n and ending with GPU 0. The obvious advantage of this particular approach is that we can train a model in this way that does not fit in one GPU. I gave you an example where a networks are becoming so large that they are not even fitting right now in the 80 GB space of 1 GPU.

However, this is also a drawback if you see, in model parallelism while computing on GPU number say 1 on whatever number certain number is in progress, all of the rest are

idle. As you can see right GPU 0 in the forward propagation works then 1, then 2, then it goes to n and in the backward its n then 4, then 3, then 2 and it goes in the backward. This problem can be solved or is solved by switching to asynchronous GPU operations.

So, what we are talking about here is a type where we are talking about only synchronous operation, where I will wait till the time the other is over and that is what is called as synchronous operations. While we can also use something called as asynchronous operations which we will see in the next set of slides.

In fact, these are the parameters that you are going to decide and use in different frameworks, like are you going to use data parallelism, are you going to use model parallelism. How many workers you are going to run it on, are you going to use synchronous or are you going to use asynchronous. These are all the decisions which are made by the by the developer of the network.