

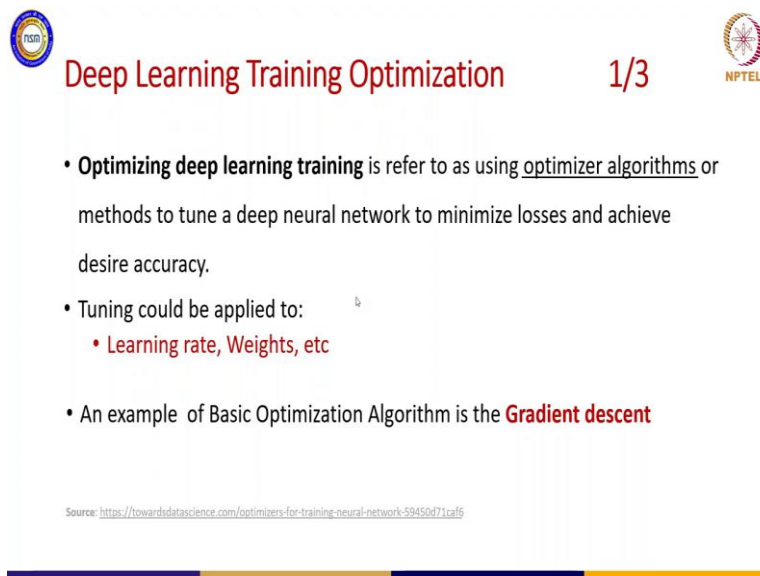
Applied Accelerated Artificial Intelligence
Dr. Tosin Adesuyi
Department of Computer Science and Engineering
Indian Institute of Technology, Madaras

End to End Accelerated Data Learning
Lecture - 30
Optimizing Deep Learning Training: Automatic Mixed Precision Part - 1

Welcome everyone to the session 2 of end to end Accelerated Deep Learning and so, this particular session would be taken by Dr. Tosin; Tosin is working as a GPU Advocate at NVIDIA and is based out in Korea and over to you Tosin for taking the session today.

Thank you Bharat. Hi everybody, I welcome you to this section which is based on end to end Accelerated Data learning and today, we will be talking about Optimizing Deep Learning Training and we will focus on Automatic Mixed Precision. So, that is what we will focus on today.

(Refer Slide Time: 01:08)



The slide features a title "Deep Learning Training Optimization" in red, with a slide number "1/3" to its right. On the left is a circular logo with "IITM" and on the right is the NPTEL logo. The main content consists of three bullet points: "Optimizing deep learning training is refer to as using optimizer algorithms or methods to tune a deep neural network to minimize losses and achieve desire accuracy.", "Tuning could be applied to:" followed by a sub-bullet "Learning rate, Weights, etc", and "An example of Basic Optimization Algorithm is the **Gradient descent**". At the bottom, a source URL is provided: <https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6>. The slide is decorated with a horizontal bar at the bottom composed of blue and yellow segments.

First of all, I will be talking about Deep Learning Training Optimization. So, what do we refer to as a deep learning training optimization? So, we are talking about optimization algorithm which is used during the training of a deep neural network, in order to achieve a desired goal and in a nutshell, you are worth reducing the loss to a minimal level. So, this is commonly used when we are training a deep neural network. So, these optimizers

that have been used. So, this optimizer, they help us to the deep neural network to be able to reach a convergence level faster.

And so, what is actually been tuned in the deep neural network? The learning rates and the weights that one have been tuned and also, the other parameters which also been tune like the the batch size also they have been tuned. So, an example of optimization algorithm is the gradient descent. So, the gradient decent is commonly used and we have types of gradient descent that are used.

(Refer Slide Time: 02:38)

The slide features the NSM logo on the left and the NPTEL logo on the right. The main title is "Deep Learning Training Optimization 2/3". Below the title, a bulleted list under "Gradient descent" includes: ✓ Stochastic Gradient Descent, ✓ Mini-Batch Gradient Descent, and ✓ Momentum(reduce high variance SGD). To the right of the list is a contour plot showing the optimization path of an optimizer starting from a point and moving towards a local minimum. Below the contour plot is a line graph showing the loss function over 100 iterations for five optimizers: SGD (red), ADAGRAD (blue), RMSPROP (green), ADADELTA (purple), and ADAM (yellow). The graph shows that ADAGRAD and ADADELTA converge much faster than the others, reaching a lower loss value by iteration 100.

Source: <https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6>

So, I have listed three here; you we have these stochastic gradient descent which have been used. This stochastic what it does is that what it picks the data in batches in terms of what randomly using probability. Also, there is the mini batch gradient descent as well. These big chunks of the batches, it feed them into the momentum.

And also, there is what we call the momentum. So, this momentum is used to reduce the high variance in the stochastic gradient distance. There are other examples of optimizer like the ADAMs, the ADAGRAD and ADADELTA and many of them have been used which I know many of us are familiar with this. So, in the event, where training model are becoming more complex, researchers are trying to solve complex tasks, it require larger models to be built and also, more data are also required to be used to build these models.

(Refer Slide Time: 03:49)



Deep Learning Training Optimization 3/3

- **Advanced optimization of deep learning training phase**
 - ✓ Memory consumption to accommodate large DNN
 - ✓ Memory bandwidth to accommodate data transfer operation
 - ✓ Tensor core computation for speedup
- **Examples include:**
 - ✓ Acceleration with GPU
 - ✓ **Mixed Precision**
 - ✓ XLA(Accelerated Linear Algebra)
 - ✓ Transfer Learning

So, in this lights, there is a need for what we call the advanced optimization for deep learning training phase. So, and when you want to go through this level, there are things which you may need to consider. So, you will be looking at the memory consumption to accommodate large DNN model and also, you will be looking at the memory bandwidth also which will be required to transfer data and you know large model might require large data and there is transfer here and there that will be needed during the training phase.

Also, you are looking at the word speed up in terms of computation and which would require some tensor core. So, now you make use of advanced optimization that can give you all these attributes. So, example is what you use GPU, you accelerate which GPU that is one level of advanced optimization and another one is the mixed precision which we are going to focus on today and there is also an aspect called the XLA which is accelerated linear algebra, this is from tensor flow a special compiler and also there is the transfer learning.

So, we will talk about transfer learning in our subsequent a course. So, we focus first of all on the mixed precision.

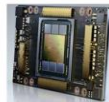
(Refer Slide Time: 05:21)



Mixed Precision



- **Mixed Precision** is the combine use of different numerical format(single and half precision computation) in the training of a deep neural network.
- ✓ single precision: **FP32 (float32)**
- ✓ half precision: **FP16 (float16)**
- Mixed Precision is possible on the following GPU architectures:



Ampere



Volta



Turing

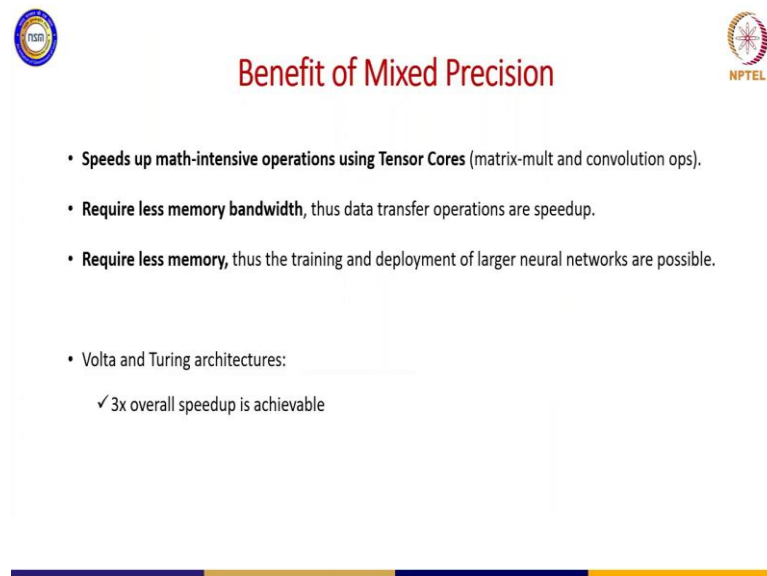
So, what do we refer to as mixed precision? A mixed precision is a using different numerical format with respect to deep learning, we can say mixed precision is the use of single and half precision for computation during the training of deep neural network. And what the word refer to as single precision? Single precision is simply the float 32 is what we refer to as single precision. Why half precision is float 16, which we call FP16?

So, we are looking at a scenario, where in the training of a deep neural network, these two precision data type are being used. So, it is an high level optimization process and how for you to use this mixed precision, there are requirement that is to look at the resources that the usage of this precision are possible. So, you must consider the GPU architecture which facilitate the use of these mixed precisions.

First of all, the one we have is the ampere. So, it is possible if your GPU is ampere architecture, you can use mixed precision is possible. For volta architecture, mixed precision is also possible and also, for tuning architecture, mixed precision is also possible.

can get more on that. But I would not want to put out more into these details because I want to focus on the empirical side of it, which is the practical side.

(Refer Slide Time: 09:26)



The slide features a title 'Benefit of Mixed Precision' in red text. It includes two logos: 'nsm' on the left and 'NPTEL' on the right. The main content is a bulleted list of benefits, with a sub-section for Volta and Turing architectures.

- **Speeds up math-intensive operations using Tensor Cores** (matrix-mult and convolution ops).
- **Require less memory bandwidth**, thus data transfer operations are speedup.
- **Require less memory**, thus the training and deployment of larger neural networks are possible.

• Volta and Turing architectures:

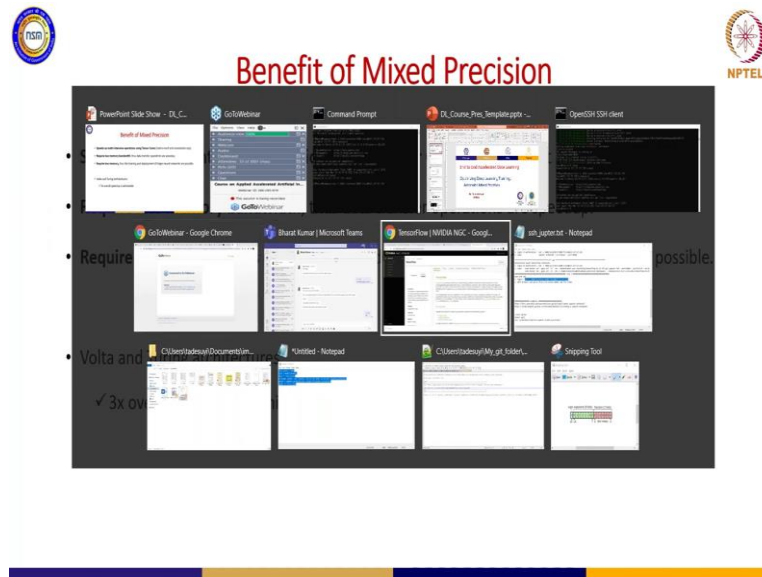
- ✓ 3x overall speedup is achievable

So, what are the benefit of mixed precision? The first thing is that with mixed precision, you are able to perform matrix multiplication and convolution. Convolution operations in your deep neural network with maximum speed up that is mass intensity operations are being performed using the tensor core. Like I said before that the tensor core is dedicated for such operation and also, it require less memory bandwidth.

So, when it requires less memory bandwidth, then you can transfer more data operation can be done. So, in a short time, then it require lesser memory. Why? Because it require lesser memory in the sense that operations are being performed in half precision mode. So, if those operation have been performed in half precision mode, either it require lesser memory and thus, you can train large models very well.

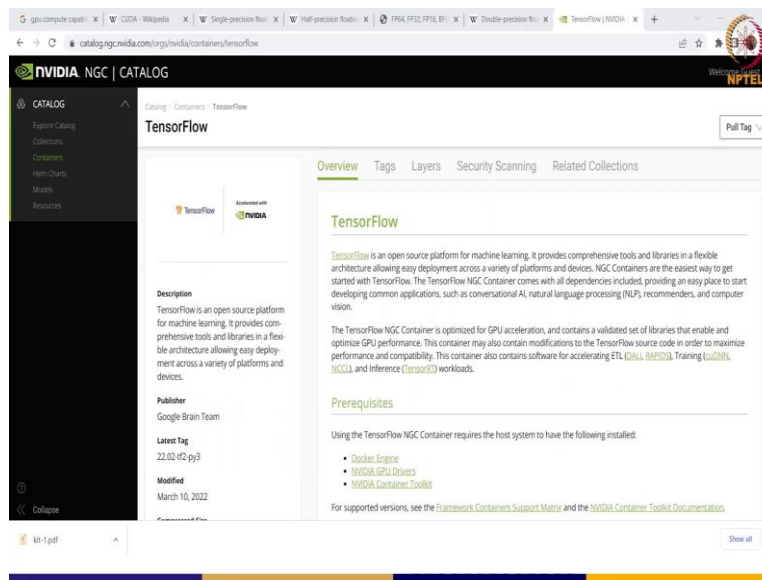
So, and after using mixed precision, so what are the benefits you can also get from there is that you get minimum of 3x speedup based on volta architecture and also, Turing architecture, if these are the GPU architecture you used. So, based on that, you can gets 3x speedup. If you use ampere, you will get definitely get more than that.

(Refer Slide Time: 11:05)



There are many if you want to know the classes of your architecture of your GPU. So, you can check online, there are several of them online which you can check on.

(Refer Slide Time: 11:15)



(Refer Slide Time: 11:18)

Architecture	GPU Models	Mobile SoC Models	Mobile Device Brands	
Maxwell	Geforce GTX 980M, Geforce GTX 970M, Geforce GTX 960M	Quadro M5000M, Quadro M4000M, Quadro M3000M	Tegra X1, Jetson Nano, DRIVE CX, DRIVE PX	
Pascal	GP100	Quadro GP100	Tesla P100	
	GP102, GP104, GP106, GP107, GP108	Nvidia TITAN Xp, Titan X, Geforce GTX 1080 Ti, GTX 1080, GTX 1070 Ti, GTX 1070, GTX 1060, GTX 1050 Ti, GTX 1050, GT 1030, GT 1010, MX300, MX330, MX250, MX230, MX150, MX130, MX110	Quadro P5000, Quadro P5000, Quadro P4000, Quadro P2000, Quadro P2000, Quadro P1000, Quadro P400, Quadro P500, Quadro P520, Quadro P600, Quadro P5000(Mobile), Quadro P4000(Mobile), Quadro P2000(Mobile)	Tesla F40, Tesla P6, Tesla P4
Volta	GV100	NVIDIA TITAN V	Quadro GV100	Tesla V100, Tesla V100S
	GV102, GV104, GV106, GV107, GV108, GV109	NVIDIA TITAN RTX, Geforce RTX 2080 Ti, RTX 2080 Super, RTX 2080, RTX 2070 Super, RTX 2070, RTX 2060 Super, RTX 2060 12GB, RTX 2060, Geforce GTX 1660 Ti, GTX 1660 Super, GTX 1660, GTX 1650 Super, GTX 1650, MX550, MX450	Quadro RTX 8000, Quadro RTX 6000, Quadro RTX 5000, Quadro RTX 4000, T1000, T800, T400, T1200(Mobile), T800(Mobile), T500(Mobile), Quadro T2000(Mobile), Quadro T1000(Mobile)	Tesla T4
Ampere	GA100		RTX A6000, RTX A5000, RTX A4000, RTX A4000, RTX A3000, RTX A2000(Mobile), RTX A4000(Mobile), RTX A3000(Mobile), RTX A2000(Mobile)	A500, B000, A100, A00, A30
	GA102, GA104, GA106, GA107	Geforce RTX 3090 Ti, RTX 3090, RTX 3080 Ti, RTX 3080 12GB, RTX 3080, RTX 3070 Ti, RTX 3070, RTX 3060 Ti, RTX 3060, RTX 3050 Ti(Mobile), RTX 3050(Mobile), RTX 2050(Mobile), MX570		A40, A16, A10, A2

(Refer Slide Time: 11:19)

Architecture	GPU Models	Mobile SoC Models	Mobile Device Brands	
Maxwell	GM107, GM108	Geforce GTX 750 Ti, Geforce GTX 750, Geforce GTX 960M, Geforce GTX 950M, Geforce 940M, Geforce 930M, Geforce GTX 940M, Geforce 930M	Quadro K1200, Quadro K2200, Quadro K620, Quadro M2000M, Quadro M1000M, Quadro M800M, Quadro K520M, M450 S10	Tesla K80, NXPTEL
	GM200, GM204, GM206	Geforce GTX Titan X, Geforce GTX 980 Ti, Geforce GTX 980, Geforce GTX 970, Geforce GTX 960, Geforce GTX 950, Geforce GTX 750 SE, Geforce GTX 880M, Geforce GTX 970M, Geforce GTX 960M	Quadro M8000, Quadro 3400, Quadro M8000, Quadro M5000, Quadro M4000, Quadro M3000, Quadro M5500, Quadro M5000M, Quadro M4000M, Quadro M3000M	Tesla M4, Tesla M40, Tesla M6, Tesla M60
Pascal	GP100		Quadro GP100	Tesla P100
	GP102, GP104, GP106, GP107, GP108	Nvidia TITAN Xp, Titan X, Geforce GTX 1080 Ti, GTX 1080, GTX 1070 Ti, GTX 1070, GTX 1060, GTX 1050 Ti, GTX 1050, GT 1030, GT 1010, MX300, MX330, MX250, MX230, MX150, MX130, MX110	Quadro P5000, Quadro P5000, Quadro P4000, Quadro P2000, Quadro P2000, Quadro P1000, Quadro P400, Quadro P500, Quadro P520, Quadro P600, Quadro P5000(Mobile), Quadro P4000(Mobile), Quadro P2000(Mobile)	Tesla F40, Tesla P6, Tesla P4
Volta	GV100	NVIDIA TITAN V	Quadro GV100	Tesla V100, Tesla V100S
	GV102, GV104, GV106, GV107, GV108, GV109	NVIDIA TITAN RTX, Geforce RTX 2080 Ti, RTX 2080 Super, RTX 2080, RTX 2070 Super, RTX 2070, RTX 2060 Super, RTX 2060 12GB, RTX 2060, Geforce GTX 1660 Ti, GTX 1660 Super, GTX 1660, GTX 1650 Super, GTX 1650, MX550, MX450	Quadro RTX 8000, Quadro RTX 6000, Quadro RTX 5000, Quadro RTX 4000, T1000, T800, T400, T1200(Mobile), T800(Mobile), T500(Mobile), Quadro T2000(Mobile), Quadro T1000(Mobile)	Tesla T4
Turing	TU102, TU104, TU106, TU108, TU109, TU117	NVIDIA TITAN RTX, Geforce RTX 2080 Ti, RTX 2080 Super, RTX 2080, RTX 2070 Super, RTX 2070, RTX 2060 Super, RTX 2060 12GB, RTX 2060, Geforce GTX 1660 Ti, GTX 1660 Super, GTX 1660, GTX 1650 Super, GTX 1650, MX550, MX450	Quadro RTX 8000, Quadro RTX 6000, Quadro RTX 5000, Quadro RTX 4000, T1000, T800, T400, T1200(Mobile), T800(Mobile), T500(Mobile), Quadro T2000(Mobile), Quadro T1000(Mobile)	Tesla T4

(Refer Slide Time: 11:48)



Mixed Precision Training



- Training with mixed precision speedup computations by performing ops in half precision format.
- Minimal information are stored in single-precision to retain as much information in critical parts of the network.
- **Training steps**
 - Porting the model to use the FP16 data type where appropriate
 - Adding loss scaling to preserve small gradient values.

Then, mixed precision training. So, after, while you do train your model using mixed precision, so ideally the training with mixed precision will help you to speed up computation. Because those operations they are performed in half precision format. That is its performed those operation using float 16, FP16 and then, the minimal information are being stored in FP32 which is the single precision. So, the process is that you are using both FP16 and you are using FP32 as well.

So, during the training, what are the steps you need to take during the training is that you need to port your model to use the FP16 data type, where it is appropriate. The only place you will not use the FP16 is if your model may be your dealing with multi-classification tasks, where you might want to use Softmax activation function. So, at the Softmax activation function, you have to use FP32 there. Then, the other part you will be you will be able to use FP16.

And the second part, step is that you need to add a loss scaling to preserve small gradient. So, why this is that? When you use mixed precision because of the computation that happen with FP16, it usually leads to small gradients which is we call it on the float such that during the gradients computation, the gradient is so small that sometimes it leads to 0 value and those 0 values affect the accuracy of the model. So, you will not be able to get the accuracy, you will get if you are using only FP32. So, then the loss of

your model has to be scaled and how do you scale? You scale by multiplying loss with a particular value.

(Refer Slide Time: 14:04)



FP16 Porting in Model



- **Tensor core enabled framework**
 - Choose **FP16** format for tensor, convolution, fully-connected layers,
 - Keep all hyper-parameters of **FP32** training session.
- **Issue with the above method**
 - ✓ Some network gradient becomes very tiny or lead to zero 0 (dynamic range too narrow, hence underflow)
 - ✓ Required gradient value to be shifted into **FP16** representable range to match accuracy of **FP32** training session.
- **Solution**
 - ✓ Scaling is required

So, during FP16 porting in your model, so how do you how will you be able to achieve this is that? You use frameworks that tensor core are enabled because the that is being performed actually by tensor core. So, because those tensor core they also have precisions which is based on FP16 and FP32.

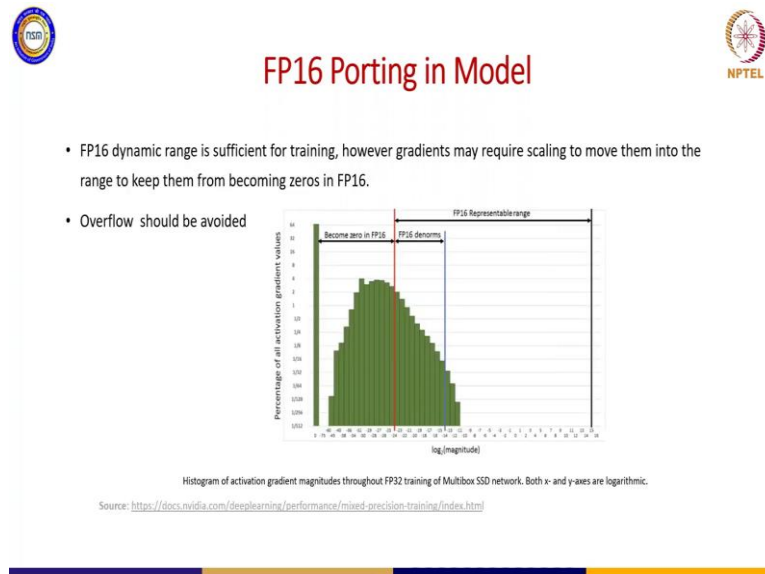
So, you choose FP16 format for a tensor which helps you to do the matrix multiplication are there, the convolution operation also it works with the fully connected layer and also, you keep all the hyper parameters of FP32 that is the floating point 32 training section there.

Now, like I said earlier, there are issues with this; but the issue that will not arise there is that FP16 has a way of what we call dynamic range being too narrow and it leads to what we call the underflow. So, gradient become very small that leads to you may have 0 gradient and this will affects your training. So, it requires a gradient, it requires such that you need to shift do some kind of shifting into FP16 representable range to match the accuracy of FP32 that you have.

So, now, how do you achieve that is the solution is that you use what we call scaling and scaling which we I have mentioned before is that you multiply by a particular threshold,

you multiply your loss by a particular threshold. And what you get there is what we call scaling, but I will still explain further as we continue.

(Refer Slide Time: 15:58)



So, FP16 as a dynamic range which is sufficient for you to train and it does this in a way that it affects the gradients of your deep neural network and so, what you need to do is to what to do scaling like I said; but while doing the scaling there is an issue that will arise as well.

Because when you scale you multiply by a particular threshold value, it can result to what we call overflow and what we mean by overflow? Overflow in the sense that you will be having values that may lead to infinity or NAN value that will lead there. So, that is what happened in that sense.

(Refer Slide Time: 16:44)



Loss Scaling



- The purpose of loss scaling is to preserve small gradient magnitudes.
- **Process**
 - A single multiplication by **scaling** the loss values computed in the **forward pass** prior to starting **backpropagation**
 - Weight gradients **unscaled** before weight update, to maintain the magnitude of updates the same as in **FP32** training
 - **FP32** copy of weights requires update and large reductions should be left in **FP32**

Training procedure

<ol style="list-style-type: none">1. Maintain a primary copy of weights in FP322. For each iteration:<ol style="list-style-type: none">i. Make an FP16 copy of the weightsii. Forward propagation (FP16 weights and activations)iii. Multiply the resulting loss with the scaling factor 5iv. Backward propagation (FP16 weights, activations, and their gradients)v. Multiply the weight gradient with 1/5vi. Complete the weight update (including gradient clipping)	<pre>loss_scale = 1024 loss = model(inputs) loss *= loss_scale # Assume 'grads' are float32. You do not want to divide float16 gradients. grads = compute_gradient(loss, model.trainable_variables) grads /= loss_scale</pre>
---	--

So, and the next phase is Loss scaling. So, how do you perform loss scaling and what is the reason for loss scaling? The reason for loss scaling is that you want to preserve this small gradient. How do you preserve it such that you are able to train your model as if you are using FP32 only.

So, the first thing you do is to what? You make a single multiplication by scaling the loss value. So, when you scale the loss value at the forward pass. So, what do you mean by forward pass? This is like the forward propagation that is you when your model forced to run your deep neural network, when it runs and it eats the output; before it comes back again which is what feeding the output, it get feeding it back to deduct from the weight.

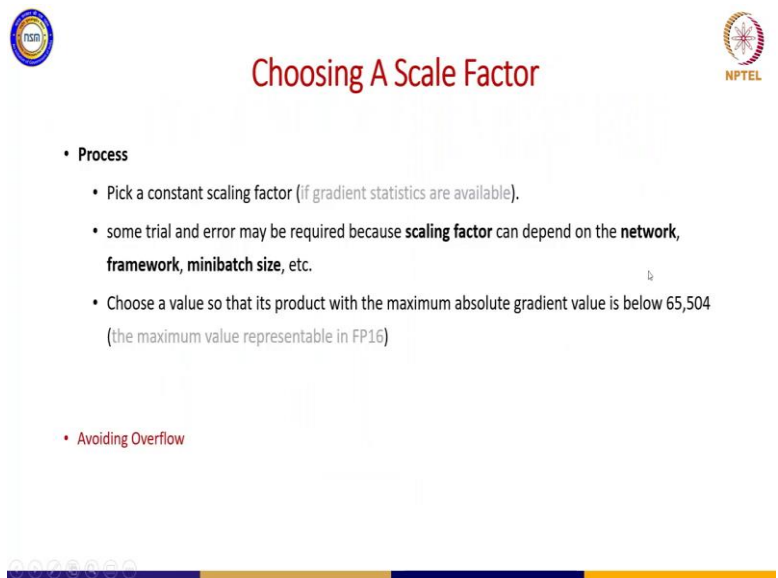
So, which is the word the process of back propagation. So, the first thing what I am trying to say is that you multiply your loss scale For example, if I assigned 1024 my loss as my loss scale, I can compute my model here get the loss and then, most of use the loss scale which is 1024 to multiply and add to the loss. So, this is what we call scaling.

So, this is done at the forward pass of your model, when your model initially run the first time. So, then you compute the gradient. Secondly, that is what you do; you compute the gradients. So, after you compute the gradient, then you on scale back again, you want scale back. So, or scale back. So, add you scale back, you divide the gradient which you computes by the value you use to scale initially. So, when you do that, then you can now updates the weights.

So, there is a procedure for doing that this is a simple algorithm to do that I would can see here. So, the first one, the first line is what you maintain a primary copy of your weight. Your weight will be in FP32, then during the iteration. So, this iteration this is what happens while you are training your deep neural network, where you can have the step and the iterations that are there. So, within that step and iteration, you can make an FP16 copy of your weights and after then, so that F16 copy of your weight, you perform that with what the forward propagation that is the initial runs.

Then, so, you scale. So, how do you scale which I have explained here, you scale you multiply the resulting loss with a scaling factor x . So, the scaling factor x is what you have here. Then you cannot do your back propagation. So, after the back propagation is performed, then you on scale back that is you divide the gradient you get there by the value of the scale and then, you cannot proceed to updates the weights.

(Refer Slide Time: 19:46)



Choosing A Scale Factor


- **Process**
 - Pick a constant scaling factor (if gradient statistics are available).
 - some trial and error may be required because **scaling factor** can depend on the **network, framework, minibatch size**, etc.
 - Choose a value so that its product with the maximum absolute gradient value is below 65,504 (the maximum value representable in FP16)
- **Avoiding Overflow**

So, here how do we know; how do we pick a scaling factor? Like ok, do we just pick 1024 or we just pick one just pick to how do you know how to pick that? So, there is a process for that is that the first one is you can pick a constant value scaling value and that is can only be done if you have the statistical gradient value for your gradient. If you have the statistics for your gradient that is when that is possible or you can use some brute force method which you refer to as trial by error to test.


If this particular scaling value that you pick, if it will not run into overflow, so you can use trial by error. Now, why do you have to use trial by error? It is because this scaling factor is dependent on some other attributes or factors like the your network size that is your model size, the framework you are using, whether you are using tensor flow or you are using pytorch or you are using mx, also the mini batch also the size of your minibatch, it also depend on that.

So, that is why you have to try to you able to get the right scaling factor for the kind of tasks you are doing and also, you can choose a value which will not exceed the maximum value which is represented in FP16 and that maximum value is the 65,400. These value have been proved to be efficient and work well and in research that it is if your value is below this range your scaling value. Then, your model would definitely do well; but while doing this, you must avoid what we call the overflow.

(Refer Slide Time: 21:44)



Training Procedure in Choosing a Scale Factor



1. Maintain a primary copy of weights in **FP32**.
2. Initialize **S** to a large value.
3. For each iteration:
 - i. Make an **FP16** copy of the weights.
 - ii. Forward propagation (**FP16** weights and activations).
 - iii. Multiply the resulting loss with the scaling factor **S**.
 - iv. Backward propagation (**FP16** weights, activations, and their gradients).
 - v. If there is an **Inf** or **NaN** in weight gradients:
 - o Reduce **S**.
 - o Skip the weight update and move to the next iteration.
 - vi. Multiply the weight gradient with $1/S$.
 - vii. Complete the weight update (including gradient clipping, etc.).
 - viii. If there hasn't been an **Inf** or **NaN** in the last **N** iterations, increase **S**.

The graph shows training loss on the y-axis (ranging from 2.5 to 5.0) and training iteration on the x-axis (ranging from 0K to 2,000K). Three curves are plotted: FP32 (black), Mixed Precision (loss scale 1) (red), and Mixed Precision (loss scale 128) (green). The FP32 curve shows a sharp spike in loss around 1,000K iterations, indicating overflow. The Mixed Precision curves show a much smoother and lower loss, demonstrating the benefits of mixed-precision training.

Training curves for the bigLSTM English language model shows the benefits of the mixed-precision training techniques. The Y-axis is training loss

Source: <https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html>

So, how do we now train because why you are trying to pick up scaling value, you need to test it during your training see that ok, this value scaling well and it is not leading to overflow or you are not experiencing an underflow. So, the first thing you do this is an algorithm here is that you maintain the primary copy of your weight which is in FP32.

So, how do you have this is that your model variable data type by default will be FP32? Then, you initialize your scaling value to a large value, you pick just pick a large value which is less than the value I mentioned here which the result will be less than this. So,

you pick that. Then, in your iteration, so you make a copy of your FP16 weights, then you perform forward propagation with it, then you scale using with the value.

Then, after you scale you perform your model perform the backward propagation. So, after the backward propagation is performed, so what you need to do is what you have to check, you check if that does not resort to overflow that is infinity or NAND in your weights your gradient weights.

So, if that on call, what you need to reduce what? You need to reduce the value of that scaling factor x , you reduce it from the initial one then. You would skip the weight update; you would not update because you can update with an infinity or a none value in your weight. So, you skip that.

So, when you skip that, after you have reduced then you start all over again. But if that is not the case then what you need to do is you on scale back your gradient by dividing it with the scale factor value that you pick and then, you complete your weight update. So, when you complete your weight update then, so after that has been successful, you check your model; I mean the iteration fully run.

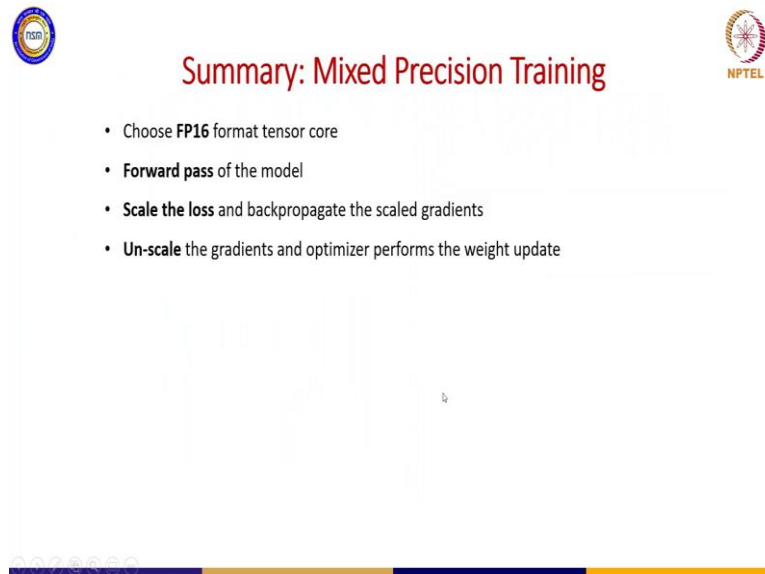
So, and if you if why the iteration fully run, you check that oh there is no infinity or none value existing in between the iteration. Then, you can also increase your scaling value since your iteration the iteration have performed, there is no overflow. So, you can increase it. So, when there is an over overflow, you reduce its and to able to get a value where overflow does not occur anymore and that after that value, a slight increase in that value may lead to overflow.

So, at exact point, you can use that particular scaling factor and this is an example here which is perform on big using big LSTM. So, what you can see here is that the FP32, the loss this the y axis is the loss; the loss this is the black one you see the loss decrease, it decreased steadily; then the mixed precision used when the loss scale is at 1, you can see its decreased and suddenly rises again.

So, which does not give good loss function, it does not give good loss. So, because the loss is increasing again. then our accuracy will be down. So, when the mixed precision loss is scaled at 128, we can see which is the green one its give the same result as if we were using FP32. So, it is it was able to achieve the same result as FP32. However, the

were able to use it to train larger models, it require less memory would gain speedup as well.

(Refer Slide Time: 25:41)




The slide features a title "Summary: Mixed Precision Training" in red text. It includes a list of four steps: "Choose FP16 format tensor core", "Forward pass of the model", "Scale the loss and backpropagate the scaled gradients", and "Un-scale the gradients and optimizer performs the weight update". The slide is framed by a blue and yellow border and contains logos for NSM and NPTEL.

Summary: Mixed Precision Training

- Choose **FP16** format tensor core
- **Forward pass** of the model
- **Scale the loss** and backpropagate the scaled gradients
- **Un-scale** the gradients and optimizer performs the weight update

So, in summary, the mixed precision training is that you have to pick choose a floating point of FP16 format, then you perform your forward pass in your model. After that you scale the loss and then, do your back propagation of the scale gradient and then, you will scale back the gradients and the optimizer performs the weight update. These are just the summary of all what have been seen.

(Refer Slide Time: 26:11)



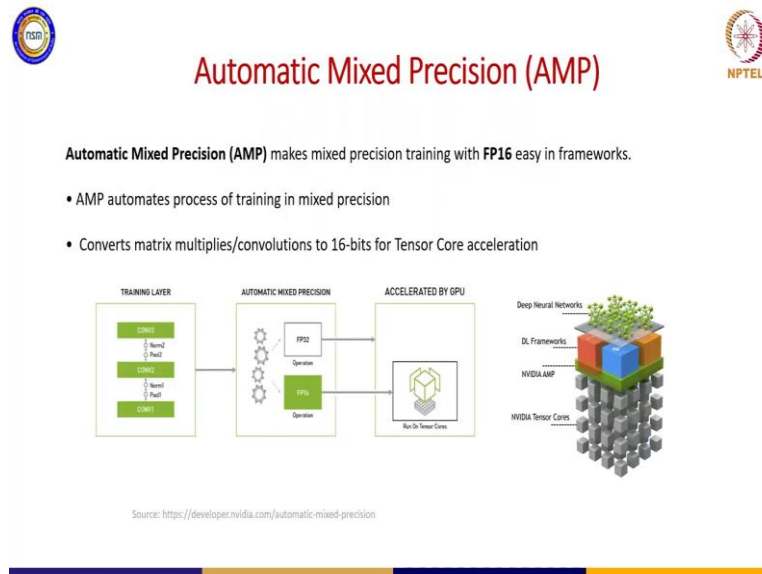
The slide features the text "Is there a simpler way?" in red. Below the text is a cartoon illustration of a yellow thinking face with a question mark above it. The slide is framed by a blue and yellow border and contains logos for NSM and NPTEL.

Is there a simpler way?



So, is there a simpler way with all this more of theoretical scale, unscale and do that. I would say yes, there is a simpler way.

(Refer Slide Time: 26:23)

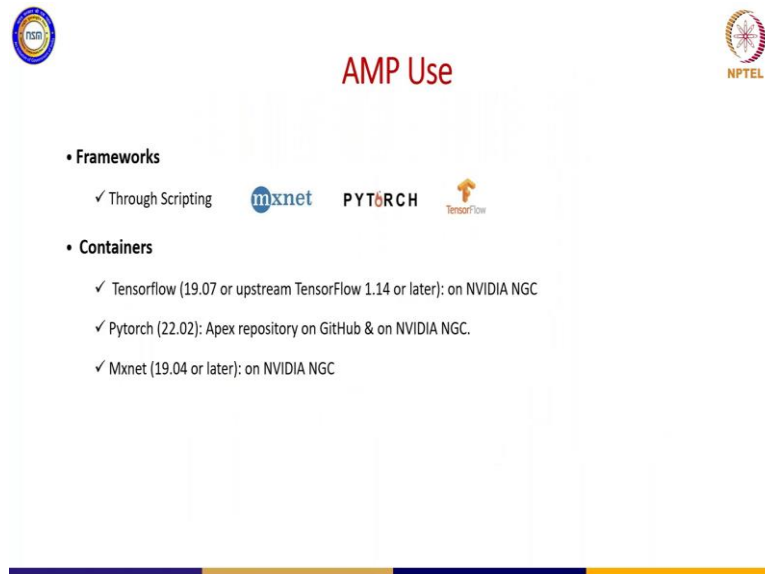


The slide features the NPTEL logo on the left and right. The title "Automatic Mixed Precision (AMP)" is centered in red. Below the title, a text box states: "Automatic Mixed Precision (AMP) makes mixed precision training with FP16 easy in frameworks." This is followed by two bullet points: "AMP automates process of training in mixed precision" and "Converts matrix multiplies/convolutions to 16-bits for Tensor Core acceleration". A diagram illustrates the workflow: "TRAINING LAYER" (containing Conv2, Conv2T, Conv3, Conv3T) feeds into "AUTOMATIC MIXED PRECISION" (showing FP32 and FP16 operations), which then feeds into "ACCELERATED BY GPU" (showing operations on NVIDIA Tensor Cores). To the right, a 3D stack of blocks represents the hardware stack: "NVIDIA Tensor Cores" at the base, followed by "NVIDIA AMP", "DL Frameworks", and "Deep Neural Networks" at the top. A source URL is provided at the bottom: "Source: https://developer.nvidia.com/automatic-mixed-precision".

So, the simplest way this available method is what we call the Automatic Mixed Precision; AMP. Now, the automatic mixed precision make things easier. So, you can perform your training using FP16 within the framework. So, you do not have to do all the manual job again. So, the automatic mixed precision automate the process of training in mixed precision. It does the automation for you. It converts the matrix then and the convolution into 16 bits for your tensor.

So, for example, what we have let us say this is our training layer, the convolution layer that you can see here. So, the training is being done using what automatic mixed precision which is both FP32 and FP16 as well and the computation that will go within the FP16, I will be run on the tensor core. And for example, this is an example here that you have your deep neural network here, the DL, the frameworks are here and the another framework is embedded with enabled with automatic mixed precision here. So, which will be handled by the tensor core.

(Refer Slide Time: 27:37)

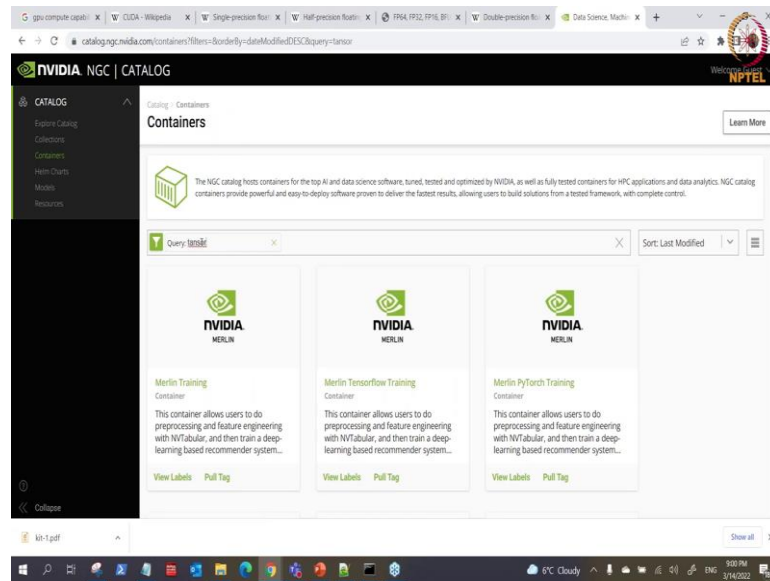


The slide features a title 'AMP Use' in red text at the top center. On the left is the NSM logo and on the right is the NPTEL logo. Below the title, there are two main sections: 'Frameworks' and 'Containers'. The 'Frameworks' section includes a checkmark for 'Through Scripting' followed by logos for mxnet, PYTORCH, and TensorFlow. The 'Containers' section lists three items, each with a checkmark: 'Tensorflow (19.07 or upstream TensorFlow 1.14 or later): on NVIDIA NGC', 'Pytorch (22.02): Apex repository on GitHub & on NVIDIA NGC.', and 'Mxnet (19.04 or later): on NVIDIA NGC'. At the bottom of the slide, there is a decorative horizontal bar with segments in dark blue, gold, and yellow.

So, automatic mixed precision use how do we use the automatic mixed precision, we can use it using our frameworks through scripting. So, it exists here in mxnet, pytorch also, you can use tensor flow as well. And then, you can use container. However, in the container, I must say for tensorflow you can use it is only possible with tensorflow version 1 that is only where you can use the automatic mixed precision.

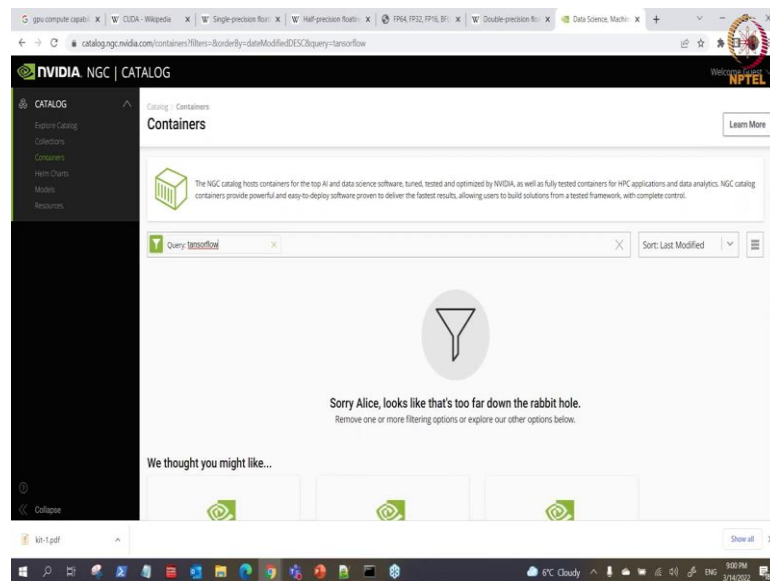
If you want to use a mixed precision, automatic mixed precision with tensorflow version 2, it might not be possible. I tried it there are so many errors that you can you might visit from there. Also, you can use pytorch, also mxnet as well. So, all of these you can pull them on the NVIDIA NGC. So, if you go to NVIDIA NGC here. So, from here so this is tensor flow. Let me come to catalog here ok; yeah.

(Refer Slide Time: 28:50)



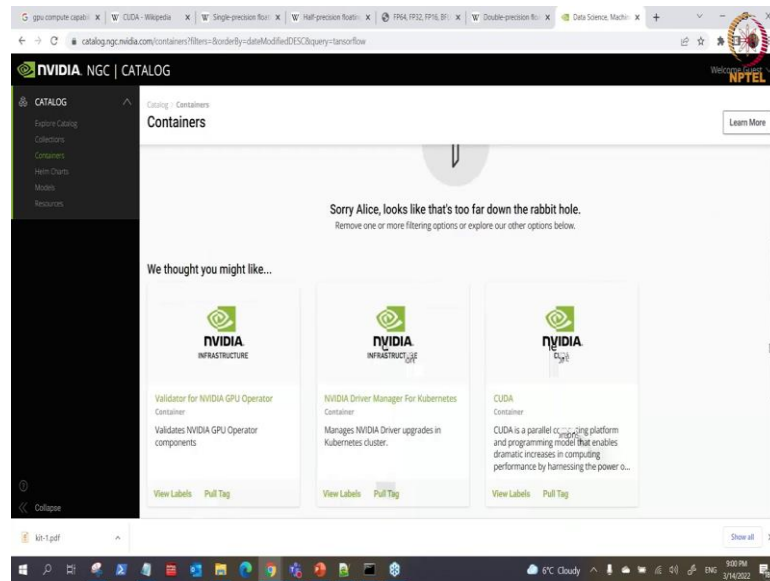
So, you can search, you can search for tensorflow or you search for pytorch.

(Refer Slide Time: 28:58)

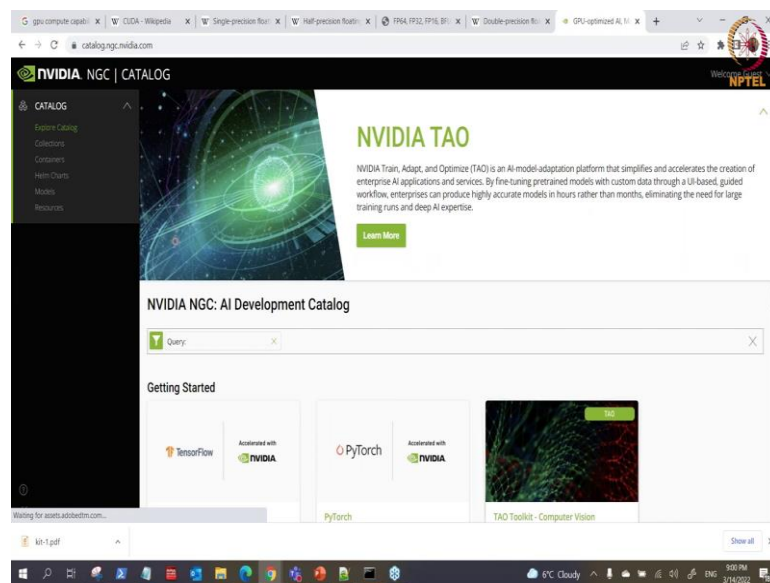


So, if you search all of that, come in ok alright.

(Refer Slide Time: 29:02)

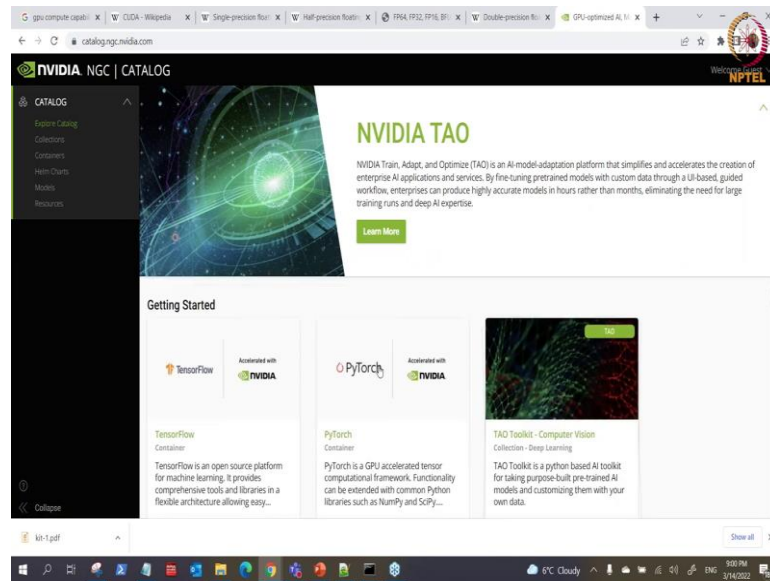


(Refer Slide Time: 29:08)

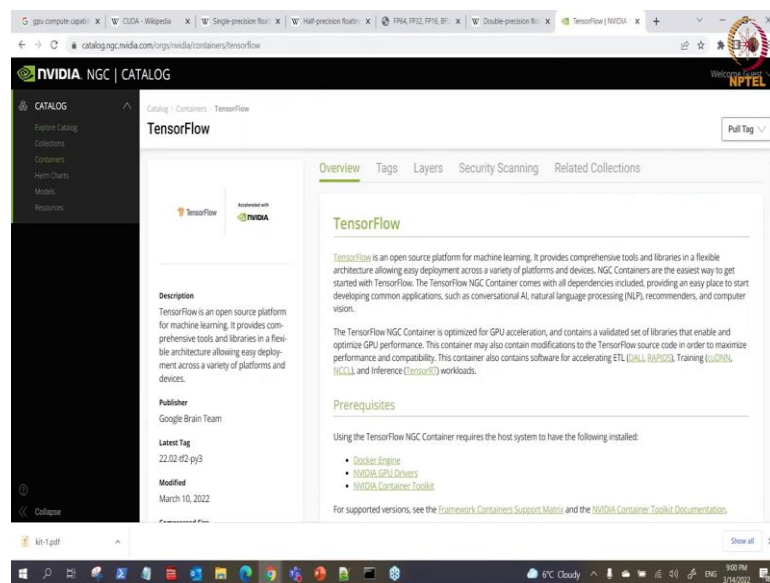


So, yeah they yeah you can just click you search for them, you can see this is tensorflow this is pytorch.

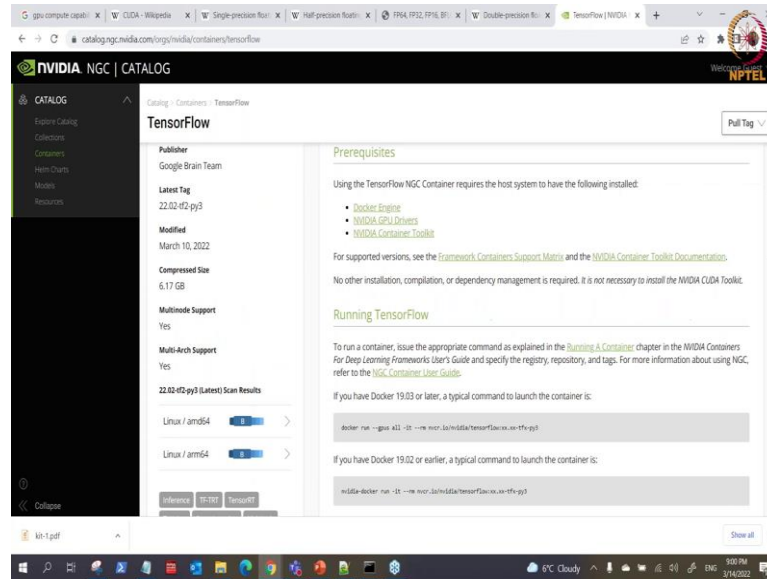
(Refer Slide Time: 29:13)



(Refer Slide Time: 29:19)

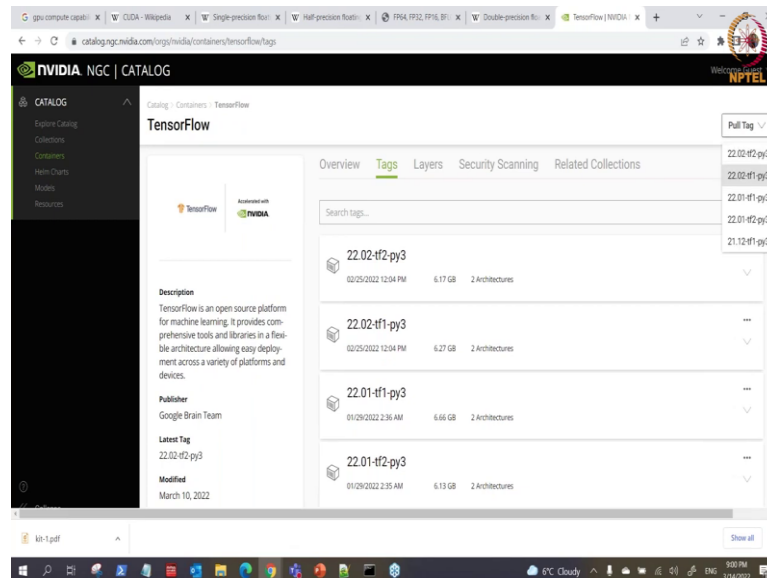


(Refer Slide Time: 29:21)

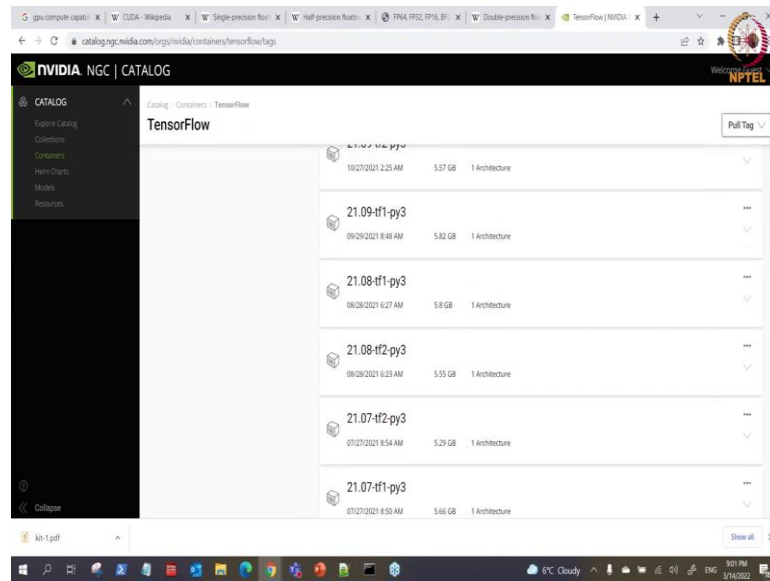


If you click on them, you can get here; you will pick. So, you can pull there are several versions here. Again, you can pull from there, but if you need to use if you are using this these are tensorflow version 2. So, here you can only use mixed precision; you cannot use automatic mixed precision.

(Refer Slide Time: 29:42)

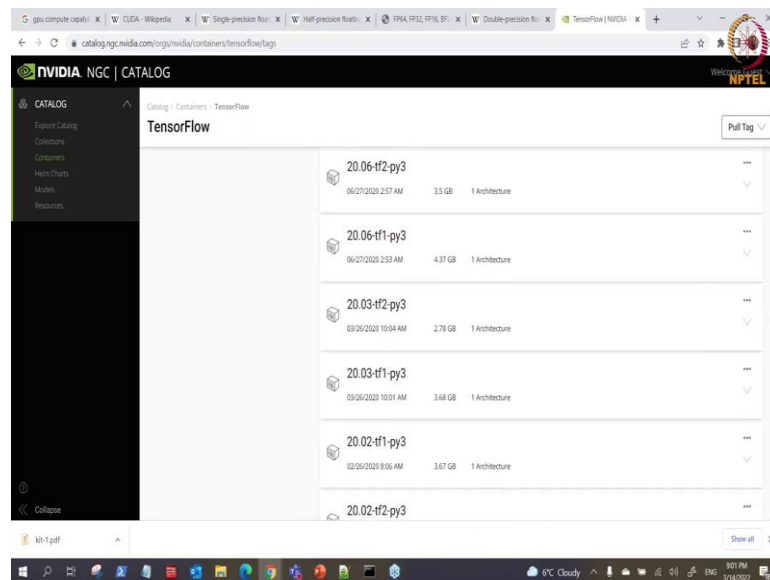


(Refer Slide Time: 29:51)

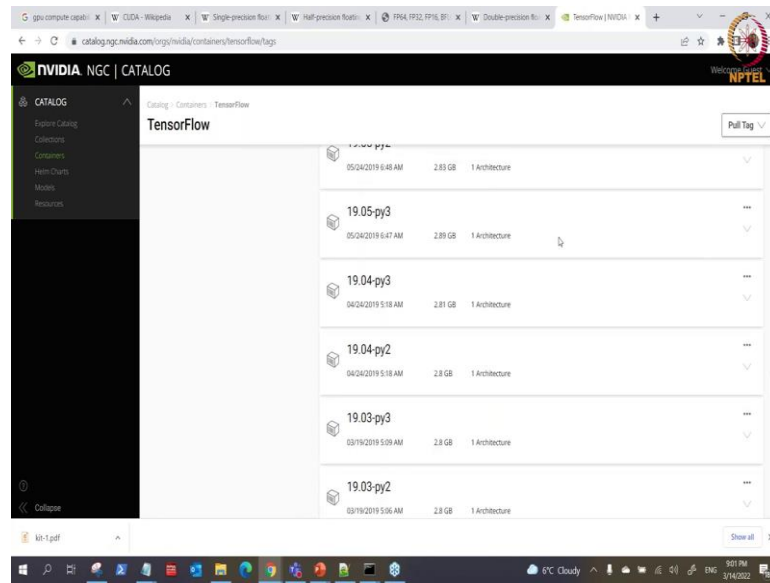


For you to use automatic mixed precision, so you click on the tag here, you come here there is a version there is version 19 here.

(Refer Slide Time: 29:55)



(Refer Slide Time: 29:57)



Version go to 19 07. So, yes, if you pull this one, this is version you will pull this. So, you just need to click here, then you pull and you can you know use that for your automatic mixed precision.