

**Applied Accelerated Artificial Intelligence**  
**Dr. Satyajit Das**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Palakkad**

**Lecture - 26**  
**Accelerated TensorFlow Part - 1**

Good evening everybody. So, today we will look into the Accelerated Tensor. So, first we will look into the data pipeline. So, we have mentioned that `tf dot data set` or `tf dot data` that particular class will give you the flexibility to maintain your data flow pipeline and different strategies we will see.

(Refer Slide Time: 00:42)

The slide is titled "Topics" and lists three bullet points: "3 ways to create a model", "Accelerated data pipelining", and "Distributed training". The TensorFlow logo is prominently displayed in the center. The slide is framed by a purple border. In the top left corner is the IIT Palakkad logo, and in the top right corner is the NPTEL logo. A small video inset of the speaker is visible in the bottom right corner.

Also in the so before going into that of course, we will see how to create the models in the 3 different ways.

So, the three different ways I mentioned in the last class that you will have the sequential way which is the more the best way to I mean start for the beginners. It is very flexible and if you want to have more control over definition of the model then you go for the functional way or you can go for the subclassing means subclassing APIs.

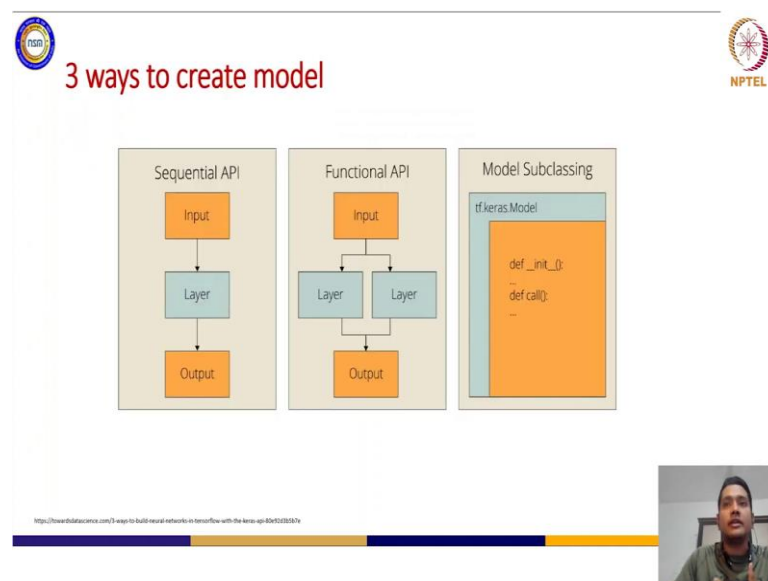
So, these 3 ways we will talk about then we will talk about the data pipelining accelerated data pipelining mostly. So, how you can actually define the pipeline so that your data bottlenecks all the data bottlenecks from different point of view maybe a

processing maybe opening the files, reading the files, different pre processing you might be applying. Also you might be reading data from let us say remote sources right. Let us say from s3 you are reading the data or maybe from hydro cluster you are reading the data.

So, all these ETL pipeline the extraction the transformation and the load. So, all these three pipelines how we can maintain very effectively to accelerate the models that we will be working and also we will talk about next the distributed training which is very interesting because here you will have many flexibilities that also we will see.

We will defining the different distributed strategies very very easy, but you need to understand what is the particular environment you are working in and that way you will be actually able to efficiently use those APIs or higher level abstractions to define what will be the distributed strategy for your particular model train.

(Refer Slide Time: 02:47)




So, let us go ahead to see these three APIs that as I was mentioning that you have the sequential API, which will have particular one input. So, maybe this is the input tensor or multi dimensional it may be and you will have one output, which is again one tensor and in the functional API you will have these inputs and outputs, but you will you have let us say you have multiple outputs you can see from these 2 layers you have multiple outputs and then you are concatenating to get one output.

So, this is just one use case. You can have any where in between. We can have multiple inputs multiple outputs multiple inputs single output and single output multiple single input multiple output and so on. And in sub classing you will have to use `keras.model`. So, this is your super class and then you will define your layers and you can connect them as you have done in PyTorch.


So, mostly we will focus on the functional API because this is bit tricky to work with, but once you just understand it, it is very easy to use it in cases like where you will be doing some transfer learning maybe or maybe multi class or multi stage classification ok.

So, or maybe working with multiple heads of classification or prediction. So, different use cases you can use this function and this is very very useful and most of the cases people use tensor flow just for functional ok.

(Refer Slide Time: 04:30)



## Quick overview



```
class WideAndDeepModel(keras.Model):
    def __init__(self, units=30, activation='relu', **kwargs):
        super().__init__(**kwargs) # handles standard arguments ( name, etc.)
        self.hidden1 = keras.layers.Dense(units, activation=activation)
        self.hidden2 = keras.layers.Dense(units, activation=activation)
        self.main_output = keras.layers.Dense(1)
        self.aux_output = keras.layers.Dense(1)

    def call(self, inputs):
        input_A, input_B = inputs
        hidden1 = self.hidden1(input_B)
        hidden2 = self.hidden2(hidden1)
        concat = keras.layers.Concatenate([input_A, hidden2])
        main_output = self.main_output(concat)
        aux_output = self.aux_output(hidden2)
        return main_output, aux_output

model = WideAndDeepModel()
```


```
model = Sequential()
model.add(Dense(4, activation='relu')) #<--- You don't have to specify
# input size. Just define the hidden layers
model.add(Dense(4, activation='relu'))
model.add(Dense(1)) # defining the optimiser and loss function

model.compile(optimizer='adam', loss='mse') # training the model
model.fit(x_train, y_train, validation_data=(x_test, y_test),
        batch_size=128, epochs=400)

## Creating the layers
input_layer = Input(shape=(3,))
layer_1 = Dense(4, activation='relu')(input_layer)
layer_2 = Dense(4, activation='relu')(layer_1)
output_layer = Dense(1, activation='linear')(layer_2)

## Defining the model by specifying the input and output layers
model = Model([input_layer, output_layer]) # defining the
# optimiser and loss function model.compile(optimizer='adam', loss='mse')

## training the model model.fit(x_train, y_train, epochs=400,
# batch_size=128, validation_data=(x_test, y_test))
```



So, quick overview of the sequential definition of the model that you have seen in the last class that we have defined one sequential model from the keras dot model and model dot add then you have different layers different defined and like dense layers. You have flattening you have convolution 2Ds.

So, all these things are there. Here we are using a 1 dense layer, 2 dense layer and 3 dense layers and we are actually this is the output dense. So, this is the output 1 and this is the output of this layer is 4. This output layer is 4, but remember here we are not

defining any input layer. So, only the hidden layers that we are defining and then we are getting the output. Because you have only one output and the one input.

So, you do not need to explicitly define the inputs and just in the compile you just can mention the, which loss you will be using optimizer you will be using and just hit the feed and it will do the training with these number of epochs and with these batches. So, this is the naive way of defining one model in sequential manner, but of course, this will suffice for your let us say most of the training models that you will be working with it. But we want to go forward with some complete advanced technique with this a functional API.

Now, let us see the layers first ok. So, layer 1 we are defining now. So, here all these layers that we have defined here and. So, what we got like this dense layer this dense layer and this dense layer.

So, all these whatever we are adding they are making one stack or let us say completely one off and then so this is the complete data structure itself ok. So, you do not need to worry about what you are getting in between right. So, this is the complete set and when you are defining the models or model layers in the functional API. So, in the functional API you need to have the layers ok.

So, dense layer convolution 2D layer, so any layer that you might have now here you see we are explicitly defining the input. So, input layer we are defining let us say one input will have this shape and with some data I do not know. So, this is just the representation. We will see one such use case and in the end of this session, but let us define the layers first. So, layer 1 layer 2 and output layer. So, these 3 dense layers we have defined.

So, now you see that this output so these are the just the layer definition. Now these layers are essentially functions which are callable ok. So, we are calling this function with input layer, we are calling this function with layer 1. So, layer 1 is essentially the definition or output of this particular layer right.

So, from the previous layers output, now we are calling this layer with layer 2 which is the previous layer. So, now you see that we are defining these layers, but we are calling them with the inputs that we are giving. So, for this case we are giving input layer which is actually we have defined the input as and then here we are using layer 1, here you are

using layer 1. So, this definition or this kind of just simple tweak calling of these functions with these parameters or input layers or output layers and output layer will essentially be the output of the your entire model, then you can do the define the model ok.


So, model equal to model and you define with this input and output set. Now, as I was pointing you towards the flexibility of using functional API in the cases, where you will have multiple inputs and multiple outputs. So, here you can define a list of input layers. So, if you have different inputs let us say 2 3 inputs then you can define list of inputs and as output if you have multiple outputs you can have multiple outputs.

So, basically one list you can create that multiple outputs you have. So, this definition actually helping you to define one model with different inputs and outputs and since each layer is callable now you can use this layer in anywhere you can for your target model right. You can call with any let us say I want to call this dense layer with this layer 1 that also you can do. So, you can connect this output layer to that layer.

So, however, you want you can do this. So, this is the flexibility I was talking about for the functional functionality and at last you have the sequence you have the subclassing way of defining. So, this is the `keras.model` which is which will be the super class for this and you will have the initializer. So, same way you have defined for the PyTorch then you connect them and then you define the model the model equal to this, you create the instance of the model right.

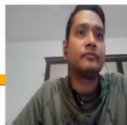

So, simple elegant and you have different ways of defining models depending on your target model and the flexibility you want to have it can use either one to create it or more than one to define the model ok.

(Refer Slide Time: 10:04)



### Why input pipeline is important

- data might not fit into memory
- data might require (randomized) pre-processing
- efficiently utilize hardware
- decouple loading + pre-processing from distribution



So, now we will move towards the input pipeline which mean data pipeline is very very important and when we are using `tf.data`, then a new world of input pipeline opens before you because all the abstractions APIs are already defined. You do not need to worry about and all these will work as your C++ optimized code in the back end.

So, the performance you will get is very high. Let us define why we need actually the input pipeline right. Input pipeline means what you just stack one after the processes from different let us say stack or different layers of your input or output. Now here let us say you want to train one data set and that data set you want to load into your memory right.

So, how you are actually doing reading the file then loading that into memory. So, before that reading also you have this open stage right. You have to open the file let us say you are working with CSV let me or maybe a or big JSON record or maybe whatever it might be right. So, the entire data let us say you have 1 TB of data you do not have the size or the memory available to feed the entire memory to read and then process it right.

So, so data might not feed into memory. So, you have to pipeline the pipeline stages are like whatever however you define the stages. It might be read open then load read open process load. So, whatever the stages you can define. So, different stages you can define depending on the requirement and you can pipeline there right.

Now, so because in each stage you will have some output and you need to store that into your data might require also pre processing as mentioning that in that pipeline you might have some pre processing inside. So, before loading and after reading you might have the pre process.

So, you can pipeline that and efficiently utilize the hardware underlying, you can use the pipeline and you will see how miraculously it will accelerate the entire process actually of data loading plus training because training on your GPU will wait for the day always and its quite fast. So, you need to feed the data right and pipelining the data efficiently will help to accelerate.

So, let us see how to decouple this loading and pre processing. So, as I was mentioning that inside your pipeline you might have data loading and pre processing these two stages and you can decouple them from the distribution of the data. So, loading will be happening in the end and before that you need to pre process it, but how you can actually use efficient pipelining to decouple them and distribute the data for load model training that we will see in the next slide.

(Refer Slide Time: 13:30)



The slide displays a list of pipeline stages under three main categories: Extract, Transform, and Load. The 'Extract' category includes 'read data from memory / storage' and 'parse file format'. The 'Transform' category includes 'text vectorization', 'image transformations', 'video temporal sampling', and 'shuffling, batching,...'. The 'Load' category includes 'transfer data to the accelerator'. The slide also features a logo in the top left corner, the NPTEL logo in the top right corner, and a small video inset in the bottom right corner showing a person speaking. A decorative bar with blue and yellow segments is located at the bottom of the slide content area.

- Extract:
  - read data from memory / storage
  - parse file format
- Transform:
  - text vectorization
  - image transformations
  - video temporal sampling
  - shuffling, batching,...
- Load:
  - transfer data to the accelerator

So, these are common stages of pipeline extract transform and load. So, extract is essentially dealing the data from memory and storage parts of file format whatever you want to do and then so reading data from memory or storage; that means, maybe local disk or maybe remote storage or whatever may be. Transform is essentially let us say

you want to normalize your data or vectorize your data or maybe shuffle your data batch your data right. All these are different transformations that you want to apply then load to the accelerator.

So, this is the load part where you will. So, you have the memory in hierarchy. So, you want to load from your RAM to your accelerator in the end. So, that is the last stage of the pipeline that you see here.

(Refer Slide Time: 14:27)

The slide is titled "The naive approach" in red text. It features a code snippet in a dark box and a Gantt chart below it. The code defines a benchmark function that iterates over epochs, samples data, performs a training step, and sleeps for 0.01 seconds. The Gantt chart, labeled "Naive", shows four horizontal bars representing different stages: "Data" (blue), "Read" (purple), "Train" (pink), and "Accumulate" (orange). The bars are sequential, indicating that each stage must wait for the previous one to complete before starting. The x-axis is labeled "time (s)".

```
def benchmark(dataset, num_epochs=2):
    start_time = time.perf_counter()
    for epoch_num in range(num_epochs):
        for sample in dataset:
            # Performing a training step
            time.sleep(0.01)
        print('Execution time:', time.perf_counter() - start_time)

benchmark(ArtificialDataset())
```

[https://www.tensorflow.org/guide/data\\_performance](https://www.tensorflow.org/guide/data_performance)

So, let us see the naive way to do this right. So, first you have obtained your record or code or file or whatever you have and then you read right. So, let us say you have read this chunk of data ok depending on the size that you want to read into your memory and then you feedback to your training model.

So, then you train them right and then again you would read the data and then again claim it when the data whichever you have read then again read time frame right. So, this is the simple sequential way of doing things this is. So, if you are not doing anything to make the pipeline available. So, this is not pipeline here this is sequential and so this is how it will happen if you do not apply the efficient pipelining that is available which we are talking.

So, this is the simple way of defining. So, if you like the code, let us say this is your training loop right. So, for epoch for sampling data sets, so for each data set you just do



this training. So, `time.sleep` is essentially emulating the training step basically let us say I am saying that wait for this time to see how data my data is available or not right.

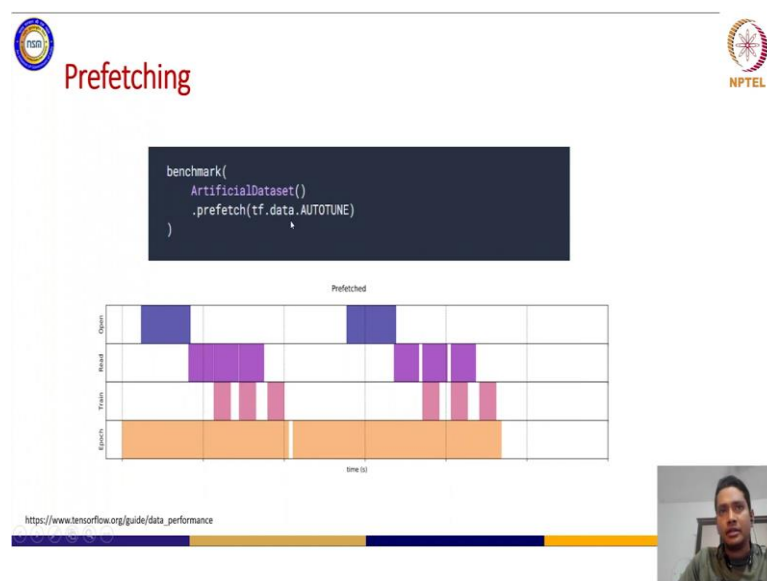
So, this is a simple example to emulate this pipelining process which is referred to your tensorflow.org guide data performance. You can go to this link and see the entire code here, but we are showing you the how to use this pipe ok. So, now this did this benchmark ok we want the benchmark with the data set that we have created, let us say artificial data set that we have created.

Now, total number of epochs I want to run for two epochs and it will iterate through the data set again this data set is tensor flow data set. So, it is iterable and you can iterate through the data set and you can feed that into the training model. So, basically here you will have the training a forward pass and backward pass and update right. So, instead for simplicity we are keeping it wait that is it and at the end you will have this performance counter ok. So, just to see how much time it is taking.

Now, the thing is that here this is the naive way of implementing a training model where you have will have the data set. So, this is exactly replicating the process that you have you are seeing here on the figure. But what we want to have? We want to have the pipeline. Now, you will see that the CPU is reading the opening the file here and reading the file here and waiting for the training to be finished here and then it will load again the data and then again the training will start.

Now, these white blocks you are seeing here the CPU and GPU is essentially waiting for the data and training to be completed right. Now, so read is done by let us say CPU and train is done by your GPU. So, reading here CPU is waiting here and GPU is waiting here and here similarly for this one.

(Refer Slide Time: 17:55)



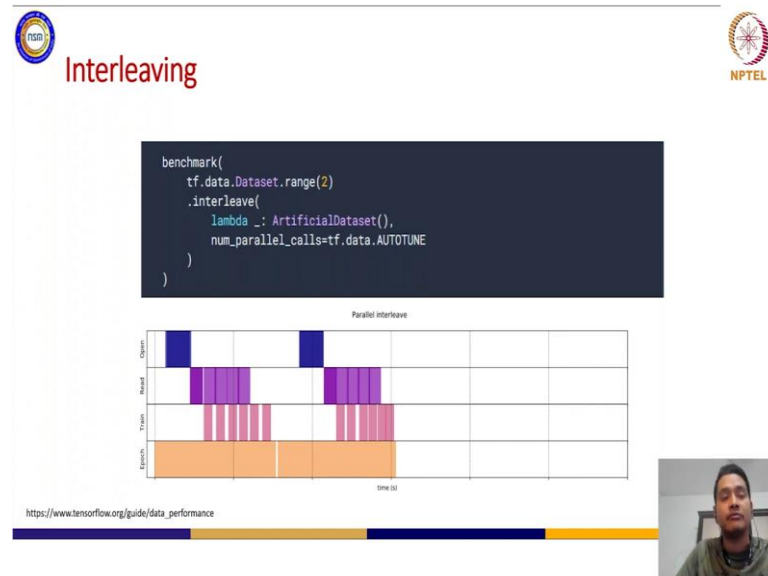
Now, I want to have overlap; that means, when the GPU is training CPU was not doing anything, I want to pipeline it. So, that the next set of data will be read and it will wait for the training to be start. So, basically if you see that the training was done here and GPU was waiting for this short time. Earlier it was very large ok. Now, the waiting time for the GPU is shortened and then again since the data is loaded here then GPU can start this training and again the data can be read by the CPU in this blank time or idle time right.

And then again it will start GPU will start the training. So, I can see that we are prefetching the data. So, this is one way of arranging your pipeline. So, that the data can be pre fetched for your CPU by your CPU to be available for the GPU to train simultaneously and how you can do that? It is very simple just you call pre fetch with this parameter `tf.data.AUTOTUNE`. What is this doing? This is saying how many pre fetches it will do while it is idling.

So, if you want to define one number here 3 2 whatever, depending on your environment if you know your CPU well, if you know your GPU well and if you know what kind of size of the RAM and size of the data you are using. You can define this number manually rather you can take the advantage of `tf.data` runtime which will define this autotune by automatically setting analyzing the environment and set this number for you ok.

So, that is the best way to do things. So, this pre fetching you can reduce the idle time for the GPU and accelerate training.

(Refer Slide Time: 19:55)



Interleaving if you have let us say different data sets to be read. So, let us say here two coloured reading is doing, but they are being interleaved; that means, one is being done at the same time and at the next time stamp another one from the data set another data set will be read somewhere else.

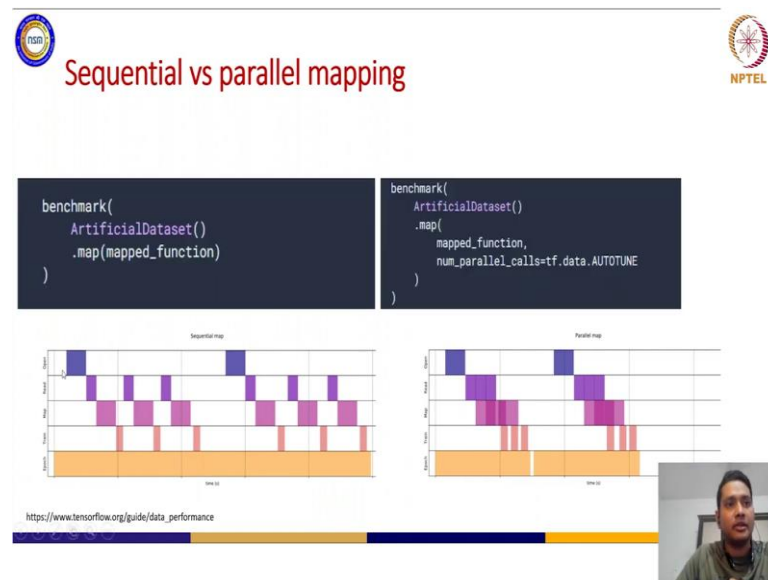
So, you can see that interleaving of different data sets we can do also and this interleaving is essentially useful for the cases where you will have the remote storage. So, I mentioned before starting this interleaving and pipelining things that you need to know your environment. So, when you are reading your data from your remote storage there the bottleneck is io bottleneck because reading and writing will have different time from different sources you are reading right.

Now, if you interleave whatever is available that will fetch the data read the data. So, that way in that case you will have accelerated training and the idle time for the GPU will be very less and if you were using simple reading with the io bottleneck then I mean it is horrible to read ok.

So, the and you can parallelize this interleaving process. So, that interleave will actually help you to interleave how different data set reading and number of parallel calls again

you can define the number here or you can use `tf.data.runtime` and the `autotune` attribute will actually fetch the number yours by the runtime itself. So, you do not need to wait. How many parallel calls will be there for the residual.

(Refer Slide Time: 21:47)



We can do parallel mapping now mapping is very very important transformation that you will do mostly a file doing this or creating this data map. So, earlier we had open sorry earlier we have to open read and the train, but here we now have map. Now, you want to pre process the data before loading this. This is another stage in that the pipeline that we have introduced here.

Now, what if you will do here? This is the naive way the sequential mapping, where you just open read map sequentially main frame again wait for the CPU to get the training is done message from the GPU, then it will again read the next set of data. But I want this to be happening parallel. The mapping to be happening parallel while it is waiting for the next data to be read and pre processed you see how much time it is waiting. I want to just reduce this waiting time collapse all these rates collapse the mapping do the parallel mapping.

And you see the waiting time for the GPU has been reduced to from this to this. You will have accelerated you will have accelerated mapping again ok next ok sorry. So, now how you will do that? You just call `.map`, so basically this is for the map transformation you want to apply on your data set and this is the map function.

So, map function is the transformation you want to do maybe let us say you want to increment your data before loading or maybe you normalize your data by extending it by 255 as given for your images that we process you want to do it in this function map function.

Any pre processing, but this is the simple way you will do definitely, but number of parallel calls if you define it with the autotune. It will just parallelize your reading fetching and mapping or pre processing. Now, remember one very tiny details that we will add into the next section ok, when we will cache these basically all the things we are doing here still now it is just happening how it will how the CPU and GPU here the CPU is doing the mapping and reading.

If you are asking why not GPU is doing the mapping or maybe for this case also this case parallel case or sequential case GPU cannot do the method because GPU do not have the general purpose processing unit right. So, CPU has. So, CPU can do the pre processing. So, there is no way GPU can do the pre processing. So, it can do only matrix multiplication or math operation or graphics operation by tracing operations.

So, apart from that it cannot do anything. So, map you have to do inside the CPU. So, CPU will do the mapping. So, the map number of parallel calls you have to define.

(Refer Slide Time: 25:20)



Caching is important, because now I want to say that why you are applying transformations every time you do the transformation you transform all the data and keep it in the cache or memory or the file wherever you want with this cache function that is it.

So, what is happening here you can see that for these set of training it is not fetching not doing the read and transformation as opposed to the previous method right because your all your transformed data is inside your memory ok. So, it is transforming all the data keeping it in the memory and it is not doing anything, but the that is it.

So, for the next epoch let us say this is the epoch. For the first epoch yes, you do the transformation and load the data, but since that was the time that CPU is taking to pre process the data and loading to your memory and in the next epoch it does not do anything, CPU does not do anything because the already the data is already present inside your memory.

So, then the GPU will just train ok, but one tiny thing again apply time consuming operations before cache and apply memory consuming operations after cache. So, this is the thumb rule for caching. If you have very much time consuming operations you do before cache because that way in the first epoch you will have bit performance here and there neglected, but once the pre processing is done then you will have much faster training in the next epochs right.

But, here if you have this function this transformation function which is memory consuming then you cannot do before cache because you cannot apply all the data to be stored in the memory by caching because it is memory consuming transformation. So, you do that after cache ok. So, this is the simple thumb rule you need to keep in mind ok.

(Refer Slide Time: 27:41)

The slide is titled "Scalar vs Vectorized mapping" and features the NPTEL logo in the top right corner. It compares two methods of applying a function to a dataset using TensorFlow's `fast_benchmark` function.

**Scalar Map:** The code on the left shows a function that applies a transformation to one item at a time. The corresponding timeline graph shows a sequential process where the mapping function (purple) is called repeatedly for each item, leading to significant idle time for the data reader (orange).

```
fast_benchmark(  
  fast_dataset  
  # Apply function one item at a time  
  .map(increment)  
  # Batch  
  .batch(256)  
)
```

**Vectorized Map:** The code on the right shows a function that applies a transformation to a batch of items. The corresponding timeline graph shows a more efficient process where the mapping function (purple) is called once for the entire batch, significantly reducing the idle time for the data reader (orange).

```
fast_benchmark(  
  fast_dataset  
  .batch(256)  
  # Apply function on a batch of items  
  # The tf.Tensor.__add__ method already handle batches  
  .map(increment)  
)
```



The URL [https://www.tensorflow.org/guide/data\\_performance](https://www.tensorflow.org/guide/data_performance) is provided at the bottom left. A small video inset of the presenter is visible in the bottom right corner.

Now, you have scalar and vectorized mappings. So, now, you can see here we are opening the file reading the file mapping the file. Now, map is doing the transformation and map is called after each scalar data let us say I want to normalize it or so each data will be called then map function will be called and then it will map then next data will be read next map will be done.

So, I do that. You can do only one map with all the batch of data, but one time that is it. How you will do that? This is the vectorized map, how you will do that? So, this is the simple way of doing it batching because in the batching you will actually batch and then map ok and you do the batch before mapping ok.

So, because now we are vectorizing the mapping equation, we are giving the batch of data to the mapping to get transformed and you can see that all this waiting time for the mapping is in just collapsed and you get a very efficient timing for your CPU as a usage will be more and the train will be faster.

(Refer Slide Time: 29:07)



```
import tensorflow_datasets as tfds


# Download the dataset and create a tf.data.Dataset
ds, info = tfds.load("mnist", split="train", with_info=True)

# Access relevant metadata with DatasetInfo
print(info.splits["train"].num_examples)
print(info.features["label"].num_classes)

# Build your input pipeline
ds = ds.batch(128).repeat(10)

# And get NumPy arrays if you'd like
for ex in tfds.as_numpy(ds):
    np_image, np_label = ex["image"], ex["label"]
```

<https://colab.research.google.com/github/tensorflow/datasets/blob/master/docs/overview.ipynb>



Now, all these state of data that we have talked about, but we have also mentioned the tfds which is the tensor flow data set is essential to be used for your training and it has lot more predefined data set that you can use like here we are using let us say mnist data set that we want to use for our loading and splitting train.

So, you use tensor flow dataset. So, tfds what will you use to load your data and batch it or whatever you want to do in the in the next subsequent processing and to get more details how you can use this tensor flow data set. You go to the link provided here from the tensor flow developers. So, you can you can see through the code complete training module for using the tfds.