Applied Accelerated Artificial Intelligence

Prof. Bharathkumar
Department of Computer Science and Engineering
Indian Institute of Technology, Palakkad

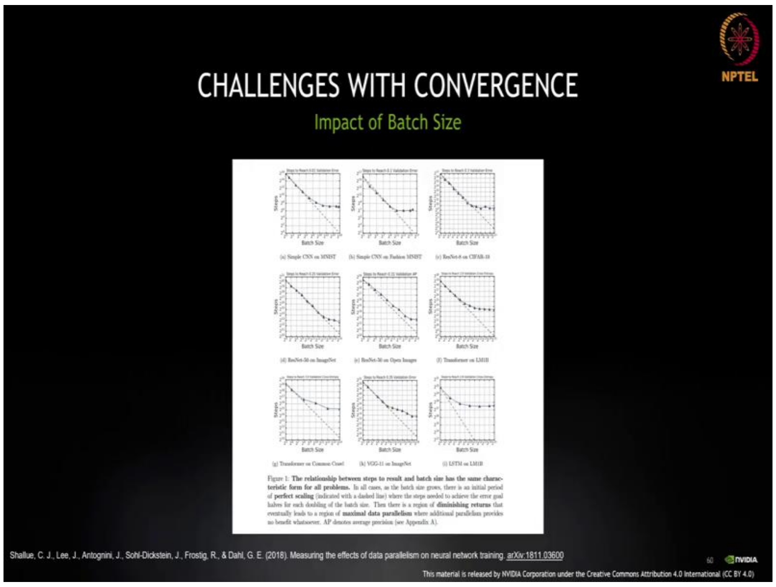
Lecture - 39
Challenges with Distributed Deep Learning Training Convergence

(Refer Slide Time: 00:14)

Session 3: CHALLENGES WITH CONVERGENCE

So, let us get started on the next topic we showed you how to use the distributed deep learning using two frameworks. But when you start using distributed deep learning, there are certain problems or challenges that you will face and that is what we are going to look in this particular certain slides that we have.

(Refer Slide Time: 00:38)



CHALLENGES WITH CONVERGENCE

Impact of Batch Size

(a) Simple CNN on MNIST (b) Simple CNN on Fashion MNIST (c) ResNet-8 on CIFAR-10

(d) ResNet-50 on ImageNet (e) ResNet-50 on Open Images (f) Transformer on LM1B

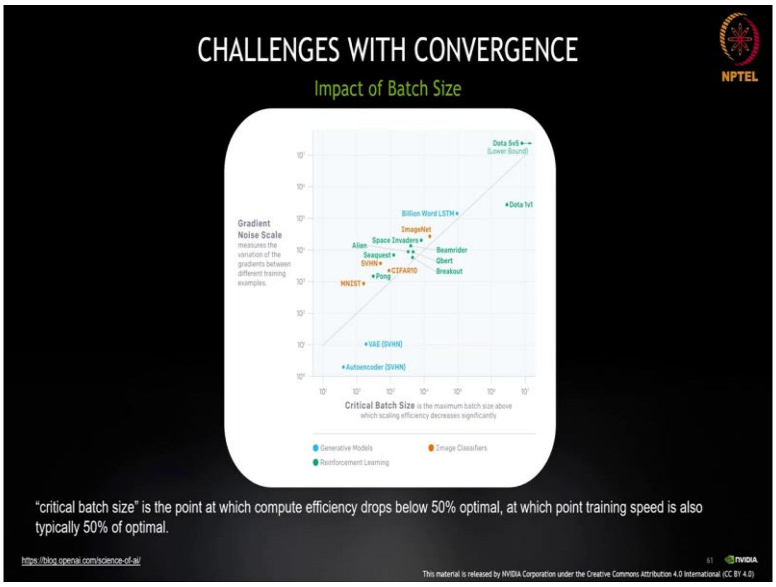
(g) Transformer on Common Crawl (h) VGG-11 on ImageNet (i) LSTM on LM1B

Figure 1: **The relationship between steps to result and batch size has the same characteristic form for all problems.** In all cases, as the batch size grows, there is an initial period of **perfect scaling** (indicated with a dashed line) where the steps needed to achieve the error goal halves for each doubling of the batch size. Then there is a region of **diminishing returns** that eventually leads to a region of **maximal data parallelism** where additional parallelism provides no benefit whatsoever. AP denotes average precision (see Appendix A).

Shallue, C. J., Lee, J., Antognini, J., Sohl-Dickstein, J., Frostig, R., & Dahl, G. E. (2018). Measuring the effects of data parallelism on neural network training. arXiv:1811.03600

We see an improvement in performance when we increase the batch size that is what we showed on the first day, but it has been found that once we hit the limit which is also referred to as a critical batch size, we exhaust the possible data parallelism that we can achieve. So, you can see a paper which kind of highlights, this part if you can see there is a particular relationship that they are trying to show by increasing the batch size and it affects when you want to improve the data parallelism and it kind of exposes a particular limit which is referred to as a critical batch size.

(Refer Slide Time: 01:22)



CHALLENGES WITH CONVERGENCE

Impact of Batch Size

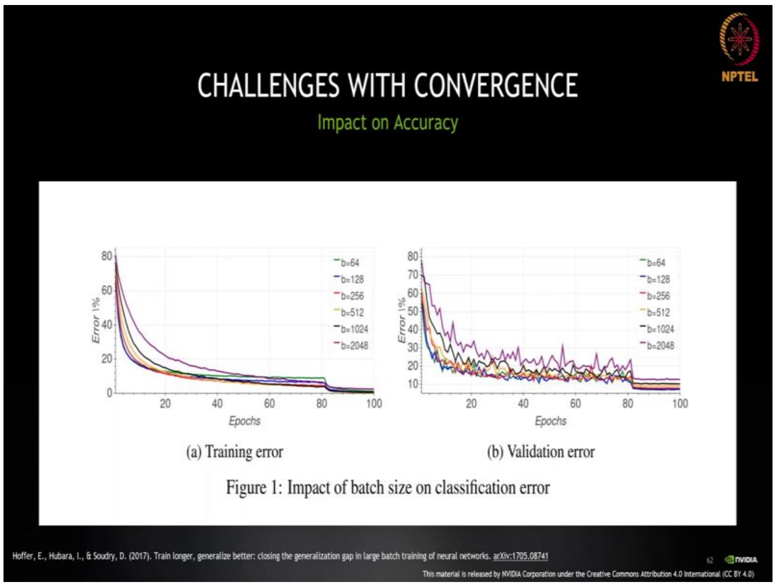
"critical batch size" is the point at which compute efficiency drops below 50% optimal, at which point training speed is also typically 50% of optimal.

https://blog.openai.com/science-of-ai/

So, here you can see a critical batch size is a point at which the compute efficiency drops below 50 percent of the optimal at which point the training speed is also typically 50 percent of optimal. The critical batch size is found out to be different for different networks and that is what its being shown here and it is a function of the gradient noise scale. As you can see here your y axis is actually gradient noise scale which measures the variation of gradient between different training examples.

So, the critical batch size will be different for different networks and it is a function of gradient noise scale. So, when the noise scale is small, looking at a lot of data in parallel quickly becomes redundant whereas, if it is large we can still learn a lot from huge batch sizes and that is what the relationship is being shown in this particular paper here.

(Refer Slide Time: 02:25)



CHALLENGES WITH CONVERGENCE

Impact on Accuracy

Figure 1: Impact of batch size on classification error

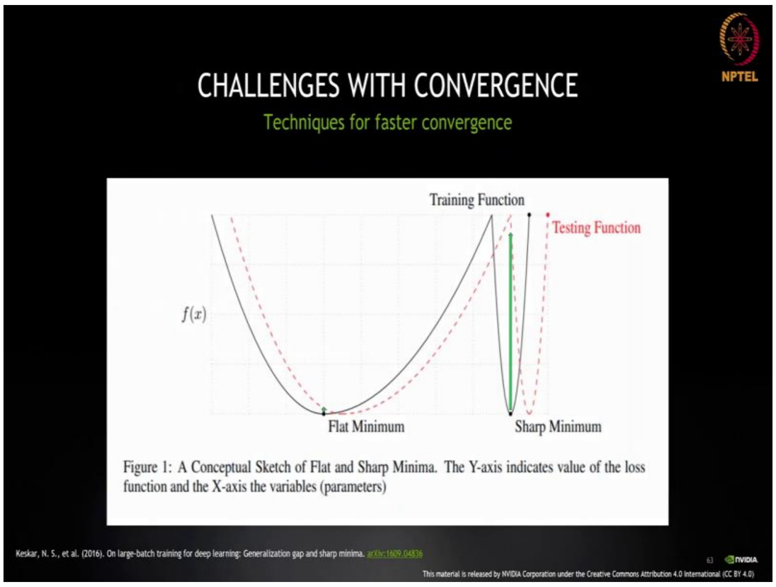
Hoffer, E., Hubara, I., & Soudry, D. (2017). Train longer, generalize better: closing the generalization gap in large batch training of neural networks. arXiv:1705.08741

62

One more problem that we find with the scaling is the loss of accuracy. What you are seeing here is a graph which is part of the paper where you can see when you change the batch size actually there is a jump you can see that the accuracy is fluctuating and that is what was also observed.

So, if you change this hyper parameter, we can notice the change in the above graph right how the error actually oscillates when you increase or decrease the batch size now that is also very very critical. So, if you change anything which is also referred to as a hyper parameter, it may result into certain problems with respect to the accuracy levels.

(Refer Slide Time: 03:12)



CHALLENGES WITH CONVERGENCE

Techniques for faster convergence

Figure 1: A Conceptual Sketch of Flat and Sharp Minima. The Y-axis indicates value of the loss function and the X-axis the variables (parameters)

Keskar, N. S., et al. (2016). On large-batch training for deep learning: Generalization gap and sharp minima. arXiv:1609.04836

63

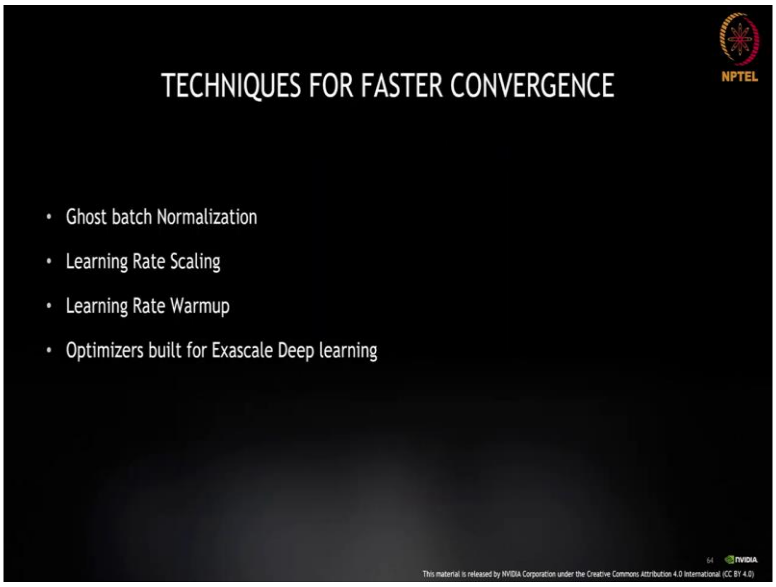
Let us look into more detail which is also highlighted in this paper where having a smaller batch size basically has a higher noise as compared to large batch size and help with getting out of local minimas which is in the case with the large batch size. Also in the above picture that you see, we see that we see two kinds of minimas sharp minima and a flat minima and these minimas are basically function of data sets when the original function is having a slight offset.

The flat minima error is primarily negligible whereas, the sharp minima function as a very large error and that is what makes it really difficult and it can create many problems and it is a function of the data set and this convergence to a sharp minima happens more frequently in the case of higher batch sizes.

And the optimizer basically can find both of them as equally good candidate. So, you can see here based on your data set this curve or this function of your loss is going to change and based on even if there was a slight offset with respect to change you did in the hyper parameter in the flat minima, it may not impact your accuracy so much.

But in case your data set was of a function in which was of type having a sharp minima even a slight change can result into huge variations and these kinds of things are what creates lot of problems and there are different ways to solve this also and we are going to look at couple of them in a bit.

(Refer Slide Time: 04:58)



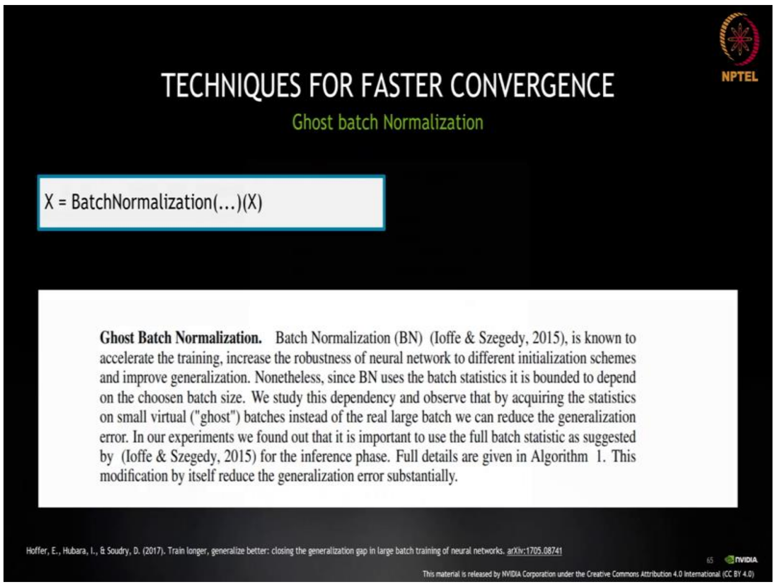
TECHNIQUES FOR FASTER CONVERGENCE

- Ghost batch Normalization
- Learning Rate Scaling
- Learning Rate Warmup
- Optimizers built for Exascale Deep learning

So, we have seen what are the problems with scaling and the reasons behind them, but let us see some of these techniques. We are going to cover four techniques which are listed here ghost batch normalization, we are going to talk about learning rate which is another hyper parameter and usually we can change the learning rate and it is not usually constant.

There is a concept of warm up which we will discuss in more details and why it is important followed by using optimizers which are not traditionally seen in the normal deep learning world, but they become really important when we are talking about petascale level of the scaling or 2000s of GPUs as well.

(Refer Slide Time: 05:48)



TECHNIQUES FOR FASTER CONVERGENCE

Ghost batch Normalization

```
X = BatchNormalization(...)(X)
```

Ghost Batch Normalization. Batch Normalization (BN) (Ioffe & Szegedy, 2015), is known to accelerate the training, increase the robustness of neural network to different initialization schemes and improve generalization. Nonetheless, since BN uses the batch statistics it is bounded to depend on the choosen batch size. We study this dependency and observe that by acquiring the statistics on small virtual ("ghost") batches instead of the real large batch we can reduce the generalization error. In our experiments we found out that it is important to use the full batch statistic as suggested by (Ioffe & Szegedy, 2015) for the inference phase. Full details are given in Algorithm 1. This modification by itself reduce the generalization error substantially.

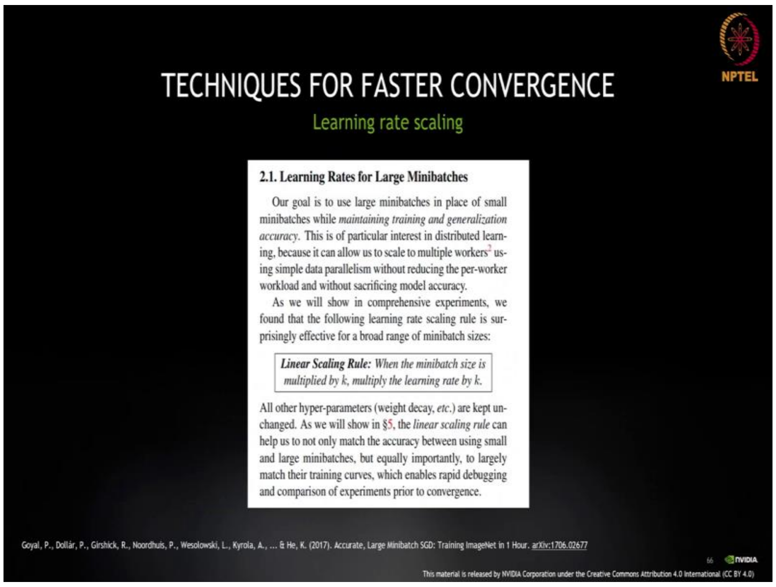
Hoffer, E., Hubara, I., & Soudry, D. (2017). Train longer, generalize better: closing the generalization gap in large batch training of neural networks. arXiv:1705.08741

65

So, batch normalization basically lead us add different noise to different parts of the batch independently. This increase in noise helps in learning better. This can be implemented in Horovod as its follows the MPI model. So, different GPUs have their own batches and we can train with them this improves the overall accuracy of our model.

So, you will see that if you use Horovod versus Tensorflow or the in a implementation of Pytorch if you distribute the work even though the model is the same everything is the same if the distribution strategy is different, you would realize that one of them is converging faster as compared to the other distribution strategy and this is quite common and you can reduce the impact of it by doing certain features like batch normalization.

(Refer Slide Time: 06:45)



TECHNIQUES FOR FASTER CONVERGENCE

Learning rate scaling

2.1. Learning Rates for Large Minibatches

Our goal is to use large minibatches in place of small minibatches while *maintaining training and generalization accuracy*. This is of particular interest in distributed learning, because it can allow us to scale to multiple workers[2] using simple data parallelism without reducing the per-worker workload and without sacrificing model accuracy.

As we will show in comprehensive experiments, we found that the following learning rate scaling rule is surprisingly effective for a broad range of minibatch sizes:

Linear Scaling Rule: *When the minibatch size is multiplied by k, multiply the learning rate by k.*

All other hyper-parameters (weight decay, *etc.*) are kept unchanged. As we will show in §5, the *linear scaling rule* can help us to not only match the accuracy between using small and large minibatches, but equally importantly, to largely match their training curves, which enables rapid debugging and comparison of experiments prior to convergence.

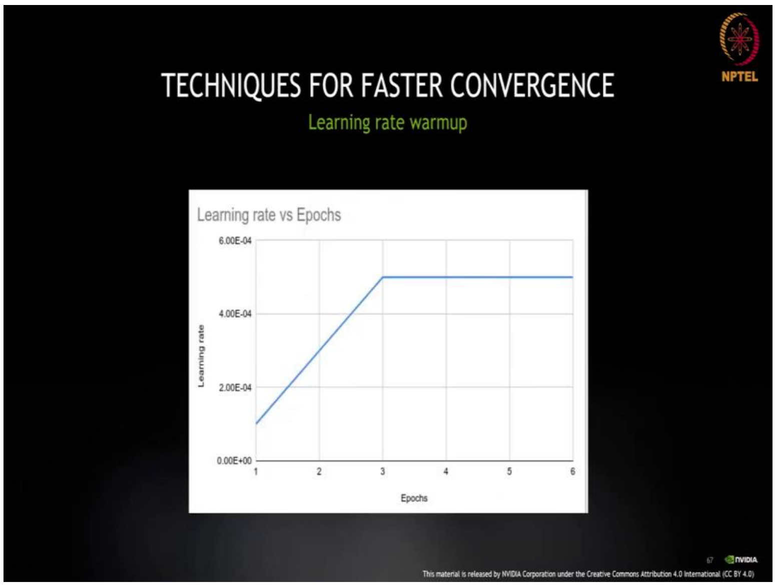
Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., ... & He, K. (2017). Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. arXiv:1706.02677

66

The second example here is for faster convergence is the learning rate. You can again see the reference to the paper by scaling the learning rate basically what this paper states is that by scaling the learning rate linearly with the batch size, it has a model convergence rate and the number of epochs needed to converge would be preserved that way and our training becomes faster. The linear scaling rule it states is as follows. It states that when the batch size is multiplied by k you should also multiply the learning rate by k.

So, if you have mini batch sizes multiplied by k if you are changing your batch size in that case your learning rate should also increase by k and that is what was the observation inside this particular paper which was there.

(Refer Slide Time: 07:42)


TECHNIQUES FOR FASTER CONVERGENCE

Learning rate warmup

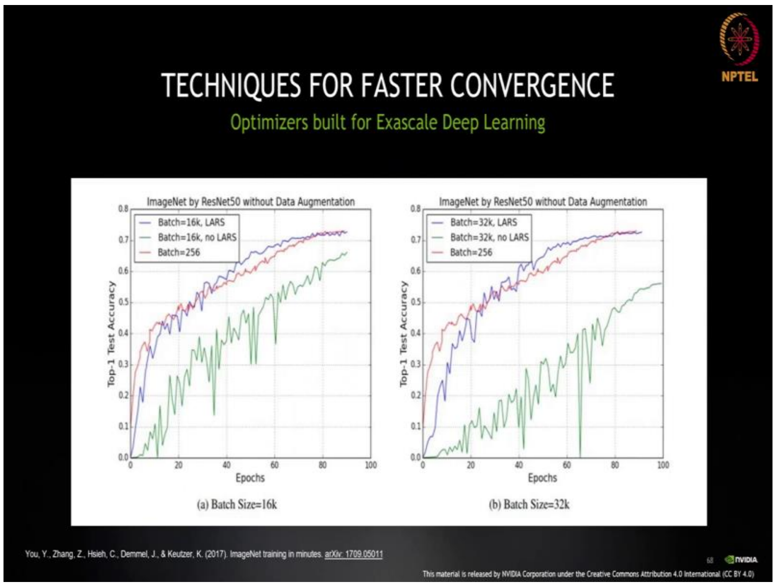
67

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

But one side effect with the linear scaling rule we saw in the last slide is that it breaks down when the network is changing very rapidly and as you would know the rapid change of network generally happens in the early stage of training because all of your weights are random. And you are actually changing at a very higher rate and that kind of occurs in the early stage and this issue can be alleviated by a properly designed learning rate warm up a strategy of using less aggressive learning rate at the start of training and increasing it gradually.

So, you can see here for the initial first couple of epochs, the learning rate is not increased at a particular a higher way, but later on it gets increased. So, that is what its very very important and its also referred to as learning rate warm up. It has really good impact at least in the initial stages because otherwise your linear scaling that we define will be basically break down.

(Refer Slide Time: 08:48)



TECHNIQUES FOR FASTER CONVERGENCE

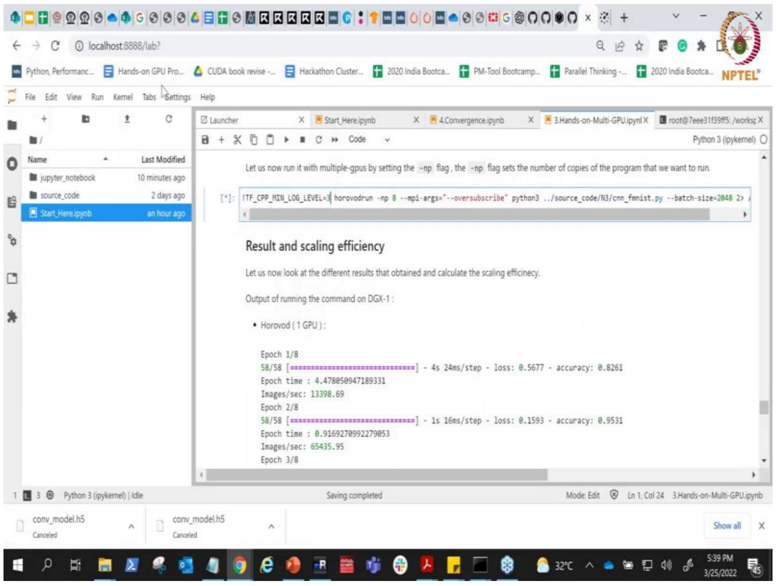
Optimizers built for Exascale Deep Learning

You, Y., Zhang, Z., Hsieh, C., Demmel, J., & Keutzer, K. (2017). ImageNet training in minutes. arXiv: 1709.05011

This material is released by NVIDIA Corporation under the Creative Commons Attribution 4.0 International (CC BY 4.0)

Finally, what we are also saying is that we can use optimizers that are built for exascale deep learning that tackle the scaling problem to a certain extent. There are different kinds of different types of optimizers which are built for exascale or petascale level deep learning like here you can see there are two graphs one of the graph is with lars which is another type of optimizer and without lars which is a traditional optimizer and you can see at the accuracy which you get is very very different right. So, and that is what it is the one that which is being observed.

(Refer Slide Time: 09:42)



Let us now run it with multiple-gpus by setting the `-np` flag , the `-np` flag sets the number of copies of the program that we want to run.

```
[*]: !TF_CPP_MIN_LOG_LEVEL=3 horovodrun -np 8 --mpi-args="--oversubscribe" python3 ../source_code/N3/cnn_fmnist.py --batch-size=2048 2>
```

Result and scaling efficiency

Let us now look at the different results that obtained and calculate the scaling efficinecy.

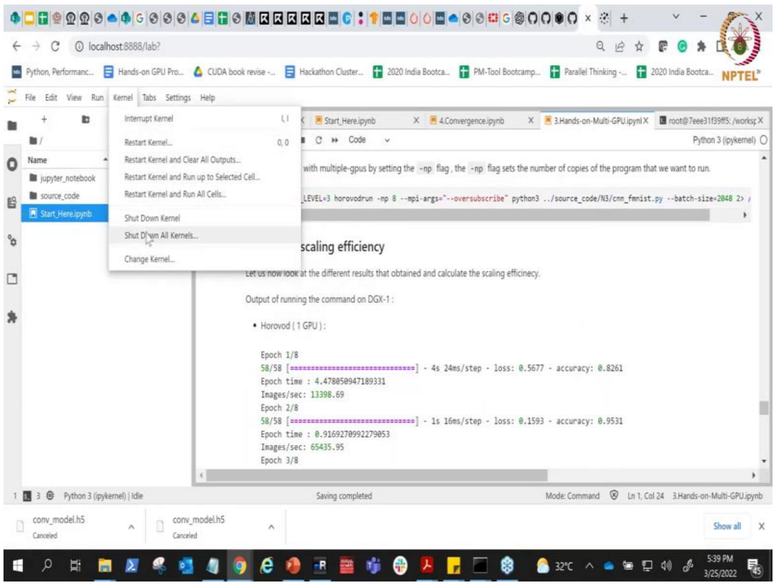
Output of running the command on DGX-1 :

- Horovod (1 GPU) :

```
Epoch 1/8
58/58 [==============================] - 4s 24ms/step - loss: 0.5677 - accuracy: 0.8261
Epoch time : 4.478050947189331
Images/sec: 13398.69
Epoch 2/8
58/58 [==============================] - 1s 16ms/step - loss: 0.1593 - accuracy: 0.9531
Epoch time : 0.9169270992279053
Images/sec: 65435.95
Epoch 3/8
```

So, let me again go back to the Jupyter notebook to give you.

(Refer Slide Time: 09:44)



with multiple-gpus by setting the `-np` flag , the `-np` flag sets the number of copies of the program that we want to run.

```
LEVEL=3 horovodrun -np 8 --mpi-args="--oversubscribe" python3 ../source_code/N3/cnn_fmnist.py --batch-size=2048 2>
```

scaling efficiency

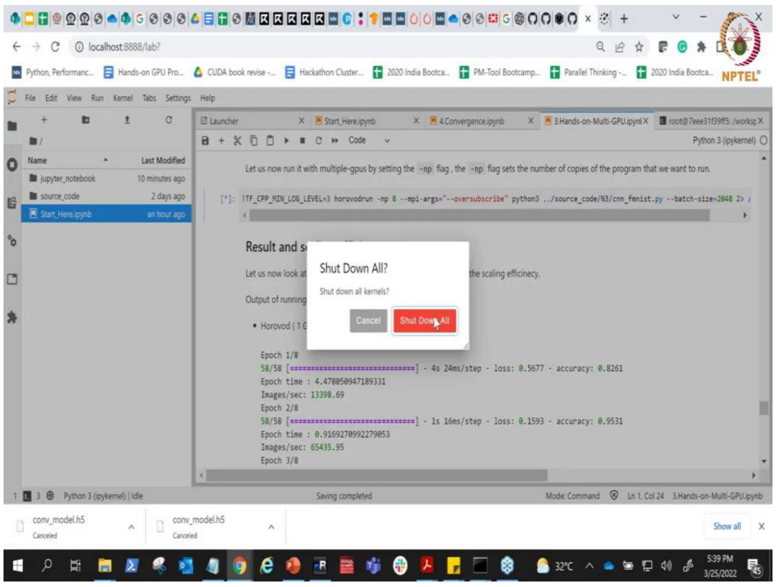
Let us now look at the different results that obtained and calculate the scaling efficinecy.

Output of running the command on DGX-1 :

- Horovod (1 GPU) :

```
Epoch 1/8
58/58 [==============================] - 4s 24ms/step - loss: 0.5677 - accuracy: 0.8261
Epoch time : 4.478050947189331
Images/sec: 13398.69
Epoch 2/8
58/58 [==============================] - 1s 16ms/step - loss: 0.1593 - accuracy: 0.9531
Epoch time : 0.9169270992279053
Images/sec: 65435.95
Epoch 3/8
```

(Refer Slide Time: 09:46)



Let us now run it with multiple-gpus by setting the `-np` flag , the `-np` flag sets the number of copies of the program that we want to run.

```
[*]: !TF_CPP_MIN_LOG_LEVEL=3 horovodrun -np 8 --mpi-args="--oversubscribe" python3 ../source_code/N3/cnn_fmnist.py --batch-size=2048 2>
```

Result and s

Let us now look at ... the scaling efficinecy.

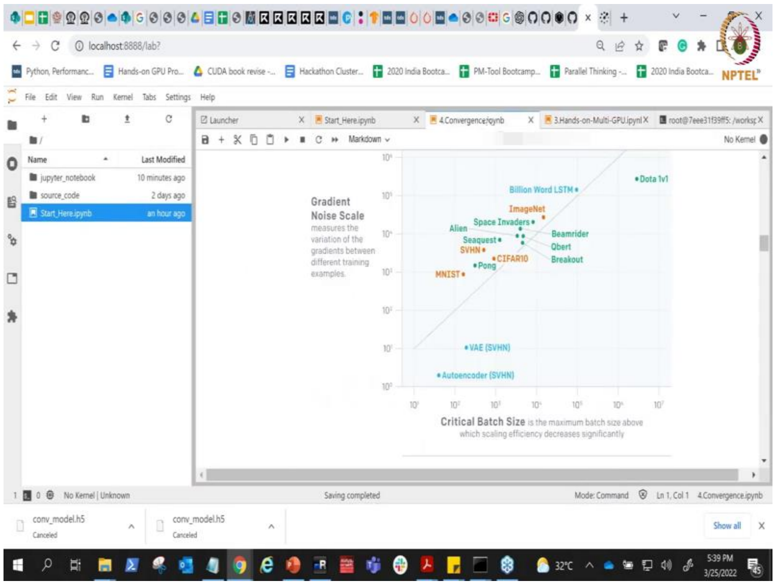
Output of running

- Horovod (1 G

```
Epoch 1/8
58/58 [==============================] - 4s 24ms/step - loss: 0.5677 - accuracy: 0.8261
Epoch time : 4.478050947189331
Images/sec: 13398.69
Epoch 2/8
58/58 [==============================] - 1s 16ms/step - loss: 0.1593 - accuracy: 0.9531
Epoch time : 0.9169270992279053
Images/sec: 65435.95
Epoch 3/8
```

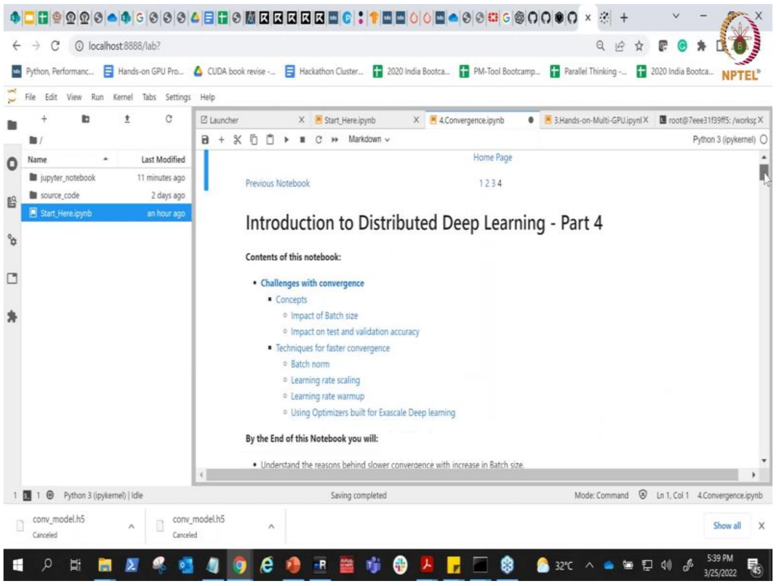
Let me just kill this overall thing.

(Refer Slide Time: 09:52)


Gradient Noise Scale
measures the variation of the gradients between different training examples.
Dota 1v1
Billion Word LSTM
ImageNet
Space Invaders
Alien
Seaquest
SVHN
Beamrider
Qbert
Breakout
CIFAR10
Pong
MNIST
VAE (SVHN)
Autoencoder (SVHN)
Critical Batch Size is the maximum batch size above which scaling efficiency decreases significantly

So, I am going to shut down all the kernels just to make sure that there is nothing which is going wrong in what I am trying to do here in this particular slide yeah.

(Refer Slide Time: 10:06)


Home Page

Previous Notebook 1 2 3 4

Introduction to Distributed Deep Learning - Part 4

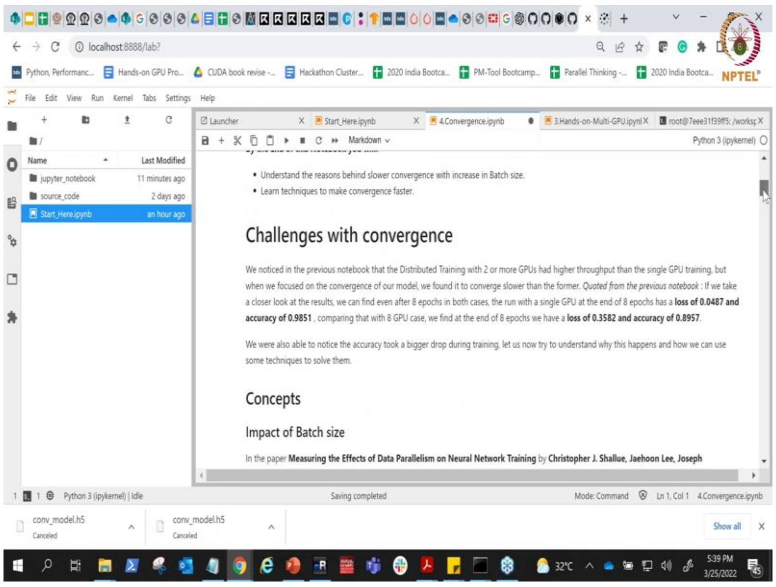
Contents of this notebook:

- **Challenges with convergence**
  - Concepts
    - Impact of Batch size
    - Impact on test and validation accuracy
  - Techniques for faster convergence
    - Batch norm
    - Learning rate scaling
    - Learning rate warmup
    - Using Optimizers built for Exascale Deep learning

By the End of this Notebook you will:

- Understand the reasons behind slower convergence with increase in Batch size

(Refer Slide Time: 10:09)



- Understand the reasons behind slower convergence with increase in Batch size.
- Learn techniques to make convergence faster.

Challenges with convergence

We noticed in the previous notebook that the Distributed Training with 2 or more GPUs had higher throughput than the single GPU training, but when we focused on the convergence of our model, we found it to converge slower than the former. *Quoted from the previous notebook* : If we take a closer look at the results, we can find even after 8 epochs in both cases, the run with a single GPU at the end of 8 epochs has a **loss of 0.0487 and accuracy of 0.9851** , comparing that with 8 GPU case, we find at the end of 8 epochs we have a **loss of 0.3582 and accuracy of 0.8957**.

We were also able to notice the accuracy took a bigger drop during training, let us now try to understand why this happens and how we can use some techniques to solve them.

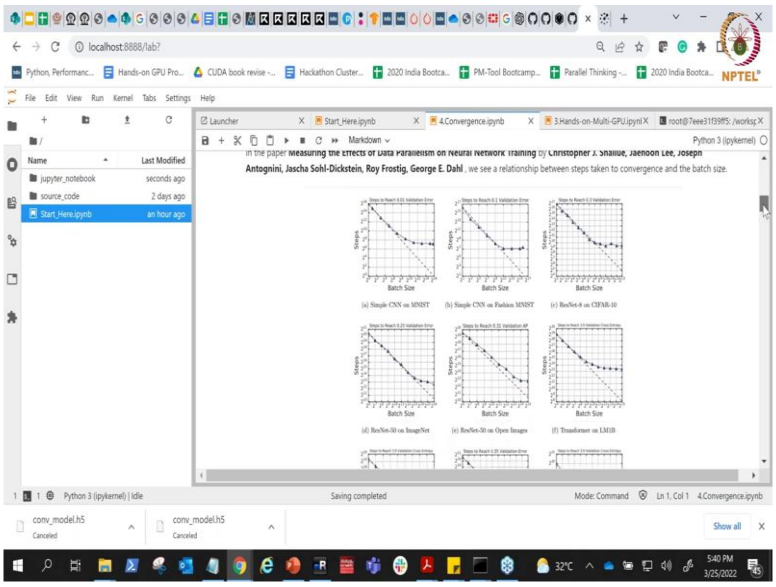
Concepts

Impact of Batch size

In the paper **Measuring the Effects of Data Parallelism on Neural Network Training** by **Christopher J. Shallue, Jaehoon Lee, Joseph**

So, at least my kernel is running now. So, we already talked about the challenges with the convergence that we saw.

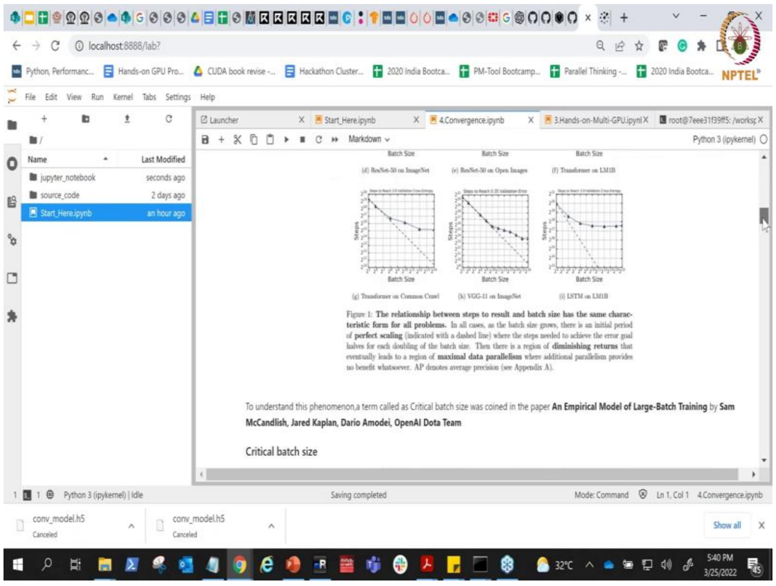
(Refer Slide Time: 10:18)



In the paper **Measuring the Effects of Data Parallelism on Neural Network Training** by **Christopher J. Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, George E. Dahl** , we see a relationship between steps taken to convergence and the batch size.

And in the last era we saw.

(Refer Slide Time: 10:28)



(d) ResNet-50 on ImageNet
(e) ResNet-50 on Open Images
(f) Transformer on LM1B
Steps
Batch Size
(g) Transformer on Common Crawl
(h) VGG-11 on ImageNet
(i) LSTM on LM1B

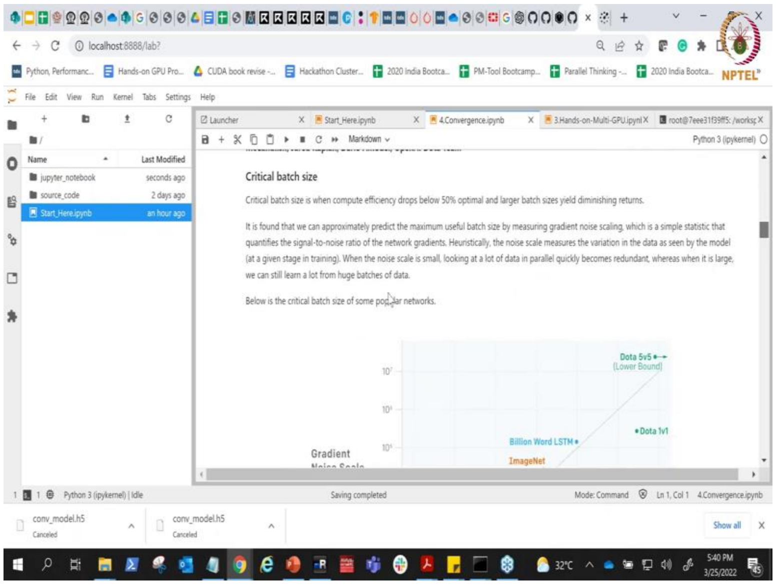
Figure 1: **The relationship between steps to result and batch size has the same characteristic form for all problems.** In all cases, as the batch size grows, there is an initial period of **perfect scaling** (indicated with a dashed line) where the steps needed to achieve the error goal halves for each doubling of the batch size. Then there is a region of **diminishing returns** that eventually leads to a region of **maximal data parallelism** where additional parallelism provides no benefit whatsoever. AP denotes average precision (see Appendix A).

To understand this phenomenon,a term called as Critical batch size was coined in the paper **An Empirical Model of Large-Batch Training** by **Sam McCandlish, Jared Kaplan, Dario Amodei, OpenAI Dota Team**

Critical batch size

So, we also talked about the impact of the batch sizes in the paper you would see the impact of it which. So, you would have a reference to all of these papers in general.

(Refer Slide Time: 10:29)



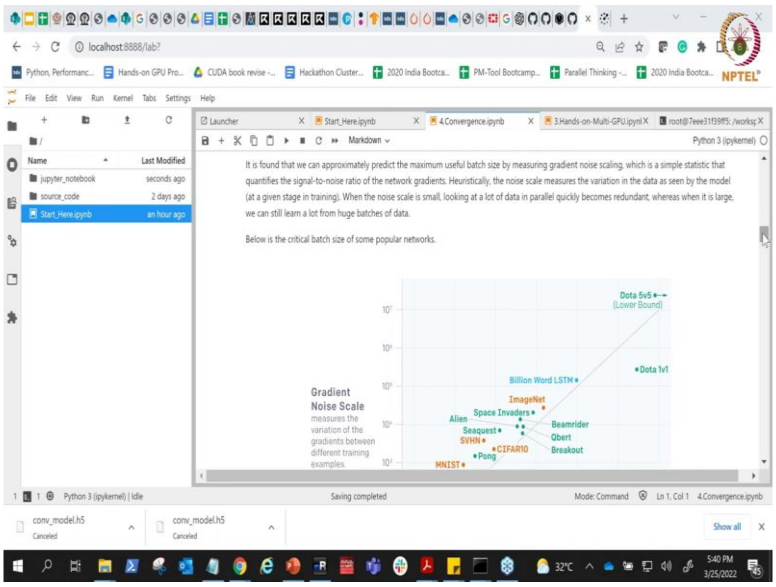
Critical batch size

Critical batch size is when compute efficiency drops below 50% optimal and larger batch sizes yield diminishing returns.

It is found that we can approximately predict the maximum useful batch size by measuring gradient noise scaling, which is a simple statistic that quantifies the signal-to-noise ratio of the network gradients. Heuristically, the noise scale measures the variation in the data as seen by the model (at a given stage in training). When the noise scale is small, looking at a lot of data in parallel quickly becomes redundant, whereas when it is large, we can still learn a lot from huge batches of data.

Below is the critical batch size of some popular networks.

(Refer Slide Time: 10:36)

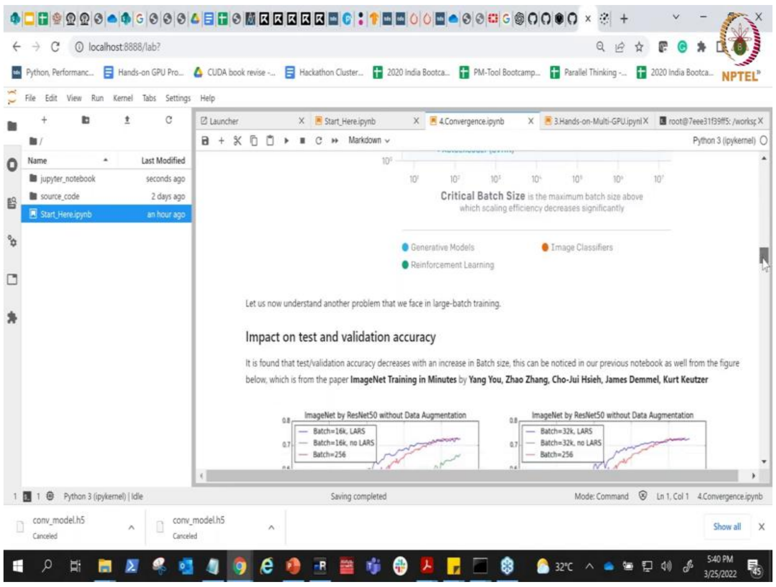


It is found that we can approximately predict the maximum useful batch size by measuring gradient noise scaling, which is a simple statistic that quantifies the signal-to-noise ratio of the network gradients. Heuristically, the noise scale measures the variation in the data as seen by the model (at a given stage in training). When the noise scale is small, looking at a lot of data in parallel quickly becomes redundant, whereas when it is large, we can still learn a lot from huge batches of data.

Below is the critical batch size of some popular networks.

So, you do not need to worry about it you will get references you can download these papers and also refer to them ok its the problem again obtained ok.

(Refer Slide Time: 10:53)



Critical Batch Size is the maximum batch size above which scaling efficiency decreases significantly
Generative Models
Image Classifiers
Reinforcement Learning

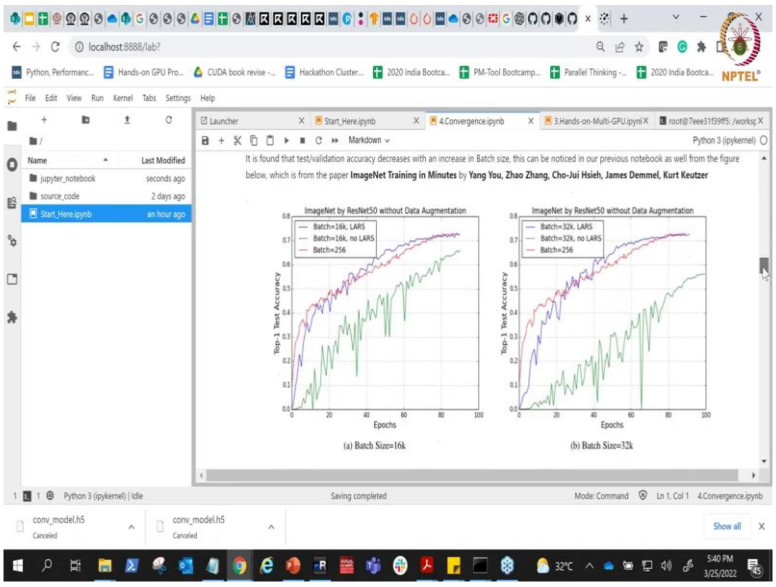
Let us now understand another problem that we face in large-batch training.

Impact on test and validation accuracy

It is found that test/validation accuracy decreases with an increase in Batch size, this can be noticed in our previous notebook as well from the figure below, which is from the paper **ImageNet Training in Minutes** by **Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, Kurt Keutzer**

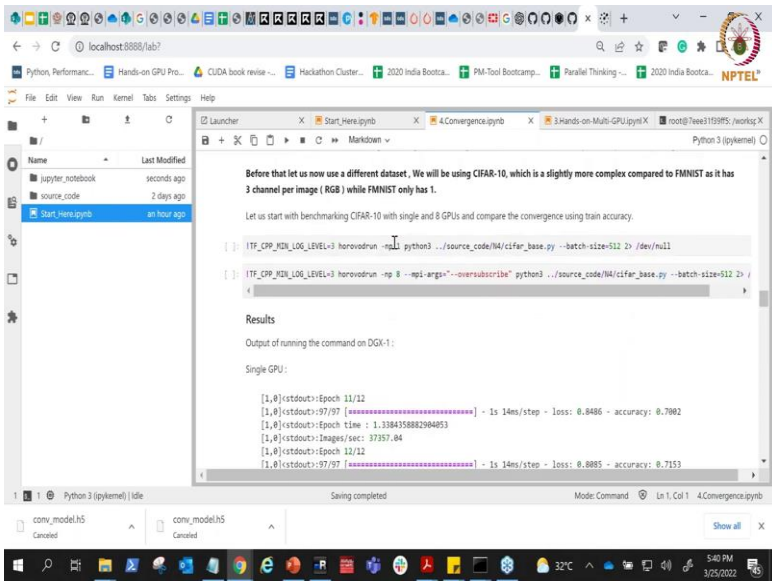
ImageNet by ResNet50 without Data Augmentation
Batch=16k, LARS
Batch=16k, no LARS
Batch=256
ImageNet by ResNet50 without Data Augmentation
Batch=32k, LARS
Batch=32k, no LARS
Batch=256

(Refer Slide Time: 10:54)


It is found that test/validation accuracy decreases with an increase in Batch size, this can be noticed in our previous notebook as well from the figure below, which is from the paper **ImageNet Training in Minutes** by **Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, Kurt Keutzer**

So, its almost the same thing that I talked to you about.

(Refer Slide Time: 10:58)


Before that let us now use a different dataset , We will be using CIFAR-10, which is a slightly more complex compared to FMNIST as it has 3 channel per image (RGB) while FMNIST only has 1.

Let us start with benchmarking CIFAR-10 with single and 8 GPUs and compare the convergence using train accuracy.

```
!TF_CPP_MIN_LOG_LEVEL=3 horovodrun -np 1 python3 ../source_code/N4/cifar_base.py --batch-size=512 2> /dev/null
```

```
!TF_CPP_MIN_LOG_LEVEL=3 horovodrun -np 8 --mpi-args="--oversubscribe" python3 ../source_code/N4/cifar_base.py --batch-size=512 2> /
```

Results

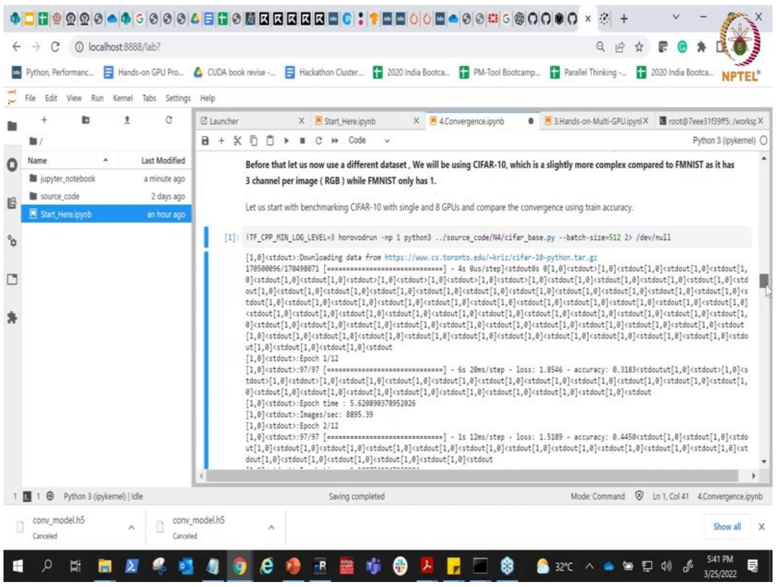
Output of running the command on DGX-1 :

Single GPU :

```
[1,0]<stdout>:Epoch 11/12
[1,0]<stdout>:97/97 [==============================] - 1s 14ms/step - loss: 0.8486 - accuracy: 0.7002
[1,0]<stdout>:Epoch time : 1.3384358882904053
[1,0]<stdout>:Images/sec: 37357.04
[1,0]<stdout>:Epoch 12/12
[1,0]<stdout>:97/97 [==============================] - 1s 14ms/step - loss: 0.8085 - accuracy: 0.7153
```

But hopefully if Horovod runs I would be able to show you a very quick demo it does not run we inform ok.

(Refer Slide Time: 11:06)



Before that let us now use a different dataset , We will be using CIFAR-10, which is a slightly more complex compared to FMNIST as it has 3 channel per image (RGB) while FMNIST only has 1.

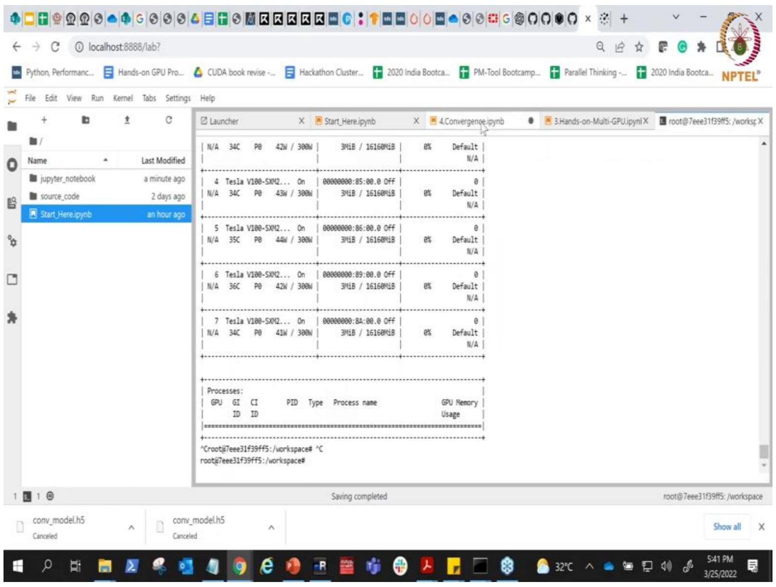
Let us start with benchmarking CIFAR-10 with single and 8 GPUs and compare the convergence using train accuracy.

```
[1]: !TF_CPP_MIN_LOG_LEVEL=3 horovodrun -np 1 python3 ../source_code/N4/cifar_base.py --batch-size=512 2> /dev/null
```

```
[1,0]<stdout>:Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170500096/170498071 [==============================] - 4s 0us/step]<stdout0s 0[1,0]<stdout>[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,
0]<stdout[1,0]<stdout[1,0]<stdout>[1,0]<stdout>[1,0]<stdout>[1,0]<stdout>[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<std
out[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<s
tdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]
<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,
0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch 1/12
[1,0]<stdout>:97/97 [==============================] - 6s 20ms/step - loss: 1.8546 - accuracy: 0.3183<stdoutut[1,0]<stdout>[1,0]<s
tdout>[1,0]<stdout>[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,
0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 5.620890378952026
[1,0]<stdout>:Images/sec: 8895.39
[1,0]<stdout>:Epoch 2/12
[1,0]<stdout>:97/97 [==============================] - 1s 12ms/step - loss: 1.5189 - accuracy: 0.4450<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<st
dout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout
```

It is running now. So, here we are using another data set this particular data set is of CIFAR which is another data set and we are basically running it and you can see here we are using horovod -np 1 which means I am practically running it at rate as in on a single GPU.

(Refer Slide Time: 11:35)

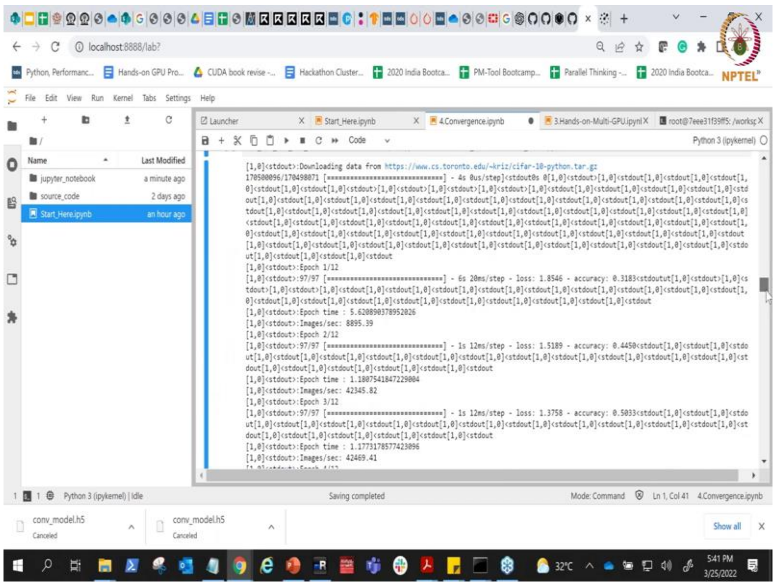


```
| N/A   34C    P0    42W / 300W |      3MiB / 16160MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   4  Tesla V100-SXM2...  On   | 00000000:85:00.0 Off |                    0 |
| N/A   34C    P0    43W / 300W |      3MiB / 16160MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   5  Tesla V100-SXM2...  On   | 00000000:86:00.0 Off |                    0 |
| N/A   35C    P0    44W / 300W |      3MiB / 16160MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   6  Tesla V100-SXM2...  On   | 00000000:89:00.0 Off |                    0 |
| N/A   36C    P0    42W / 300W |      3MiB / 16160MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   7  Tesla V100-SXM2...  On   | 00000000:8A:00.0 Off |                    0 |
| N/A   34C    P0    41W / 300W |      3MiB / 16160MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
+-----------------------------------------------------------------------------+
^Croot@7eee31f39ff5:/workspace# ^C
root@7eee31f39ff5:/workspace#
```

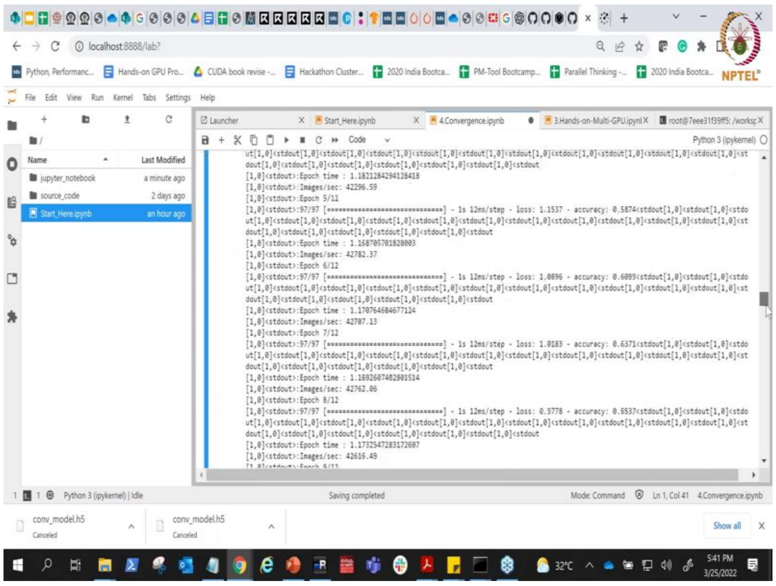
Because that single GPU is the one which we want to observe the accuracy levels ok.

(Refer Slide Time: 11:40)



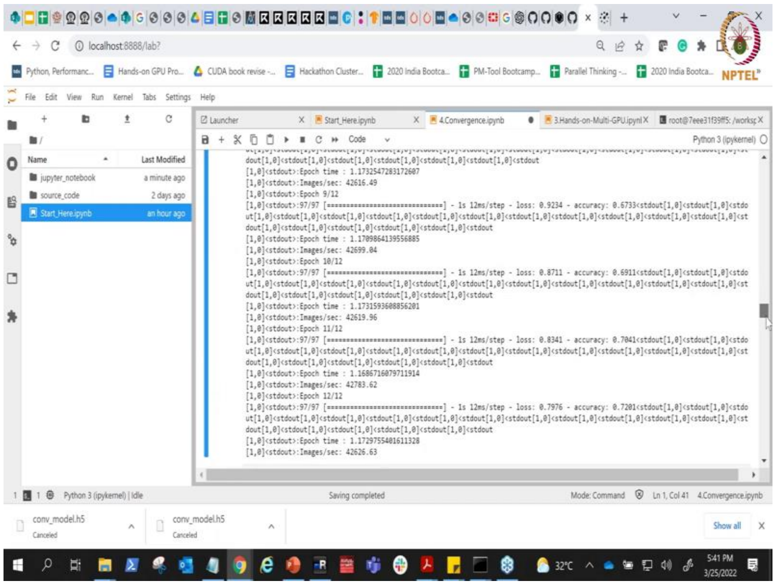
```
[1,0]<stdout>:Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170500096/170498071 [==============================] - 4s 0us/step]<stdout0s 0[1,0]<stdout>[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,
0]<stdout[1,0]<stdout[1,0]<stdout>[1,0]<stdout>[1,0]<stdout>[1,0]<stdout>[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<std
out[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<s
tdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]
<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,
0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch 1/12
[1,0]<stdout>:97/97 [==============================] - 6s 20ms/step - loss: 1.8546 - accuracy: 0.3183<stdoutut[1,0]<stdout>[1,0]<s
tdout>[1,0]<stdout>[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,
0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 5.620890378952026
[1,0]<stdout>:Images/sec: 8895.39
[1,0]<stdout>:Epoch 2/12
[1,0]<stdout>:97/97 [==============================] - 1s 12ms/step - loss: 1.5189 - accuracy: 0.4450<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<st
dout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 1.1807541847229004
[1,0]<stdout>:Images/sec: 42345.82
[1,0]<stdout>:Epoch 3/12
[1,0]<stdout>:97/97 [==============================] - 1s 12ms/step - loss: 1.3758 - accuracy: 0.5033<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<st
dout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 1.1773178577423096
[1,0]<stdout>:Images/sec: 42469.41
```

(Refer Slide Time: 11:40)



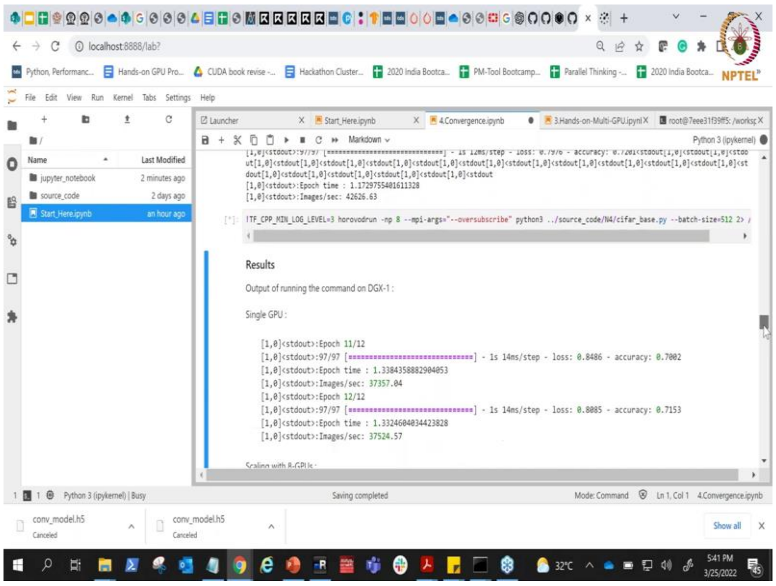
```
ut[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<st
dout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 1.1821284294128418
[1,0]<stdout>:Images/sec: 42296.59
[1,0]<stdout>:Epoch 5/12
[1,0]<stdout>:97/97 [==============================] - 1s 12ms/step - loss: 1.1537 - accuracy: 0.5874<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<st
dout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 1.168705701828003
[1,0]<stdout>:Images/sec: 42782.37
[1,0]<stdout>:Epoch 6/12
[1,0]<stdout>:97/97 [==============================] - 1s 12ms/step - loss: 1.0896 - accuracy: 0.6099<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<st
dout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 1.170764684677124
[1,0]<stdout>:Images/sec: 42707.13
[1,0]<stdout>:Epoch 7/12
[1,0]<stdout>:97/97 [==============================] - 1s 12ms/step - loss: 1.0183 - accuracy: 0.6371<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<st
dout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 1.1692607402801514
[1,0]<stdout>:Images/sec: 42762.06
[1,0]<stdout>:Epoch 8/12
[1,0]<stdout>:97/97 [==============================] - 1s 12ms/step - loss: 0.9778 - accuracy: 0.6537<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<st
dout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 1.1732547283172607
[1,0]<stdout>:Images/sec: 42616.49
```

(Refer Slide Time: 11:41)



```
dout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 1.1732547283172607
[1,0]<stdout>:Images/sec: 42616.49
[1,0]<stdout>:Epoch 9/12
[1,0]<stdout>:97/97 [==============================] - 1s 12ms/step - loss: 0.9234 - accuracy: 0.6733<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<st
dout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 1.1709864139556885
[1,0]<stdout>:Images/sec: 42699.04
[1,0]<stdout>:Epoch 10/12
[1,0]<stdout>:97/97 [==============================] - 1s 12ms/step - loss: 0.8711 - accuracy: 0.6911<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<st
dout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 1.1731593608856201
[1,0]<stdout>:Images/sec: 42619.96
[1,0]<stdout>:Epoch 11/12
[1,0]<stdout>:97/97 [==============================] - 1s 12ms/step - loss: 0.8341 - accuracy: 0.7041<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<st
dout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 1.1686716079711914
[1,0]<stdout>:Images/sec: 42783.62
[1,0]<stdout>:Epoch 12/12
[1,0]<stdout>:97/97 [==============================] - 1s 12ms/step - loss: 0.7976 - accuracy: 0.7201<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<st
dout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 1.1729755401611328
[1,0]<stdout>:Images/sec: 42626.63
```

(Refer Slide Time: 11:42)



```
ut[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<st
dout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 1.1729755401611328
[1,0]<stdout>:Images/sec: 42626.63
```

```
[*]: !TF_CPP_MIN_LOG_LEVEL=3 horovodrun -np 8 --mpi-args="--oversubscribe" python3 ../source_code/N4/cifar_base.py --batch-size=512 2> /
```

Results

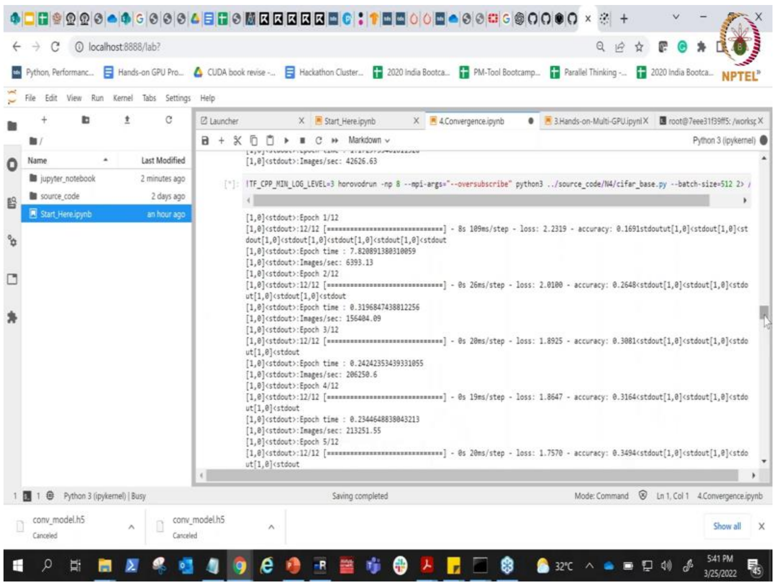
Output of running the command on DGX-1 :

Single GPU :

```
[1,0]<stdout>:Epoch 11/12
[1,0]<stdout>:97/97 [==============================] - 1s 14ms/step - loss: 0.8486 - accuracy: 0.7002
[1,0]<stdout>:Epoch time : 1.3384358882904053
[1,0]<stdout>:Images/sec: 37357.04
[1,0]<stdout>:Epoch 12/12
[1,0]<stdout>:97/97 [==============================] - 1s 14ms/step - loss: 0.8085 - accuracy: 0.7153
[1,0]<stdout>:Epoch time : 1.3324604034423828
[1,0]<stdout>:Images/sec: 37524.57
```

So, we have got the result for the single GPU the images per second is around 42 or so, let us see if you are able to run it on the 8 GPU also and most probably it should run if it does not run here I will just stick to the theory and we can see.

(Refer Slide Time: 12:10)



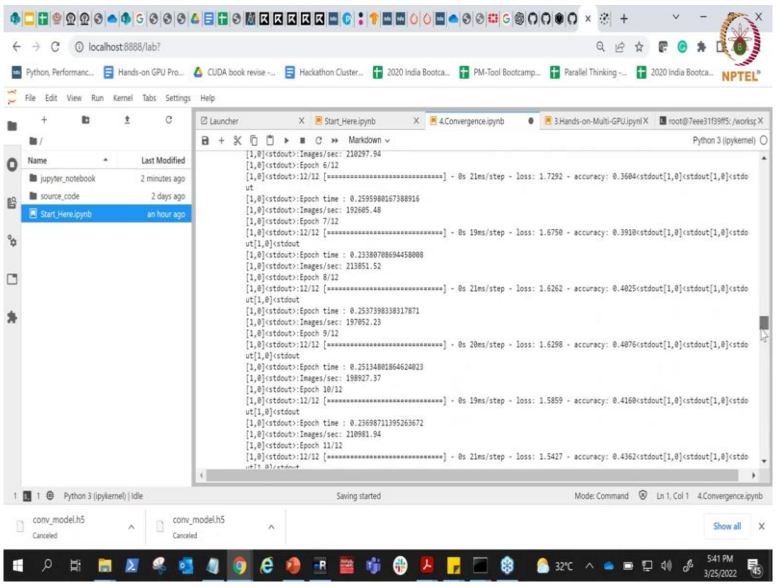
```
[1,0]<stdout>:Images/sec: 42626.63

[*]: !TF_CPP_MIN_LOG_LEVEL=3 horovodrun -np 8 --mpi-args="--oversubscribe" python3 ../source_code/N4/cifar_base.py --batch-size=512 2> /

[1,0]<stdout>:Epoch 1/12
[1,0]<stdout>:12/12 [==============================] - 8s 109ms/step - loss: 2.2319 - accuracy: 0.1691stdoutut[1,0]<stdout[1,0]<st
dout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 7.820891380310059
[1,0]<stdout>:Images/sec: 6393.13
[1,0]<stdout>:Epoch 2/12
[1,0]<stdout>:12/12 [==============================] - 0s 26ms/step - loss: 2.0100 - accuracy: 0.2648<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.3196847438812256
[1,0]<stdout>:Images/sec: 156404.09
[1,0]<stdout>:Epoch 3/12
[1,0]<stdout>:12/12 [==============================] - 0s 20ms/step - loss: 1.8925 - accuracy: 0.3081<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.24242353439331055
[1,0]<stdout>:Images/sec: 206250.6
[1,0]<stdout>:Epoch 4/12
[1,0]<stdout>:12/12 [==============================] - 0s 19ms/step - loss: 1.8647 - accuracy: 0.3164<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.2344648838043213
[1,0]<stdout>:Images/sec: 213251.55
[1,0]<stdout>:Epoch 5/12
[1,0]<stdout>:12/12 [==============================] - 0s 20ms/step - loss: 1.7570 - accuracy: 0.3494<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
```

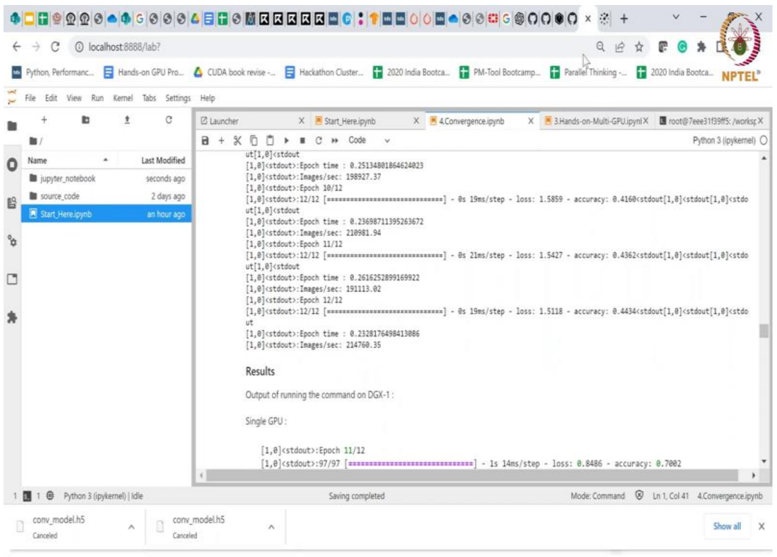
So, it started running. The most important thing for here also for us is the accuracy and that is what we want to look at.

(Refer Slide Time: 12:20)



```
[1,0]<stdout>:Images/sec: 210297.94
[1,0]<stdout>:Epoch 6/12
[1,0]<stdout>:12/12 [==============================] - 0s 21ms/step - loss: 1.7292 - accuracy: 0.3604<stdout[1,0]<stdout[1,0]<stdo
ut
[1,0]<stdout>:Epoch time : 0.2595980167388916
[1,0]<stdout>:Images/sec: 192605.48
[1,0]<stdout>:Epoch 7/12
[1,0]<stdout>:12/12 [==============================] - 0s 19ms/step - loss: 1.6750 - accuracy: 0.3910<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.23380708694458008
[1,0]<stdout>:Images/sec: 213851.52
[1,0]<stdout>:Epoch 8/12
[1,0]<stdout>:12/12 [==============================] - 0s 21ms/step - loss: 1.6262 - accuracy: 0.4025<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.2537398338317871
[1,0]<stdout>:Images/sec: 197052.23
[1,0]<stdout>:Epoch 9/12
[1,0]<stdout>:12/12 [==============================] - 0s 20ms/step - loss: 1.6298 - accuracy: 0.4076<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.25134801864624023
[1,0]<stdout>:Images/sec: 198927.37
[1,0]<stdout>:Epoch 10/12
[1,0]<stdout>:12/12 [==============================] - 0s 19ms/step - loss: 1.5859 - accuracy: 0.4160<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.23698711395263672
[1,0]<stdout>:Images/sec: 210981.94
[1,0]<stdout>:Epoch 11/12
[1,0]<stdout>:12/12 [==============================] - 0s 21ms/step - loss: 1.5427 - accuracy: 0.4362<stdout[1,0]<stdout[1,0]<stdo
```

(Refer Slide Time: 12:22)



```
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.25134801864624023
[1,0]<stdout>:Images/sec: 198927.37
[1,0]<stdout>:Epoch 10/12
[1,0]<stdout>:12/12 [==============================] - 0s 19ms/step - loss: 1.5859 - accuracy: 0.4160<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.23698711395263672
[1,0]<stdout>:Images/sec: 210981.94
[1,0]<stdout>:Epoch 11/12
[1,0]<stdout>:12/12 [==============================] - 0s 21ms/step - loss: 1.5427 - accuracy: 0.4362<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.2616252899169922
[1,0]<stdout>:Images/sec: 191113.02
[1,0]<stdout>:Epoch 12/12
[1,0]<stdout>:12/12 [==============================] - 0s 19ms/step - loss: 1.5118 - accuracy: 0.4434<stdout[1,0]<stdout[1,0]<stdo
ut
[1,0]<stdout>:Epoch time : 0.2328176498413086
[1,0]<stdout>:Images/sec: 214760.35
```

Results

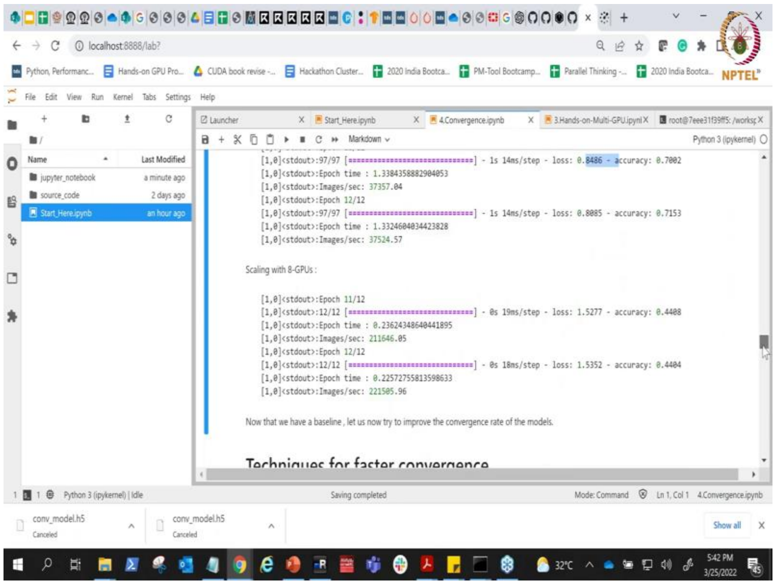
Output of running the command on DGX-1 :

Single GPU :

```
[1,0]<stdout>:Epoch 11/12
[1,0]<stdout>:97/97 [==============================] - 1s 14ms/step - loss: 0.8486 - accuracy: 0.7002
```

So, you can see here the accuracy for us after running on 12 epochs is 0.4434 while the accuracy here is 0.7201. So, the single GPU run gave 72 percent accuracy while the 8 GPU run gave around it gave me around only 42 percent accuracy. So, there is obviously, a loss which we describe sometime back which we can get and that is what is also highlighted that you are getting around 70 some or 80 percentage when you run it on single versus when you run it on multi GPU.

(Refer Slide Time: 13:08)



```
[1,0]<stdout>:97/97 [==============================] - 1s 14ms/step - loss: 0.8486 - accuracy: 0.7002
[1,0]<stdout>:Epoch time : 1.3384358882904053
[1,0]<stdout>:Images/sec: 37357.04
[1,0]<stdout>:Epoch 12/12
[1,0]<stdout>:97/97 [==============================] - 1s 14ms/step - loss: 0.8085 - accuracy: 0.7153
[1,0]<stdout>:Epoch time : 1.3324604034423828
[1,0]<stdout>:Images/sec: 37524.57
```

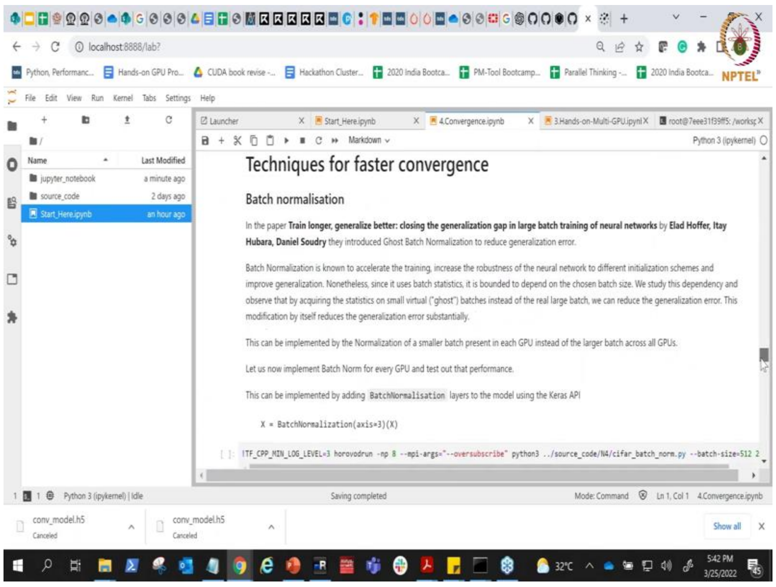
Scaling with 8-GPUs :

```
[1,0]<stdout>:Epoch 11/12
[1,0]<stdout>:12/12 [==============================] - 0s 19ms/step - loss: 1.5277 - accuracy: 0.4408
[1,0]<stdout>:Epoch time : 0.23624348640441895
[1,0]<stdout>:Images/sec: 211646.05
[1,0]<stdout>:Epoch 12/12
[1,0]<stdout>:12/12 [==============================] - 0s 18ms/step - loss: 1.5352 - accuracy: 0.4404
[1,0]<stdout>:Epoch time : 0.22572755813598633
[1,0]<stdout>:Images/sec: 221505.96
```

Now that we have a baseline , let us now try to improve the convergence rate of the models.

Techniques for faster convergence

(Refer Slide Time: 13:13)



Techniques for faster convergence

Batch normalisation

In the paper **Train longer, generalize better: closing the generalization gap in large batch training of neural networks** by **Elad Hoffer, Itay Hubara, Daniel Soudry** they introduced Ghost Batch Normalization to reduce generalization error.

Batch Normalization is known to accelerate the training, increase the robustness of the neural network to different initialization schemes and improve generalization. Nonetheless, since it uses batch statistics, it is bounded to depend on the chosen batch size. We study this dependency and observe that by acquiring the statistics on small virtual ("ghost") batches instead of the real large batch, we can reduce the generalization error. This modification by itself reduces the generalization error substantially.

This can be implemented by the Normalization of a smaller batch present in each GPU instead of the larger batch across all GPUs.

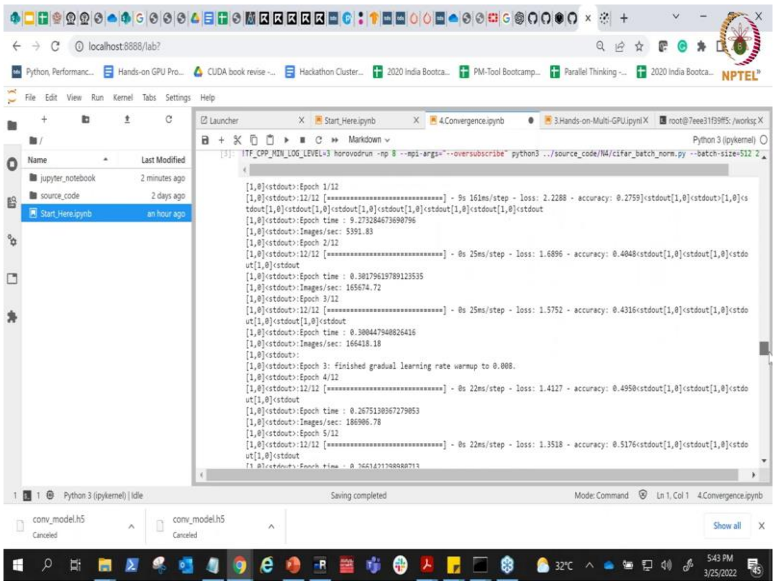
Let us now implement Batch Norm for every GPU and test out that performance.

This can be implemented by adding `BatchNormalisation` layers to the model using the Keras API

```
X = BatchNormalization(axis=3)(X)
```

```
[ ]: !TF_CPP_MIN_LOG_LEVEL=3 horovodrun -np 8 --mpi-args="--oversubscribe" python3 ../source_code/N4/cifar_batch_norm.py --batch-size=512 2
```

(Refer Slide Time: 13:20)



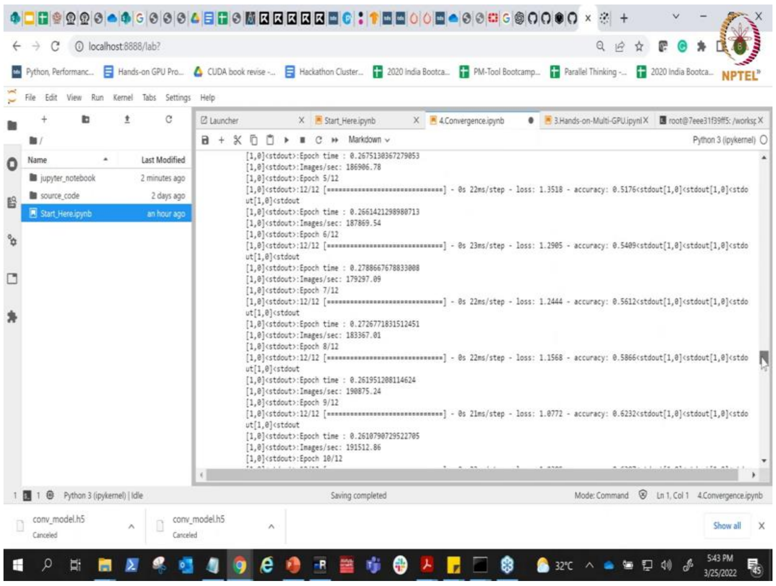
```
[3]: !TF_CPP_MIN_LOG_LEVEL=3 horovodrun -np 8 --mpi-args="--oversubscribe" python3 ../source_code/N4/cifar_batch_norm.py --batch-size=512 2

[1,0]<stdout>:Epoch 1/12
[1,0]<stdout>:12/12 [==============================] - 9s 161ms/step - loss: 2.2288 - accuracy: 0.2759]<stdout[1,0]<stdout>[1,0]<s
tdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 9.273284673690796
[1,0]<stdout>:Images/sec: 5391.83
[1,0]<stdout>:Epoch 2/12
[1,0]<stdout>:12/12 [==============================] - 0s 25ms/step - loss: 1.6896 - accuracy: 0.4048<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.30179619789123535
[1,0]<stdout>:Images/sec: 165674.72
[1,0]<stdout>:Epoch 3/12
[1,0]<stdout>:12/12 [==============================] - 0s 25ms/step - loss: 1.5752 - accuracy: 0.4316<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.300447940826416
[1,0]<stdout>:Images/sec: 166418.18
[1,0]<stdout>:
[1,0]<stdout>:Epoch 3: finished gradual learning rate warmup to 0.008.
[1,0]<stdout>:Epoch 4/12
[1,0]<stdout>:12/12 [==============================] - 0s 22ms/step - loss: 1.4127 - accuracy: 0.4950<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.2675130367279053
[1,0]<stdout>:Images/sec: 186906.78
[1,0]<stdout>:Epoch 5/12
[1,0]<stdout>:12/12 [==============================] - 0s 22ms/step - loss: 1.3518 - accuracy: 0.5176<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.2661421298980713
```

So, the first technique that we mentioned for faster convergence was the use of batch normalization and for the existing code we are adding batch normalization to the same code it just that we added one more line of batch normalization here to the same code. And let us say there is a improve in the accuracy by adding this batch normalization. We are running again on 8 GPUs please see that I am running again on 8 GPUs.

So, previously by running on 8 GPUs we were able to get higher images per second with 8 GPUs I got 2,14,000 images per second versus 42,000 images per second when I was running on a single GPU, but my accuracy reduced.

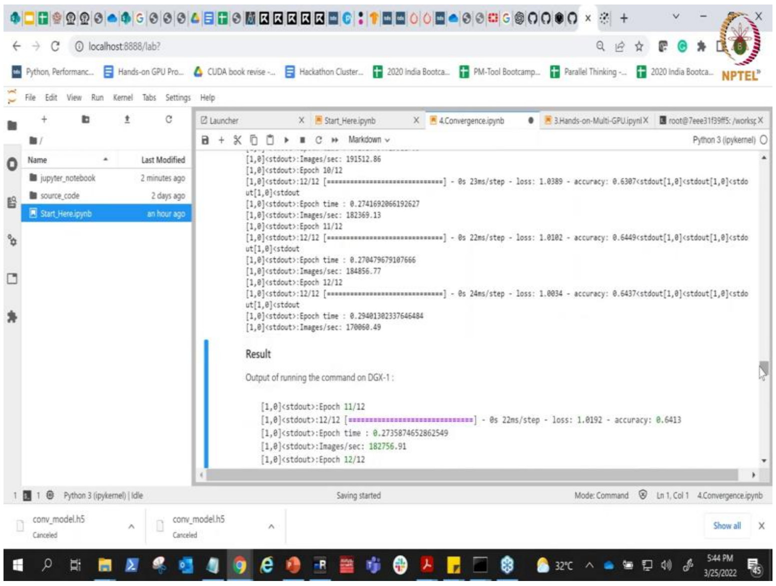
(Refer Slide Time: 14:14)



```
[1,0]<stdout>:Epoch time : 0.2675130367279053
[1,0]<stdout>:Images/sec: 186906.78
[1,0]<stdout>:Epoch 5/12
[1,0]<stdout>:12/12 [==============================] - 0s 22ms/step - loss: 1.3518 - accuracy: 0.5176<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.2661421298980713
[1,0]<stdout>:Images/sec: 187869.54
[1,0]<stdout>:Epoch 6/12
[1,0]<stdout>:12/12 [==============================] - 0s 23ms/step - loss: 1.2905 - accuracy: 0.5409<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.2788667678833008
[1,0]<stdout>:Images/sec: 179297.09
[1,0]<stdout>:Epoch 7/12
[1,0]<stdout>:12/12 [==============================] - 0s 22ms/step - loss: 1.2444 - accuracy: 0.5612<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.2726771831512451
[1,0]<stdout>:Images/sec: 183367.01
[1,0]<stdout>:Epoch 8/12
[1,0]<stdout>:12/12 [==============================] - 0s 22ms/step - loss: 1.1568 - accuracy: 0.5866<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.261951208114624
[1,0]<stdout>:Images/sec: 190875.24
[1,0]<stdout>:Epoch 9/12
[1,0]<stdout>:12/12 [==============================] - 0s 21ms/step - loss: 1.0772 - accuracy: 0.6232<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.2610790729522705
[1,0]<stdout>:Images/sec: 191512.86
[1,0]<stdout>:Epoch 10/12
```

So, it is it was of lesser use for me if my accuracy is lower then I have to run for larger epochs which means it is of it is of no use.

(Refer Slide Time: 14:18)



```
[1,0]<stdout>:Images/sec: 191512.86
[1,0]<stdout>:Epoch 10/12
[1,0]<stdout>:12/12 [==============================] - 0s 23ms/step - loss: 1.0389 - accuracy: 0.6307<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.2741692066192627
[1,0]<stdout>:Images/sec: 182369.13
[1,0]<stdout>:Epoch 11/12
[1,0]<stdout>:12/12 [==============================] - 0s 22ms/step - loss: 1.0102 - accuracy: 0.6449<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.270479679107666
[1,0]<stdout>:Images/sec: 184856.77
[1,0]<stdout>:Epoch 12/12
[1,0]<stdout>:12/12 [==============================] - 0s 24ms/step - loss: 1.0034 - accuracy: 0.6437<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.29401302337646484
[1,0]<stdout>:Images/sec: 170060.49
```

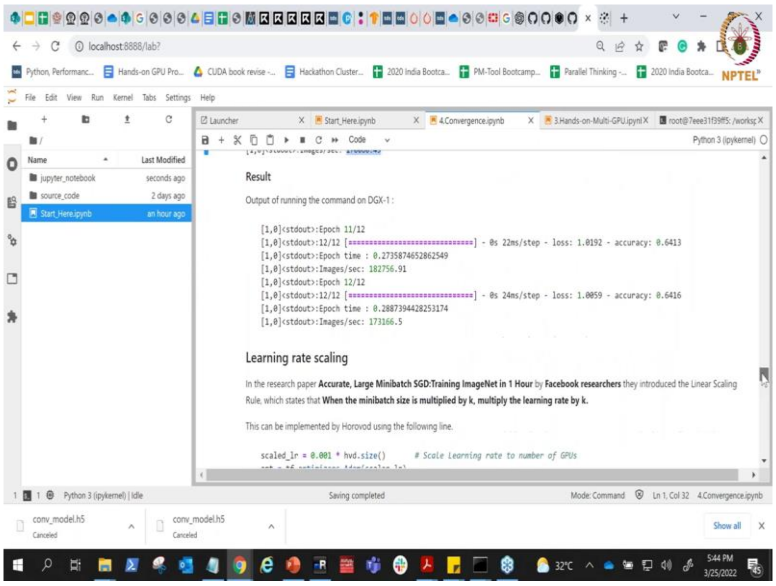
Result

Output of running the command on DGX-1 :

```
[1,0]<stdout>:Epoch 11/12
[1,0]<stdout>:12/12 [==============================] - 0s 22ms/step - loss: 1.0192 - accuracy: 0.6413
[1,0]<stdout>:Epoch time : 0.2735874652862549
[1,0]<stdout>:Images/sec: 182756.91
[1,0]<stdout>:Epoch 12/12
```

You can see here now the accuracy have increased from 40 to something in 64 by having the batch normalization. So, the by adding batch normalization my images per second decreased slightly, but I got more accuracy.

(Refer Slide Time: 14:40)



Result

Output of running the command on DGX-1 :

```
[1,0]<stdout>:Epoch 11/12
[1,0]<stdout>:12/12 [==============================] - 0s 22ms/step - loss: 1.0192 - accuracy: 0.6413
[1,0]<stdout>:Epoch time : 0.2735874652862549
[1,0]<stdout>:Images/sec: 182756.91
[1,0]<stdout>:Epoch 12/12
[1,0]<stdout>:12/12 [==============================] - 0s 24ms/step - loss: 1.0059 - accuracy: 0.6416
[1,0]<stdout>:Epoch time : 0.2887394428253174
[1,0]<stdout>:Images/sec: 173166.5
```

Learning rate scaling

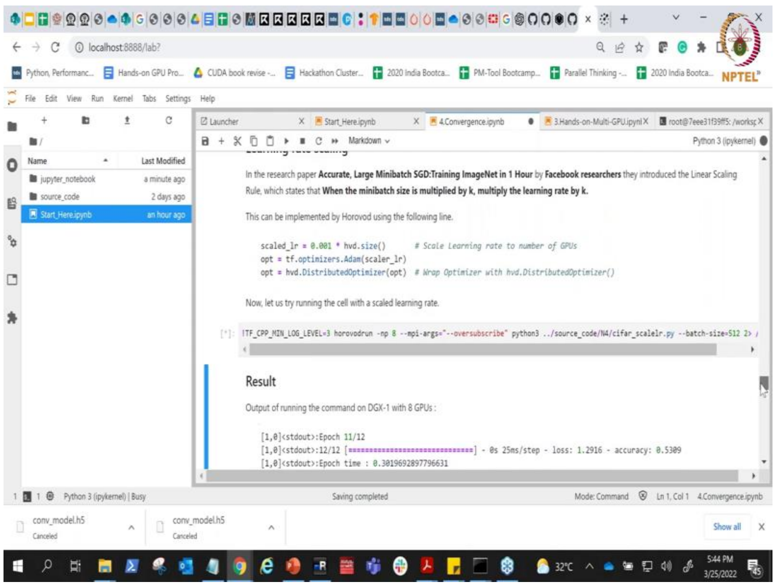
In the research paper **Accurate, Large Minibatch SGD:Training ImageNet in 1 Hour** by **Facebook researchers** they introduced the Linear Scaling Rule, which states that **When the minibatch size is multiplied by k, multiply the learning rate by k.**

This can be implemented by Horovod using the following line.

```
scaled_lr = 0.001 * hvd.size()       # Scale Learning rate to number of GPUs
```

So, that is another part which needs to be seen that by adding some of these techniques you might end up reducing your overall throughput, but you what you are trying to find is the right balance of accuracy how many epochs you need right.

(Refer Slide Time: 14:52)



In the research paper **Accurate, Large Minibatch SGD:Training ImageNet in 1 Hour** by **Facebook researchers** they introduced the Linear Scaling Rule, which states that **When the minibatch size is multiplied by k, multiply the learning rate by k.**

This can be implemented by Horovod using the following line.

```
scaled_lr = 0.001 * hvd.size()       # Scale Learning rate to number of GPUs
opt = tf.optimizers.Adam(scaler_lr)
opt = hvd.DistributedOptimizer(opt)  # Wrap Optimizer with hvd.DistributedOptimizer()
```

Now, let us try running the cell with a scaled learning rate.

```
[*]: !TF_CPP_MIN_LOG_LEVEL=3 horovodrun -np 8 --mpi-args="--oversubscribe" python3 ../source_code/N4/cifar_scalelr.py --batch-size=512 2>
```

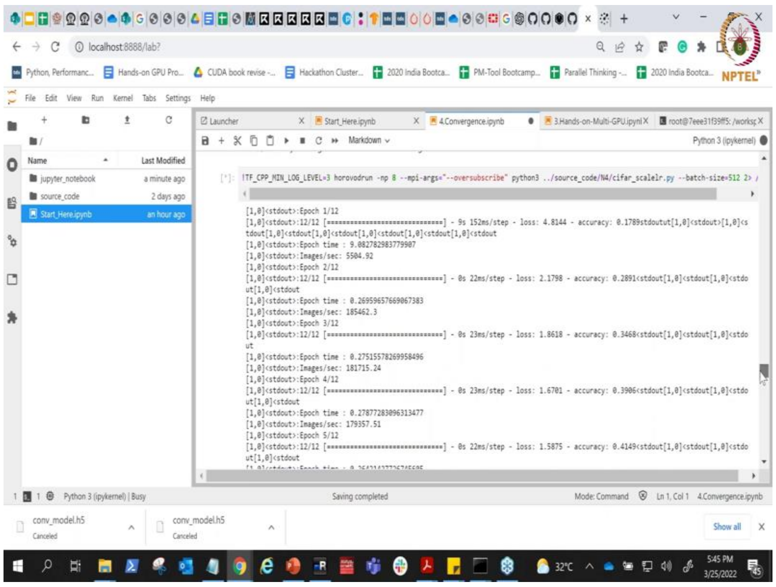
Result

Output of running the command on DGX-1 with 8 GPUs :

```
[1,0]<stdout>:Epoch 11/12
[1,0]<stdout>:12/12 [==============================] - 0s 25ms/step - loss: 1.2916 - accuracy: 0.5309
[1,0]<stdout>:Epoch time : 0.3019692897796631
```

Then the next thing that we also talked about is changing the learning rate and you can see here we are trying to change the learning rate based on the; based on the number of GPUs here you are seeing that I am multiplying it by 0.01 and we are going to change the learning rate and not keep it make it adaptive rather than keeping it constant.

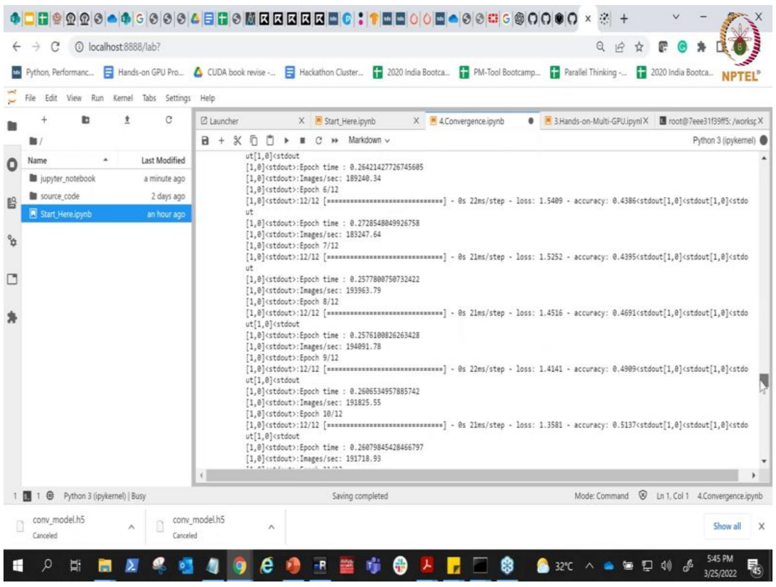
(Refer Slide Time: 15:17)



```
!TF_CPP_MIN_LOG_LEVEL=3 horovodrun -np 8 --mpi-args="--oversubscribe" python3 ../source_code/N4/cifar_scalelr.py --batch-size=512 2> /
```

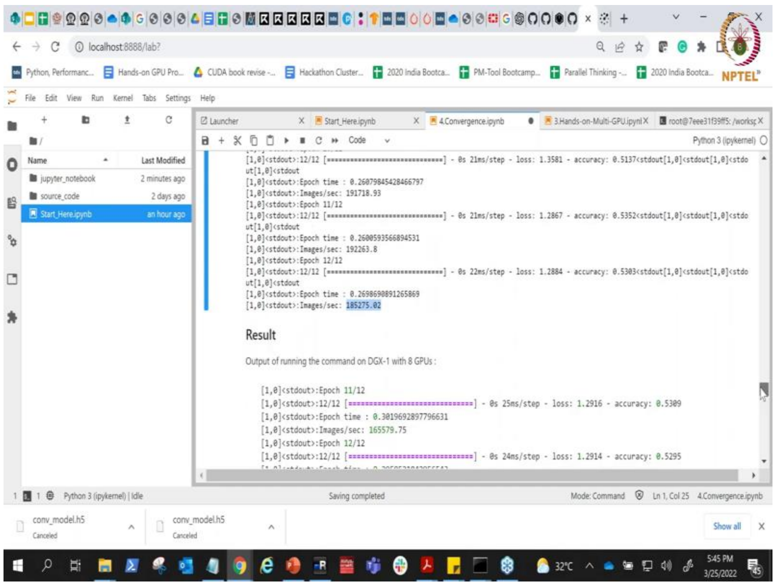
```
[1,0]<stdout>:Epoch 1/12
[1,0]<stdout>:12/12 [==============================] - 9s 152ms/step - loss: 4.8144 - accuracy: 0.1789stdoutut[1,0]<stdout>[1,0]<s
tdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 9.082782983779907
[1,0]<stdout>:Images/sec: 5504.92
[1,0]<stdout>:Epoch 2/12
[1,0]<stdout>:12/12 [==============================] - 0s 22ms/step - loss: 2.1798 - accuracy: 0.2891<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.26959657669067383
[1,0]<stdout>:Images/sec: 185462.3
[1,0]<stdout>:Epoch 3/12
[1,0]<stdout>:12/12 [==============================] - 0s 23ms/step - loss: 1.8618 - accuracy: 0.3468<stdout[1,0]<stdout[1,0]<stdo
ut
[1,0]<stdout>:Epoch time : 0.27515578269958496
[1,0]<stdout>:Images/sec: 181715.24
[1,0]<stdout>:Epoch 4/12
[1,0]<stdout>:12/12 [==============================] - 0s 23ms/step - loss: 1.6701 - accuracy: 0.3906<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.27877283096313477
[1,0]<stdout>:Images/sec: 179357.51
[1,0]<stdout>:Epoch 5/12
[1,0]<stdout>:12/12 [==============================] - 0s 22ms/step - loss: 1.5875 - accuracy: 0.4149<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
```

(Refer Slide Time: 15:35)



```
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.26421427726745605
[1,0]<stdout>:Images/sec: 189240.34
[1,0]<stdout>:Epoch 6/12
[1,0]<stdout>:12/12 [==============================] - 0s 22ms/step - loss: 1.5409 - accuracy: 0.4386<stdout[1,0]<stdout[1,0]<stdo
ut
[1,0]<stdout>:Epoch time : 0.2728548049926758
[1,0]<stdout>:Images/sec: 183247.64
[1,0]<stdout>:Epoch 7/12
[1,0]<stdout>:12/12 [==============================] - 0s 21ms/step - loss: 1.5252 - accuracy: 0.4395<stdout[1,0]<stdout[1,0]<stdo
ut
[1,0]<stdout>:Epoch time : 0.2577800750732422
[1,0]<stdout>:Images/sec: 193963.79
[1,0]<stdout>:Epoch 8/12
[1,0]<stdout>:12/12 [==============================] - 0s 21ms/step - loss: 1.4516 - accuracy: 0.4691<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.2576100826263428
[1,0]<stdout>:Images/sec: 194091.78
[1,0]<stdout>:Epoch 9/12
[1,0]<stdout>:12/12 [==============================] - 0s 22ms/step - loss: 1.4141 - accuracy: 0.4909<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.2606534957885742
[1,0]<stdout>:Images/sec: 191825.55
[1,0]<stdout>:Epoch 10/12
[1,0]<stdout>:12/12 [==============================] - 0s 21ms/step - loss: 1.3581 - accuracy: 0.5137<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.2607984542846679 7
[1,0]<stdout>:Images/sec: 191718.93
```

(Refer Slide Time: 15:38)



```
[1,0]<stdout>:12/12 [==============================] - 0s 21ms/step - loss: 1.3581 - accuracy: 0.5137<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.26079845428466797
[1,0]<stdout>:Images/sec: 191718.93
[1,0]<stdout>:Epoch 11/12
[1,0]<stdout>:12/12 [==============================] - 0s 21ms/step - loss: 1.2867 - accuracy: 0.5352<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.2600593566894531
[1,0]<stdout>:Images/sec: 192263.8
[1,0]<stdout>:Epoch 12/12
[1,0]<stdout>:12/12 [==============================] - 0s 22ms/step - loss: 1.2884 - accuracy: 0.5303<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.2698690891265869
[1,0]<stdout>:Images/sec: 185275.02
```

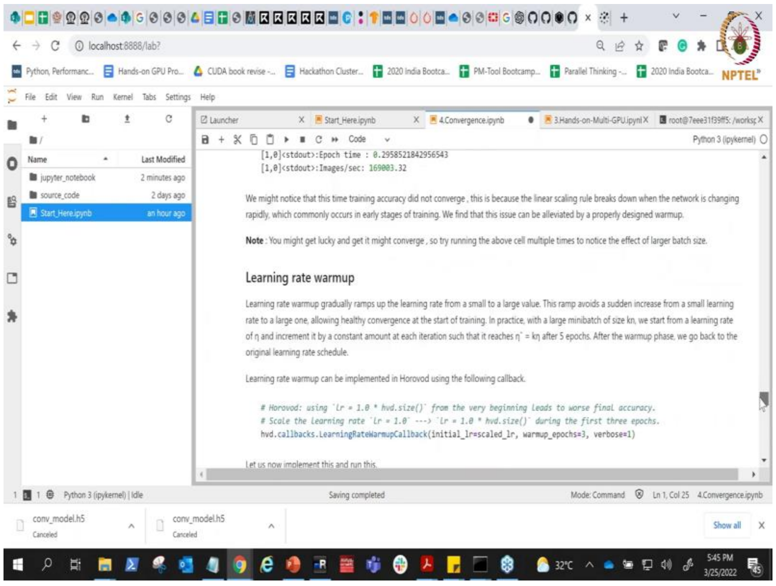
Result

Output of running the command on DGX-1 with 8 GPUs :

```
[1,0]<stdout>:Epoch 11/12
[1,0]<stdout>:12/12 [==============================] - 0s 25ms/step - loss: 1.2916 - accuracy: 0.5309
[1,0]<stdout>:Epoch time : 0.3019692897796631
[1,0]<stdout>:Images/sec: 165579.75
[1,0]<stdout>:Epoch 12/12
[1,0]<stdout>:12/12 [==============================] - 0s 24ms/step - loss: 1.2914 - accuracy: 0.5295
```

So, once we do that let us see if there is improve in accuracy or reduce in accuracy not so much. So, we can see here the accuracy increase was better than the better than the normal one. So, the learning rate did increase it, but batch normalization basically had a higher impact as compared to changing the learning rate. So, the learning rate increased from for 0.46 to 0.53 in 12 epochs and you can see here this is also really good, but still it is lesser than the learning rate.

(Refer Slide Time: 16:08)



```
[1,0]<stdout>:Epoch time : 0.2958521842956543
[1,0]<stdout>:Images/sec: 169003.32
```

We might notice that this time training accuracy did not converge , this is because the linear scaling rule breaks down when the network is changing rapidly, which commonly occurs in early stages of training. We find that this issue can be alleviated by a properly designed warmup.

Note : You might get lucky and get it might converge , so try running the above cell multiple times to notice the effect of larger batch size.

Learning rate warmup

Learning rate warmup gradually ramps up the learning rate from a small to a large value. This ramp avoids a sudden increase from a small learning rate to a large one, allowing healthy convergence at the start of training. In practice, with a large minibatch of size kn, we start from a learning rate of η and increment it by a constant amount at each iteration such that it reaches η^ = kη after 5 epochs. After the warmup phase, we go back to the original learning rate schedule.

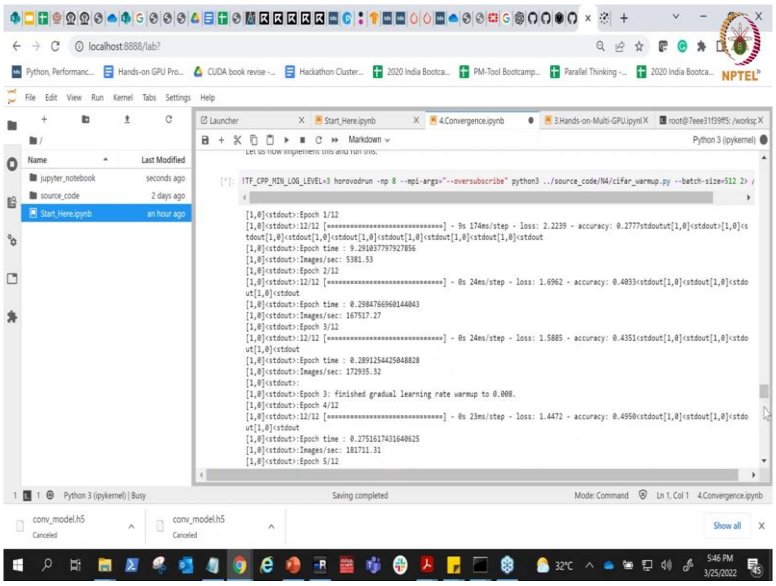
Learning rate warmup can be implemented in Horovod using the following callback.

```
# Horovod: using `lr = 1.0 * hvd.size()` from the very beginning leads to worse final accuracy.
# Scale the learning rate `lr = 1.0` ---> `lr = 1.0 * hvd.size()` during the first three epochs.
hvd.callbacks.LearningRateWarmupCallback(initial_lr=scaled_lr, warmup_epochs=3, verbose=1)
```

Let us now implement this and run this.

So, we talked about the problem with learning rate which was the initial warm up which can create these kinds of problems.

(Refer Slide Time: 16:15)



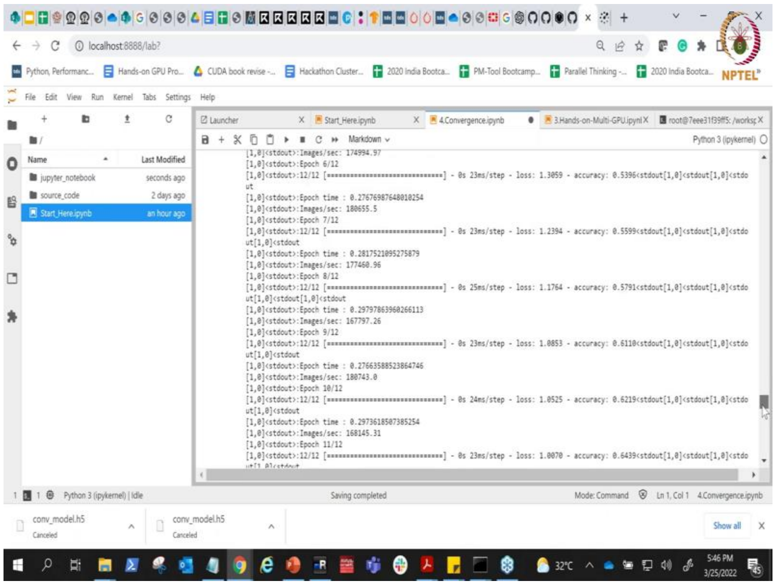
```
[*]: !TF_CPP_MIN_LOG_LEVEL=3 horovodrun -np 8 --mpi-args="--oversubscribe" python3 ../source_code/N4/cifar_warmup.py --batch-size=512 2> /

[1,0]<stdout>:Epoch 1/12
[1,0]<stdout>:12/12 [==============================] - 9s 174ms/step - loss: 2.2239 - accuracy: 0.2777stdoutut[1,0]<stdout>[1,0]<s
tdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 9.291037797927856
[1,0]<stdout>:Images/sec: 5381.53
[1,0]<stdout>:Epoch 2/12
[1,0]<stdout>:12/12 [==============================] - 0s 24ms/step - loss: 1.6962 - accuracy: 0.4033<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.2984766960144043
[1,0]<stdout>:Images/sec: 167517.27
[1,0]<stdout>:Epoch 3/12
[1,0]<stdout>:12/12 [==============================] - 0s 24ms/step - loss: 1.5885 - accuracy: 0.4351<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.2891254425048828
[1,0]<stdout>:Images/sec: 172935.32
[1,0]<stdout>:
[1,0]<stdout>:Epoch 3: finished gradual learning rate warmup to 0.008.
[1,0]<stdout>:Epoch 4/12
[1,0]<stdout>:12/12 [==============================] - 0s 23ms/step - loss: 1.4472 - accuracy: 0.4950<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.2751617431640625
[1,0]<stdout>:Images/sec: 181711.31
[1,0]<stdout>:Epoch 5/12
```

To solve that you can pass parameters like Horovod dot callback and you can do a learning rate warm up and you can define that what is your warm up till how much do you want to warm up and here you can see we are saying for 3 epochs should be considered as warm ups and then later on you can do the scaling of your of the learning rate.

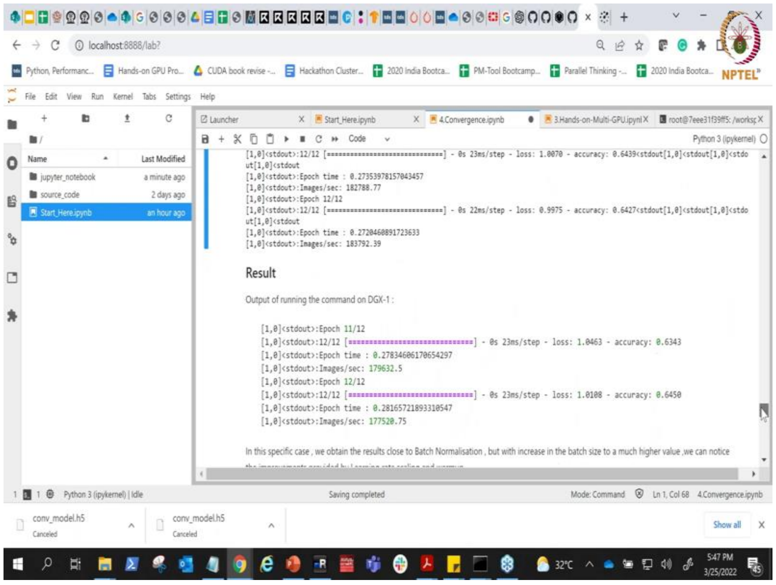
And let us see if it does have a impact on the accuracy. So, the normal learning rate was giving me around 53 percent accuracy.

(Refer Slide Time: 17:01)



```
[1,0]<stdout>:Images/sec: 174994.97
[1,0]<stdout>:Epoch 6/12
[1,0]<stdout>:12/12 [==============================] - 0s 23ms/step - loss: 1.3059 - accuracy: 0.5396<stdout[1,0]<stdout[1,0]<stdo
ut
[1,0]<stdout>:Epoch time : 0.27676987648010254
[1,0]<stdout>:Images/sec: 180655.5
[1,0]<stdout>:Epoch 7/12
[1,0]<stdout>:12/12 [==============================] - 0s 23ms/step - loss: 1.2394 - accuracy: 0.5599<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.2817521095275879
[1,0]<stdout>:Images/sec: 177460.96
[1,0]<stdout>:Epoch 8/12
[1,0]<stdout>:12/12 [==============================] - 0s 25ms/step - loss: 1.1764 - accuracy: 0.5791<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.29797863960266113
[1,0]<stdout>:Images/sec: 167797.26
[1,0]<stdout>:Epoch 9/12
[1,0]<stdout>:12/12 [==============================] - 0s 23ms/step - loss: 1.0853 - accuracy: 0.6110<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.27663588523864746
[1,0]<stdout>:Images/sec: 180743.0
[1,0]<stdout>:Epoch 10/12
[1,0]<stdout>:12/12 [==============================] - 0s 24ms/step - loss: 1.0525 - accuracy: 0.6219<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.2973618507385254
[1,0]<stdout>:Images/sec: 168145.31
[1,0]<stdout>:Epoch 11/12
[1,0]<stdout>:12/12 [==============================] - 0s 23ms/step - loss: 1.0070 - accuracy: 0.6439<stdout[1,0]<stdout[1,0]<stdo
```

(Refer Slide Time: 17:04)



```
[1,0]<stdout>:12/12 [==============================] - 0s 23ms/step - loss: 1.0070 - accuracy: 0.6439<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.27353978157043457
[1,0]<stdout>:Images/sec: 182788.77
[1,0]<stdout>:Epoch 12/12
[1,0]<stdout>:12/12 [==============================] - 0s 22ms/step - loss: 0.9975 - accuracy: 0.6427<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.2720460891723633
[1,0]<stdout>:Images/sec: 183792.39
```

Result

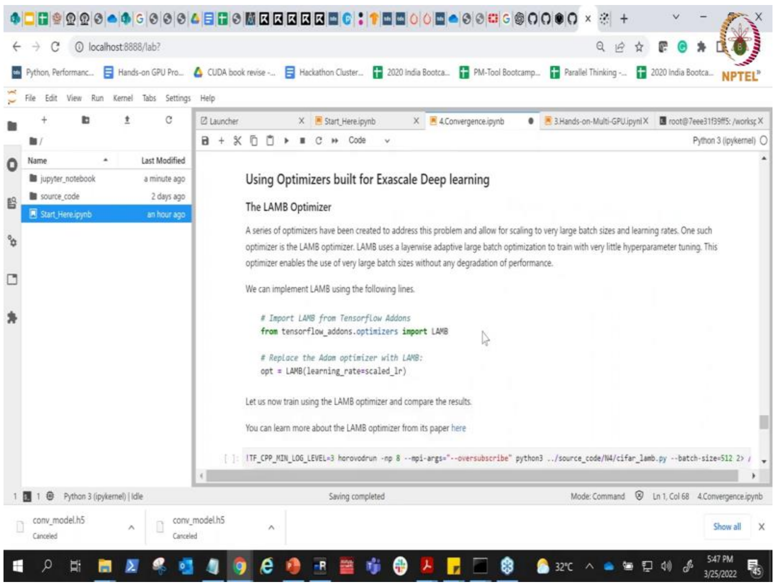
Output of running the command on DGX-1 :

```
[1,0]<stdout>:Epoch 11/12
[1,0]<stdout>:12/12 [==============================] - 0s 23ms/step - loss: 1.0463 - accuracy: 0.6343
[1,0]<stdout>:Epoch time : 0.27834606170654297
[1,0]<stdout>:Images/sec: 179632.5
[1,0]<stdout>:Epoch 12/12
[1,0]<stdout>:12/12 [==============================] - 0s 23ms/step - loss: 1.0108 - accuracy: 0.6450
[1,0]<stdout>:Epoch time : 0.28165721893310547
[1,0]<stdout>:Images/sec: 177520.75
```

In this specific case , we obtain the results close to Batch Normalisation , but with increase in the batch size to a much higher value ,we can notice

So, you can see here again it has increased it has come back to 64 percent as compared to 46 and my images per second is also is better. So, you can see here this impact of these hyper parameters is quite a lot when you move towards distributed deep learning and if you do not do enough research and if you do not use some of these techniques.

(Refer Slide Time: 17:26)



Using Optimizers built for Exascale Deep learning

The LAMB Optimizer

A series of optimizers have been created to address this problem and allow for scaling to very large batch sizes and learning rates. One such optimizer is the LAMB optimizer. LAMB uses a layerwise adaptive large batch optimization to train with very little hyperparameter tuning. This optimizer enables the use of very large batch sizes without any degradation of performance.

We can implement LAMB using the following lines.

```
# Import LAMB from Tensorflow Addons
from tensorflow_addons.optimizers import LAMB

# Replace the Adam optimizer with LAMB:
opt = LAMB(learning_rate=scaled_lr)
```

Let us now train using the LAMB optimizer and compare the results.

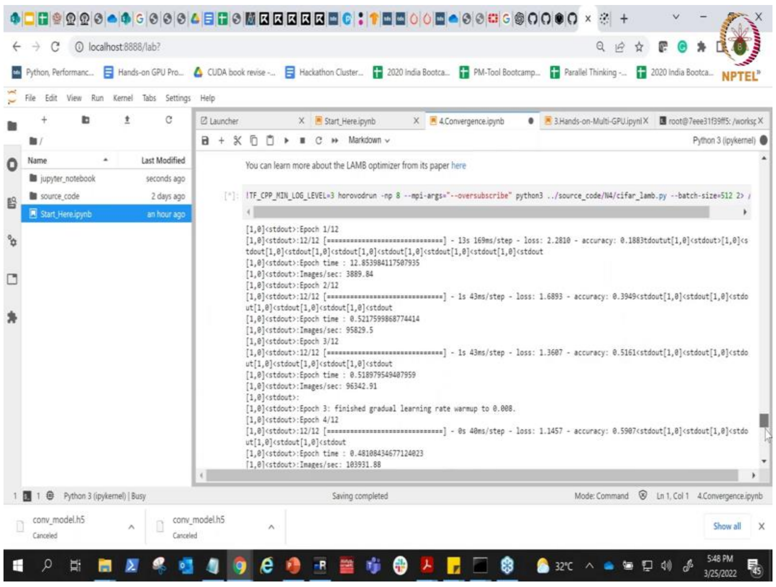
You can learn more about the LAMB optimizer from its paper here

```
[ ]: !TF_CPP_MIN_LOG_LEVEL=3 horovodrun -np 8 --mpi-args="--oversubscribe" python3 ../source_code/N4/cifar_lamb.py --batch-size=512 2> /
```

You might end up using all the GPUs, but then you will have to use larger epochs to converge and then all of your effort in terms of reducing the time just goes away. Last, but not the least basically I am I also talked about the uses of different kinds of optimizers which are meant for these kinds of higher level of scaling like here there is a different optimizer which is called as a lamb optimizer which uses a layerwise adaptive large batch optimization to train.

And it is well known because it requires very less or little hyper parameter tuning which is very good and you can enable very large batch sizes in this without having to degrade any performance. So, rather than using the normal ones you can see here you can use this optimizer called as lamb and let us see if we are what kind of a accuracy levels we can get via using this new optimizer.

(Refer Slide Time: 18:13)

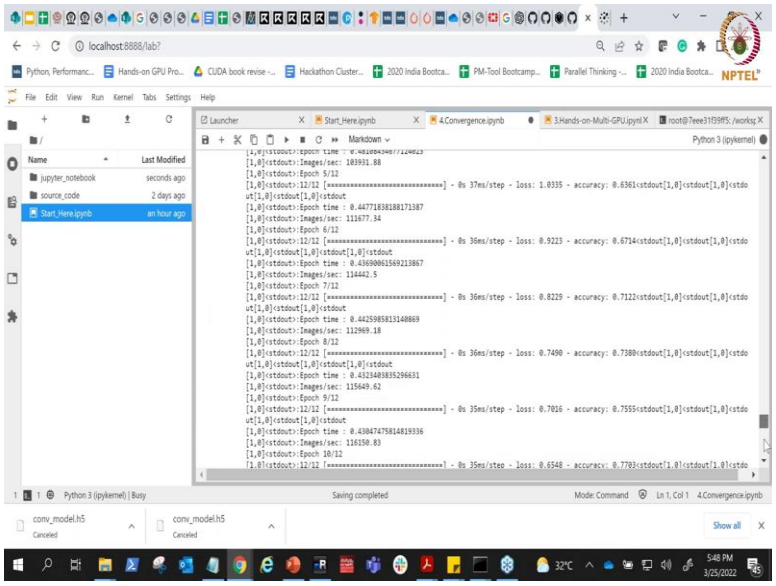


You can learn more about the LAMB optimizer from its paper here

```
[*]: !TF_CPP_MIN_LOG_LEVEL=3 horovodrun -np 8 --mpi-args="--oversubscribe" python3 ../source_code/N4/cifar_lamb.py --batch-size=512 2> /
```

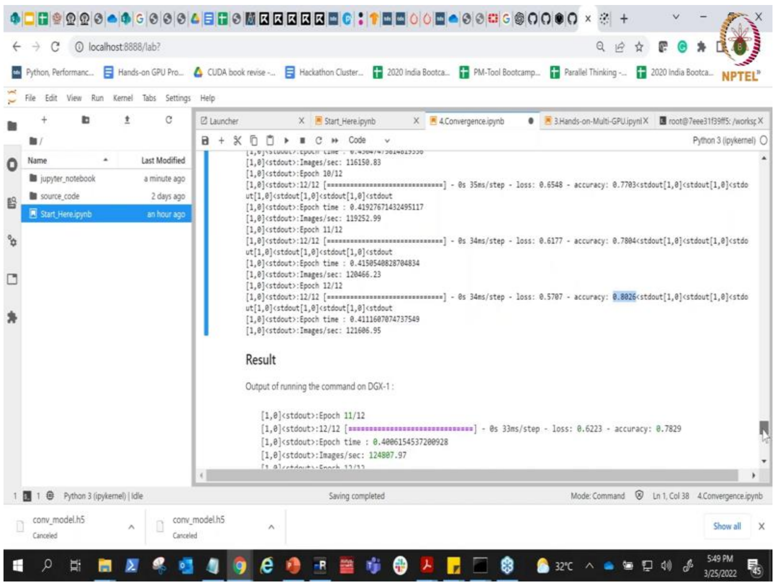
```
[1,0]<stdout>:Epoch 1/12
[1,0]<stdout>:12/12 [==============================] - 13s 169ms/step - loss: 2.2810 - accuracy: 0.1883tdoutut[1,0]<stdout>[1,0]<s
tdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 12.853984117507935
[1,0]<stdout>:Images/sec: 3889.84
[1,0]<stdout>:Epoch 2/12
[1,0]<stdout>:12/12 [==============================] - 1s 43ms/step - loss: 1.6893 - accuracy: 0.3949<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.5217599868774414
[1,0]<stdout>:Images/sec: 95829.5
[1,0]<stdout>:Epoch 3/12
[1,0]<stdout>:12/12 [==============================] - 1s 43ms/step - loss: 1.3607 - accuracy: 0.5161<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.518979549407959
[1,0]<stdout>:Images/sec: 96342.91
[1,0]<stdout>:
[1,0]<stdout>:Epoch 3: finished gradual learning rate warmup to 0.008.
[1,0]<stdout>:Epoch 4/12
[1,0]<stdout>:12/12 [==============================] - 0s 40ms/step - loss: 1.1457 - accuracy: 0.5907<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.48108434677124023
[1,0]<stdout>:Images/sec: 103931.88
```

(Refer Slide Time: 18:49)



```
[1,0]<stdout>:Images/sec: 103931.88
[1,0]<stdout>:Epoch 5/12
[1,0]<stdout>:12/12 [==============================] - 0s 37ms/step - loss: 1.0335 - accuracy: 0.6361<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.44771838188171387
[1,0]<stdout>:Images/sec: 111677.34
[1,0]<stdout>:Epoch 6/12
[1,0]<stdout>:12/12 [==============================] - 0s 36ms/step - loss: 0.9223 - accuracy: 0.6714<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.43690061569213867
[1,0]<stdout>:Images/sec: 114442.5
[1,0]<stdout>:Epoch 7/12
[1,0]<stdout>:12/12 [==============================] - 0s 36ms/step - loss: 0.8229 - accuracy: 0.7122<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.4425985813140869
[1,0]<stdout>:Images/sec: 112969.18
[1,0]<stdout>:Epoch 8/12
[1,0]<stdout>:12/12 [==============================] - 0s 36ms/step - loss: 0.7490 - accuracy: 0.7380<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.4323403835296631
[1,0]<stdout>:Images/sec: 115649.62
[1,0]<stdout>:Epoch 9/12
[1,0]<stdout>:12/12 [==============================] - 0s 35ms/step - loss: 0.7016 - accuracy: 0.7555<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.43047475814819336
[1,0]<stdout>:Images/sec: 116150.83
[1,0]<stdout>:Epoch 10/12
[1,0]<stdout>:12/12 [==============================] - 0s 35ms/step - loss: 0.6548 - accuracy: 0.7703<stdout[1,0]<stdout[1,0]<stdo
```

(Refer Slide Time: 18:52)



```
[1,0]<stdout>:Images/sec: 116150.83
[1,0]<stdout>:Epoch 10/12
[1,0]<stdout>:12/12 [==============================] - 0s 35ms/step - loss: 0.6548 - accuracy: 0.7703<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.41927671432495117
[1,0]<stdout>:Images/sec: 119252.99
[1,0]<stdout>:Epoch 11/12
[1,0]<stdout>:12/12 [==============================] - 0s 34ms/step - loss: 0.6177 - accuracy: 0.7804<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.41505408287004834
[1,0]<stdout>:Images/sec: 120466.23
[1,0]<stdout>:Epoch 12/12
[1,0]<stdout>:12/12 [==============================] - 0s 34ms/step - loss: 0.5707 - accuracy: 0.8026<stdout[1,0]<stdout[1,0]<stdo
ut[1,0]<stdout[1,0]<stdout[1,0]<stdout
[1,0]<stdout>:Epoch time : 0.4111607074737549
[1,0]<stdout>:Images/sec: 121606.95
```

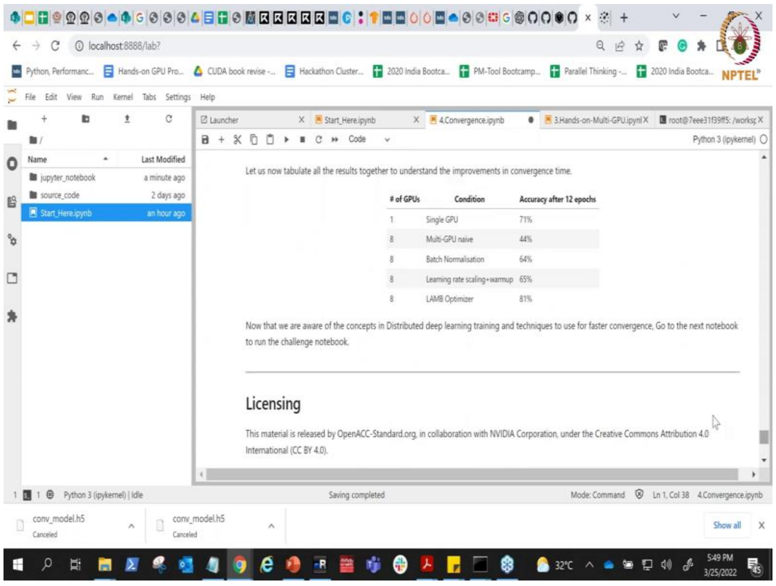
Result

Output of running the command on DGX-1 :

```
[1,0]<stdout>:Epoch 11/12
[1,0]<stdout>:12/12 [==============================] - 0s 33ms/step - loss: 0.6223 - accuracy: 0.7829
[1,0]<stdout>:Epoch time : 0.4006154537200928
[1,0]<stdout>:Images/sec: 124807.97
```

So, you can see here it is actually surpassed the capability of our traditional model also you can see here it has reached almost 80 percent accuracy which is very good it has surpass the normal optimizers and it was able to scale also not at the rate otherwise, but it was able to do many more instead of 46,000 it was able to do 1,21,000 images per second which is almost like 3 x speedup and, but also maintaining the accuracy level that you would have expected out of it.

(Refer Slide Time: 19:28)



Let us now tabulate all the results together to understand the improvements in convergence time.

# of GPUs	Condition	Accuracy after 12 epochs
1	Single GPU	71%
8	Multi-GPU naive	44%
8	Batch Normalisation	64%
8	Learning rate scaling+warmup	65%
8	LAMB Optimizer	81%

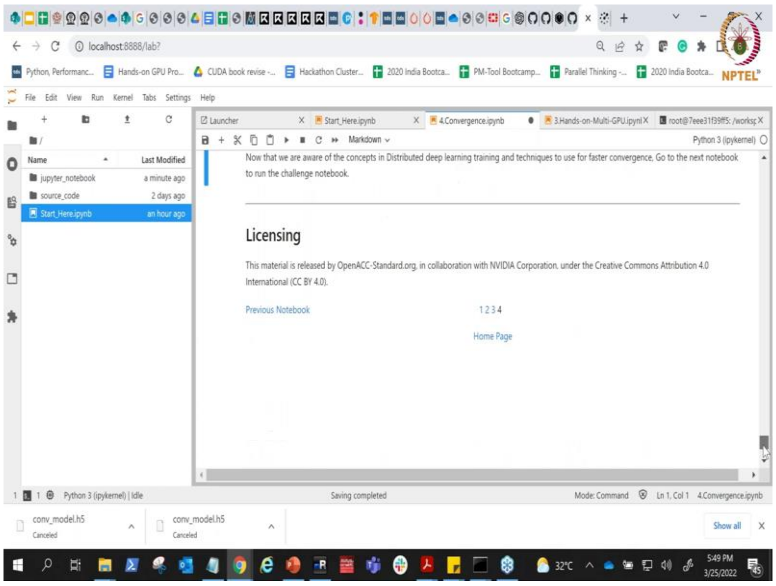
Now that we are aware of the concepts in Distributed deep learning training and techniques to use for faster convergence, Go to the next notebook to run the challenge notebook.

Licensing

This material is released by OpenACC-Standard.org, in collaboration with NVIDIA Corporation, under the Creative Commons Attribution 4.0 International (CC BY 4.0).

So, if you see we started with single GPU which gave around 70 percent we use a naive implementation of multi-GPU which reduced it, but then we use techniques like batch normalization learning rate with warm up and then finally, we also looked at certain additional optimizers which are much more suited when we talk about distributed deep learning.

(Refer Slide Time: 19:50)



Now that we are aware of the concepts in Distributed deep learning training and techniques to use for faster convergence, Go to the next notebook to run the challenge notebook.

Licensing

This material is released by OpenACC-Standard.org, in collaboration with NVIDIA Corporation, under the Creative Commons Attribution 4.0 International (CC BY 4.0).

Previous Notebook 1 2 3 4

Home Page

So, with that we are kind of done with the three series lecture that we were having on the distributed deep learning and a just a very quick recap in terms of what all things which we did, today with the increase in data set and the model sizes its almost impossible to be productive without having to use processes like GPU, but not just one processes, but many of them to make sure that we are basically working on research and not just waiting for it to finish for multiple days or weeks or months.

With that comes the challenge of different kinds of a frameworks needs to support this distribution, we saw many frameworks supporting it we saw example of running Horovod and TensorFlow Pytorch also supports it and we also saw the impact of running this frameworks on different sets of topology on different sets of hardware and what kind of a impact it can have we also demonstrated to you how the GPU computing infrastructure is not about the hardware alone.

Behind the scenes, the frameworks like TensorFlow Pytorch and all they use a highly optimized library like NCCL to do all of this effective communication in a very very low latency high bandwidth manner. We also saw how it can choose really high speed bandwidth or a low latency interconnects like NVLink and you optimize your scaling efficiency.

Finally, we move towards even if you do all of this there are higher chances that you may end up not getting the accuracy that you are expecting out of your code for which you need to know additional techniques that we saw like batch normalization learning rate warm up and also using other kinds of optimizers which are traditionally not used when you are running it on a non distributed environment. With that we are done with the three lecture series of distributed deep learning.