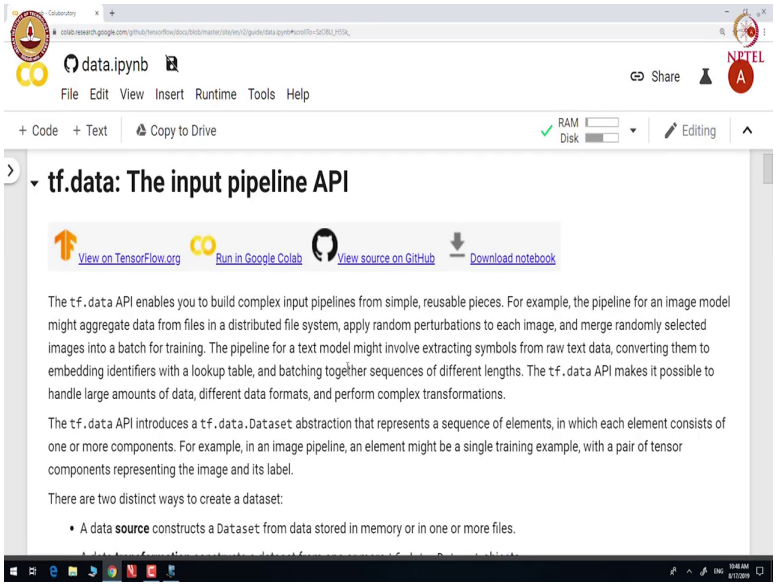
Practical Machine Learning
Dr. Ashish Tendulkar
Department of Computer Science Engineering
Indian Institute of Technology, Bombay

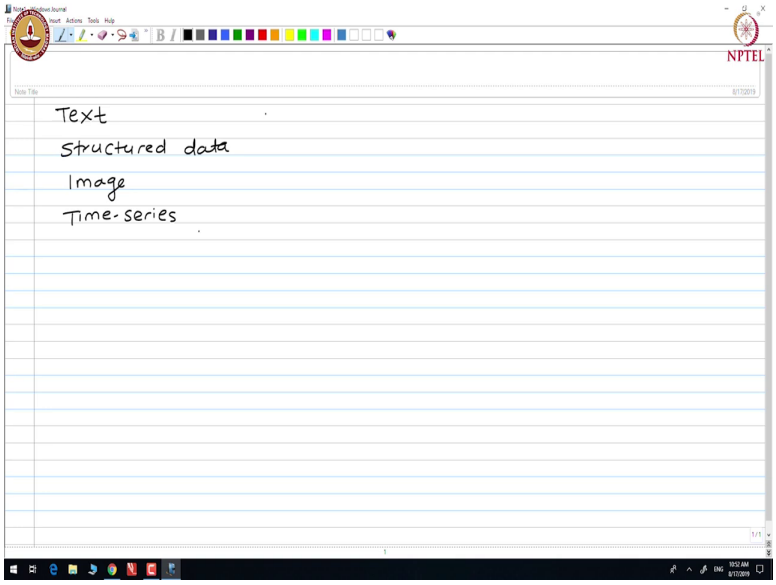
Lecture – 12
Building Data Pipelines for Tensorflow- Part 1

(Refer Slide Time: 00:13)



[FL] We will study how to build input pipelines for TensorFlow. As you know, TensorFlow processes different kind of data sets. We have already seen couple of those kinds of data sets in this course so far: one was the Fashion-MNIST dataset.

(Refer Slide Time: 00:43)



Apart from image dataset, then TensorFlow supports text data, it also supports structured data and sequence data sets like time series data. You can see that the data sets are quite varied, and their requirements are also quite different. For structured data, we want to pre-process the data by say normalizing it or by getting rid of missing values. In case of images, we might want to read images from multiple files on the system. Then we do image augmentation by rotating the images or by applying other kind of transformations on the image.

In case of text data, we want to read the text data, and extract tokens from the data by processing the text before we can use it in TensorFlow. Other important thing that we do in text data is to obtain embeddings for the words and convert words into integers or into some numeric data. So, the API for input pipeline building should be versatile to support all these kinds of operations on the varied data.

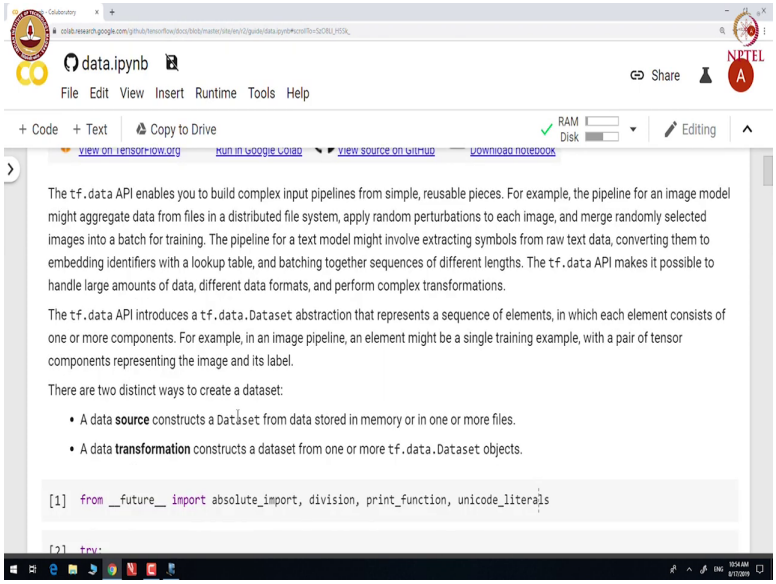
Apart from the variety of structures, the data can be very big that it may not be possible to fit that data in memory or data may be small data that can be easily fit in memory. So, this API also should support both the scenarios where it should be equally easy for the programmers to read data in memory as well as data that is sitting on the disc. So, tf.data API that is implemented by TensorFlow supports input pipeline building. So, the tf.data API enables us to build complex input pipelines from simple reusable pieces. It makes it

possible to handle large amount of data different data format and perform complex transformations on the data. It introduces *tf.data.dataset* abstraction that represents a sequence of elements, in which each element consists of one or more components.

To take a concrete example of an image pipeline, an element might be a single training example with a pair of tensors representing image and its label.

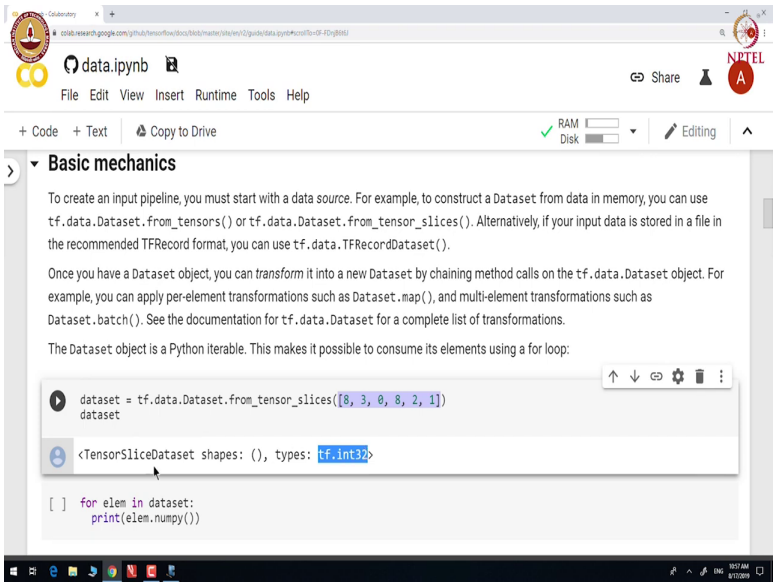
How do we create these datasets? There are two distinct ways of creating the dataset. We construct the dataset from data source, where the data might be stored in memory or it could be stored in one or more files.

(Refer Slide Time: 05:05)



We can also perform transformations on one or more dataset to get a new dataset object.

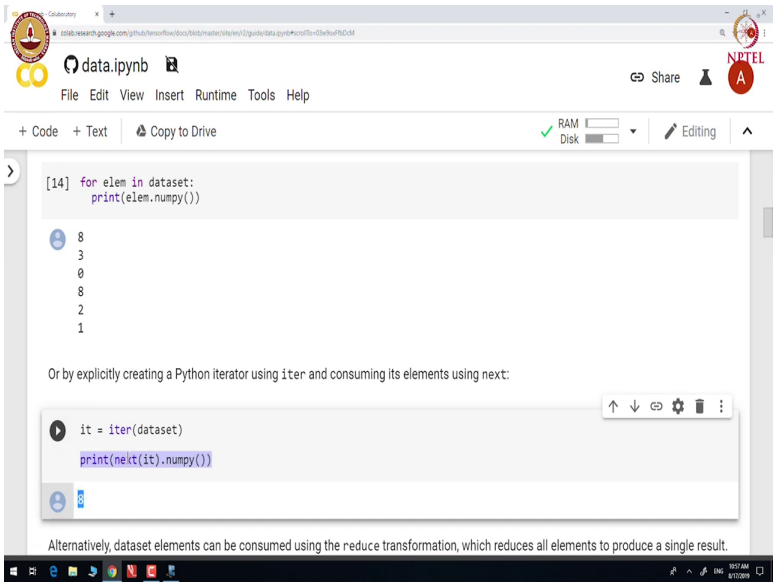
(Refer Slide Time: 05:19)



Let us try to understand basic mechanics of the data source and how to create the input pipeline from the data source. So, we will first construct data source based on data that is in memory. We can use one of the two functions: *from_tensors* or *from_tensor_slices* from *tf.data.dataset* API.

Alternatively, if a data is stored in TFRecord format, we can use the *tf.data.tfrecord* dataset for reading that data. Let us construct a dataset from tensor slices, here the data is stored in an array there are 6 elements. Let us construct the dataset using *from_tensor_slices()* function. You can see that the resulting dataset is a scalar because the shape is *null*, and it stores *int32* type of elements in it.

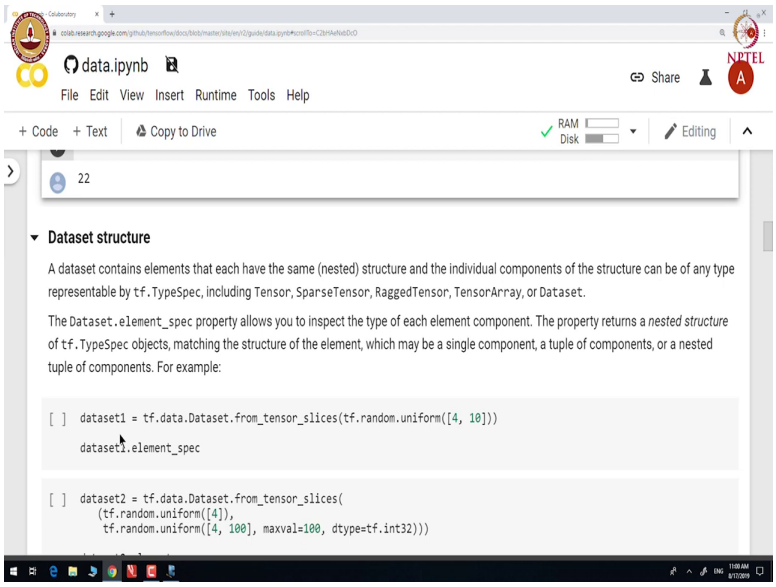
(Refer Slide Time: 07:01)



Let us look at elements in the dataset. You can see that this particular dataset has 6 scalars which are integers. This dataset is constructed from this particular array. We can also use python iterator for iterating over the dataset. We can use *next()* to get the next element in the dataset. So, here we show you the first element of the dataset which is 8.

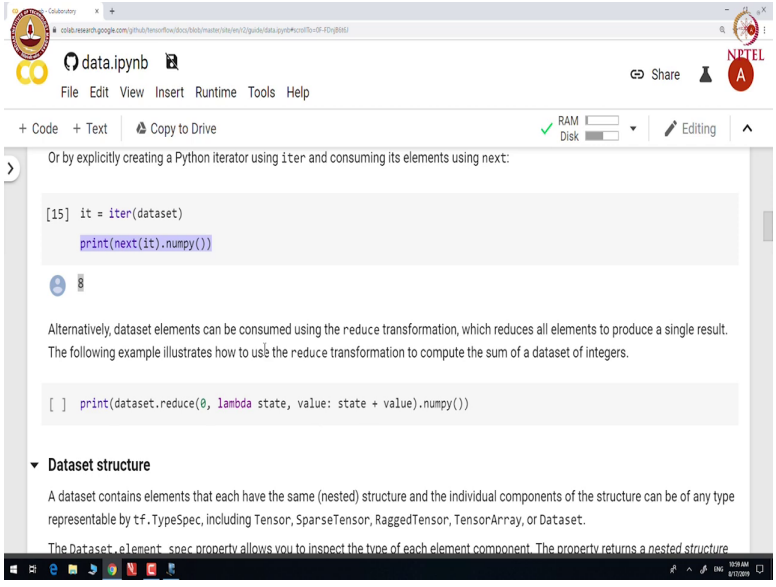
Apart from creating the dataset from the data that is stored in memory or in files, we can also perform transformation on the existing dataset objects to obtain a new dataset. We can apply this transformation on every element in the dataset using a *map()* function, or we can apply this transformation on multiple elements using a *batch()* function.

(Refer Slide Time: 08:31)



So, let us try to use a reduced transformation that reduces all the elements to produce a single result. To take a concrete example, we take the dataset that we constructed here, and we reduce the dataset to its sum.

(Refer Slide Time: 08:53)



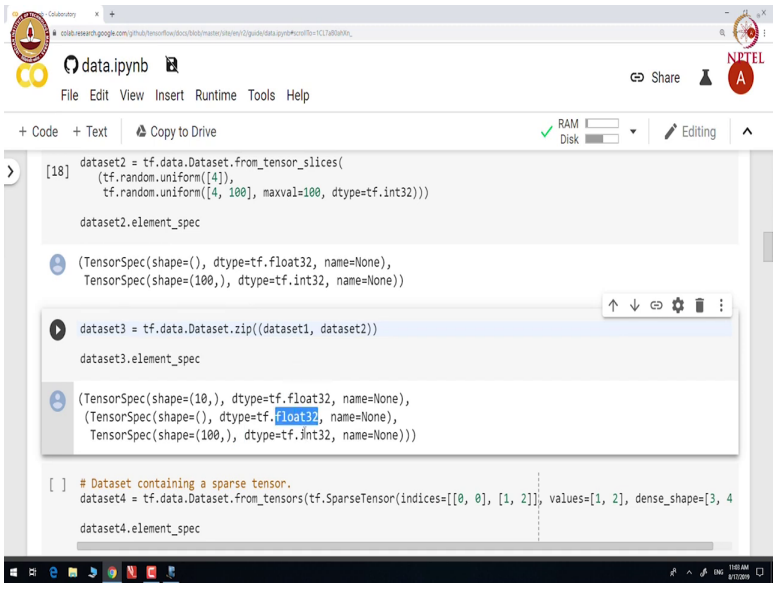
So, here we use a *reduce()* where we give the initial state which is 0 and then we define a *lambda()* on state on the and value, where we add the existing state to the value, and store the results into the state variable. As we go through the entire dataset, the effect of

this particular reduced transformation is to get the sum of dataset of integers. Let us run it and check the results. We can see that all the elements in this data source sums to 22 which is what is the output that we got here.

Let us understand the structure of dataset. A dataset contain elements where each element has the same structure, and individual component of the structure can be of any type represented by *tf.TypeSpec*. This includes sparse tensors, tensors, tensor array or dataset. Dataset.element_spec property allows us to inspect the type of each element component.

So, let us check out the element_spec of a dataset here in a concrete example. So, here we construct the dataset from tensor slices, where we have created a 4 x 10 tensor with random values from uniform distribution. Let us run this code cell to understand the element specification. We can see that each element in this dataset is a vector or 1D tensor containing 1D tensor with shape of 10, and each value stored in this tensor is a float value.

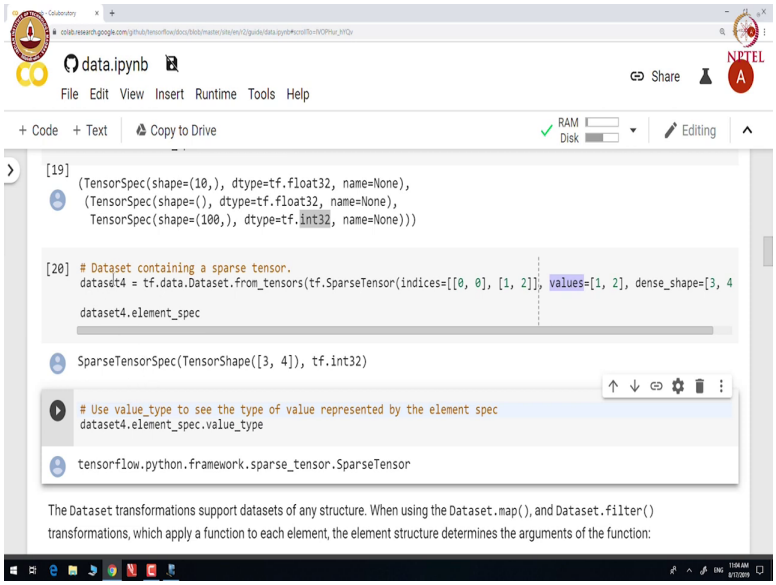
(Refer Slide Time: 11:47)



Let us construct another tensor and here you can see that this particular tensor stores ordered pairs of two tensors: one is a scalar and second is a vector. We can zip dataset1 and dataset2 and inspect the element_spec of the resulting dataset. We can see that the resulting dataset has element which is a pair of tensor where the first tensor is a vector or

1D tensor with shape 10, then we have a scalar followed by another 1D tensor with shape of 100. We store float values in first two tensors, and integer values in the last tensor.

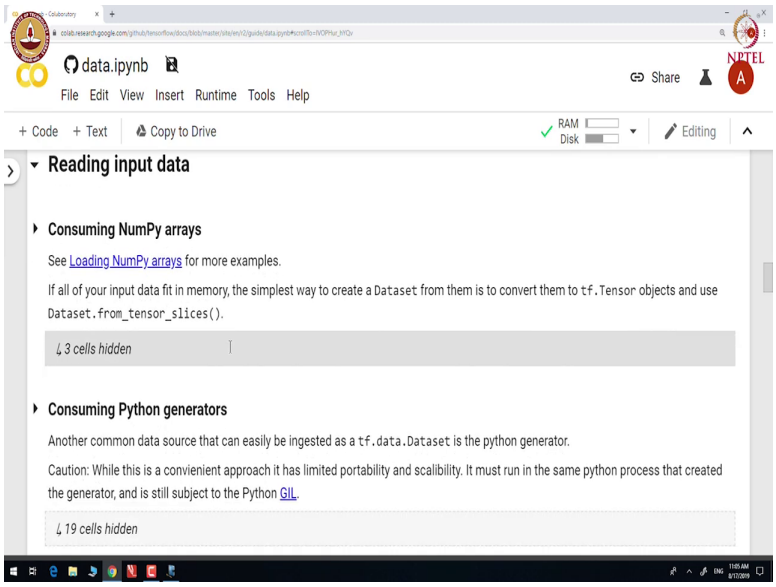
(Refer Slide Time: 13:05)



We can also create a sparse tensor where we specify indices where there are values present in the tensor, and provide the values in the values argument. And it also give the dense shape of the tensor. Let us look at the element specification of this sparse tensor. So, we can see that we have a sparse tensor, where the tensor is a 2D tensor with shape 3 x 4 and each element is a 32-bit integer.

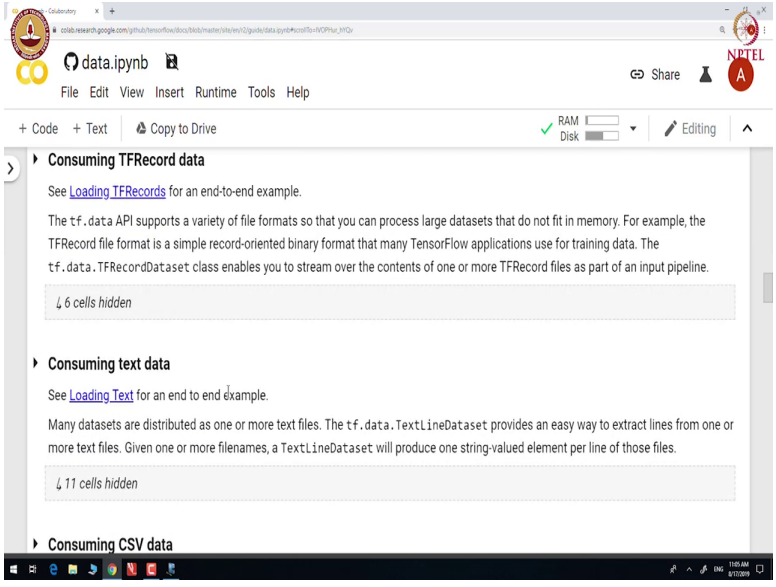
We can use *value_type* to see the type of the value represented by the element spec. So, we can see that in this dataset 4 each value is a sparse tensor. The dataset transformation supports dataset of any structure. When using map and filter transformation which apply a function to each element, the element structure determines the argument of the function.

(Refer Slide Time: 14:33)

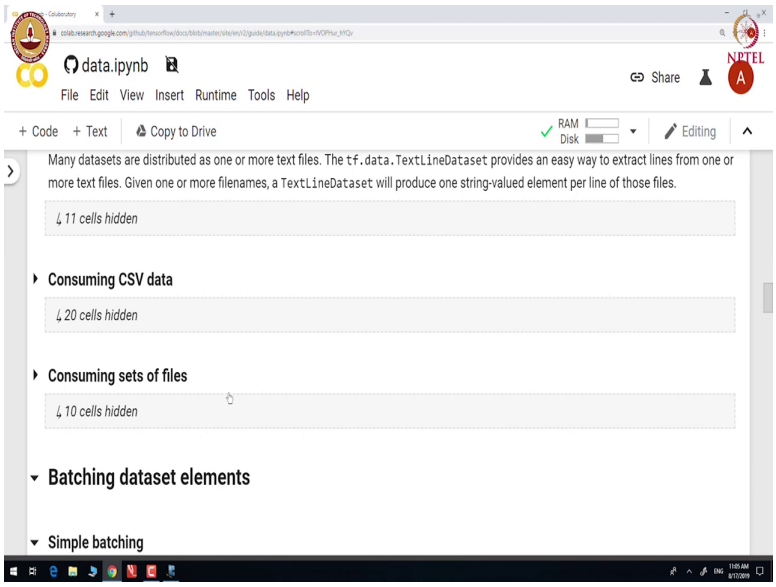


So, we can construct dataset from different sources. We can construct it by consuming NumPy array, by consuming python generators, TFRecord, by consuming text data or CSV data or by consuming bunch of files.

(Refer Slide Time: 14:47)

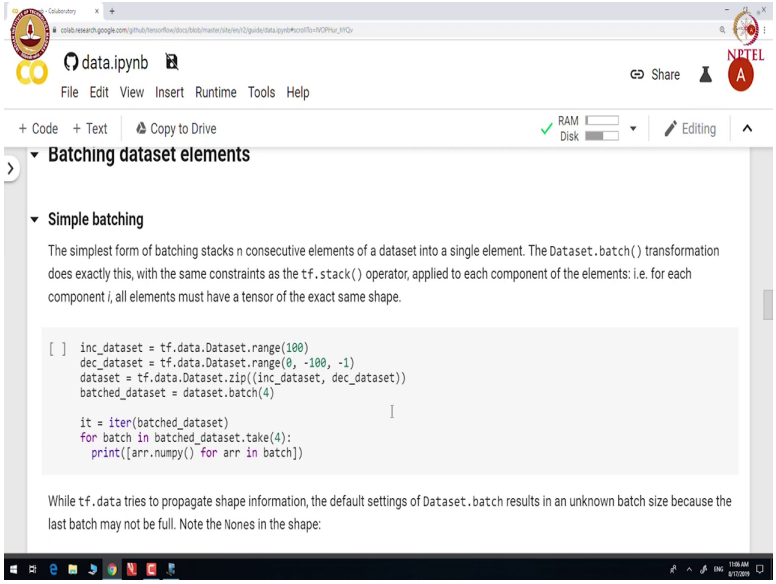


(Refer Slide Time: 14:51)



So, in the next session, we will go through example of each of these data type and see how to construct a dataset object based on these different formats. Let us look at some of the operations on the dataset element.

(Refer Slide Time: 15:17)

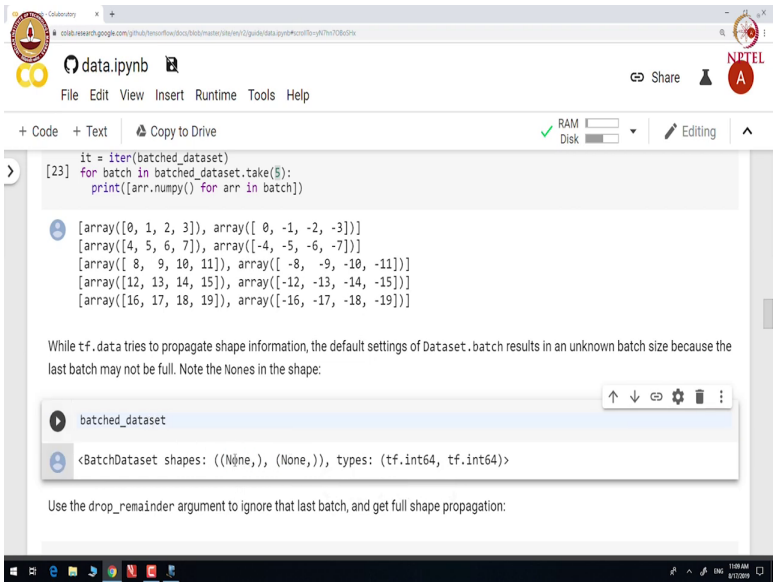


One of the important operation that we use during the training is batching. The simplest form of batching stacks and consecutive elements of a dataset into a single element. We use batch transformation with same constraint as tf.stack( ) operator. The batch is applied

to each component of the element. And here there is a condition that all elements must have a tensor of exactly the same shape.

Let us see a concrete example of batching. So, here we construct two dataset. One is inc_dataset containing values between 0 to 100, and then dec_dataset containing values between 0 to -100. We construct a new dataset by zipping both these data sets. We construct a batch of 4 elements using batch transformation, and we call that dataset as batched_dataset. Let us iterate our batched_dataset and see the elements in the batched_ dataset.

(Refer Slide Time: 16:51)

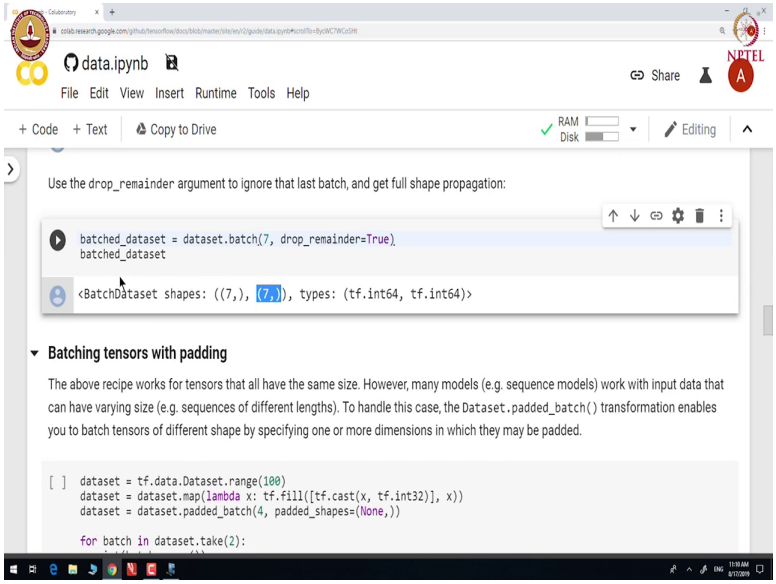


So, you can see that since we have constructed batches of 4, from the inc_dataset we get 0, 1, 2 and three as the first batch of four. From the dec_dataset the first batch of four has 0,-1, -2, -3. The second element starts at 4 for the inc_dataset, whereas dec_dataset it start at -4 and goes up to -7. And we print first four elements by using the take() function and using 4 as an argument to the take() function. We print first four elements in the batch.

Let us try to change it to 5 and you can see that you have we have five elements that are captured here or the first five entries in the batched_dataset are printed on the screen. While *tf.data* tries to propagate shape information, the default setting of *dataset.batch*

results in an unknown batch size, because the last batch may not be full. So, we can check the shape of the batched_dataset and we can see that there is a *none* in the shape.

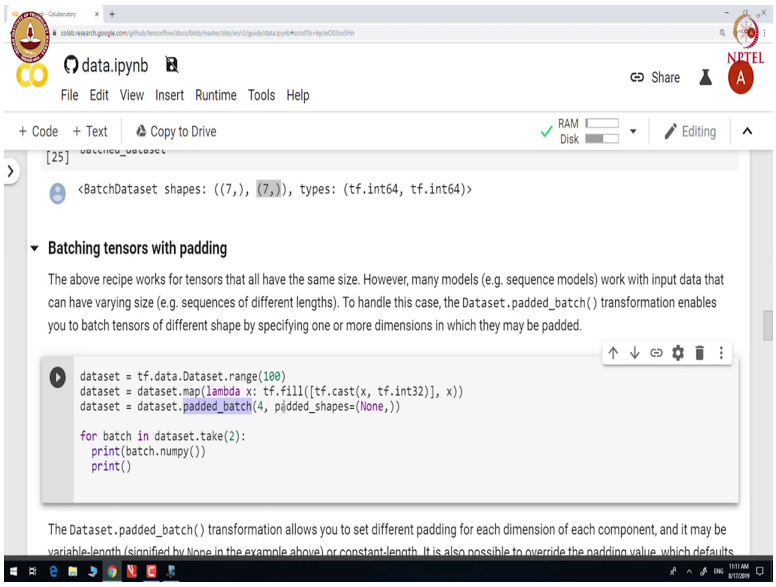
(Refer Slide Time: 18:43)



We can use drop_remainder argument to ignore the last batch and get full shape propagation. So, let us say if we batched dataset into a batch size of 7 and said drop remainder = *true*, we will checkout the shape of the resulting batch. Now, we can see that the batch is a full shape which is each element or each tensor is a 1D tensor with shape of 7. So, this is particular recipe works for tensor that have the same size.

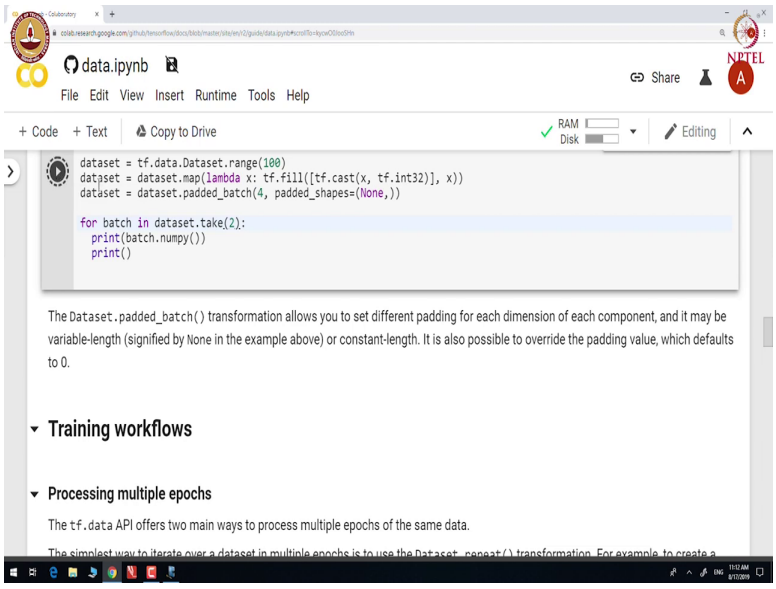
However, there could be many models which might have varying size of tensors. To handle this case, we use *padded_batch* transformation. It enables us to batch tensors of different shapes by specifying one or more dimension in which they may be padded.

(Refer Slide Time: 20:05)

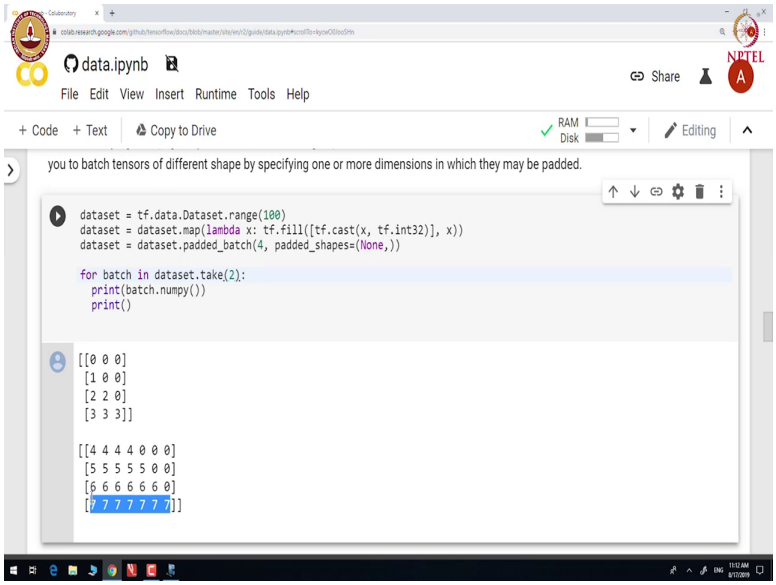


Let us look at a concrete example. So, we can start the dataset of elements between 0 to a 100. And then we define a *lambda()* function that repeats the element by the element time. For example, the number 1 will be repeated once, number 2 will be repeated twice, number 3 will be repeated twice and so on. And we applied *padded_batch* transformation on the dataset by specifying *padded_shape* and the number of elements in each batch which is 4.

(Refer Slide Time: 20:53)

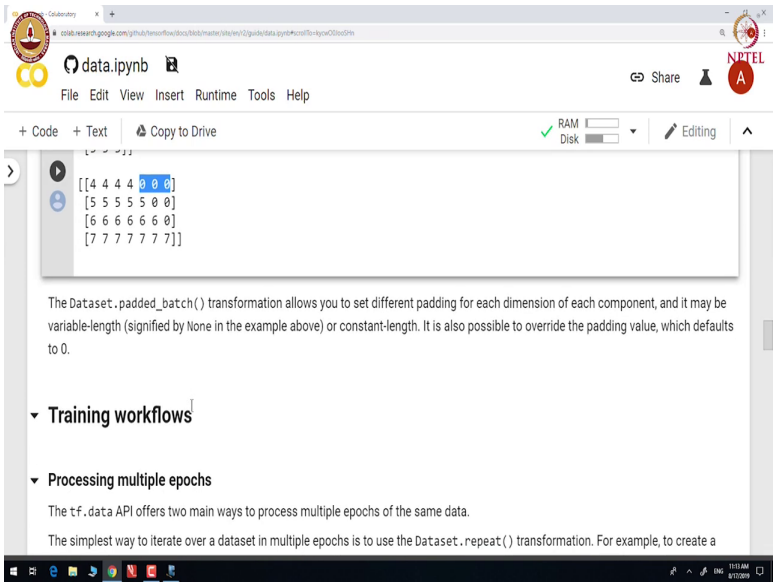


(Refer Slide Time: 21:07)



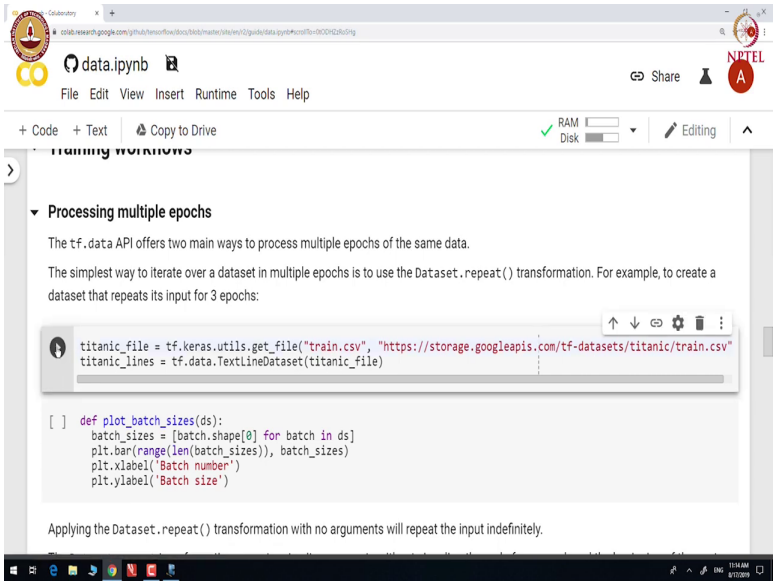
Let us look at the first two batches. You can see that in the first case since the last element which is 3 is repeated three times. We have padding of 3 applied on the first element, padding up 2 as applied on the on the second element, padding up 1 was applied on the third element. Second batch has 7, which is repeated seven times. So, each element from 4 to 6 are extended to shape 7 by padding the required number of zeros.

(Refer Slide Time: 21:59)



So, the padded_ batch transformation allows us to set different padding for each dimension of each component. It is also possible to override the padding value which was 0 in the example.

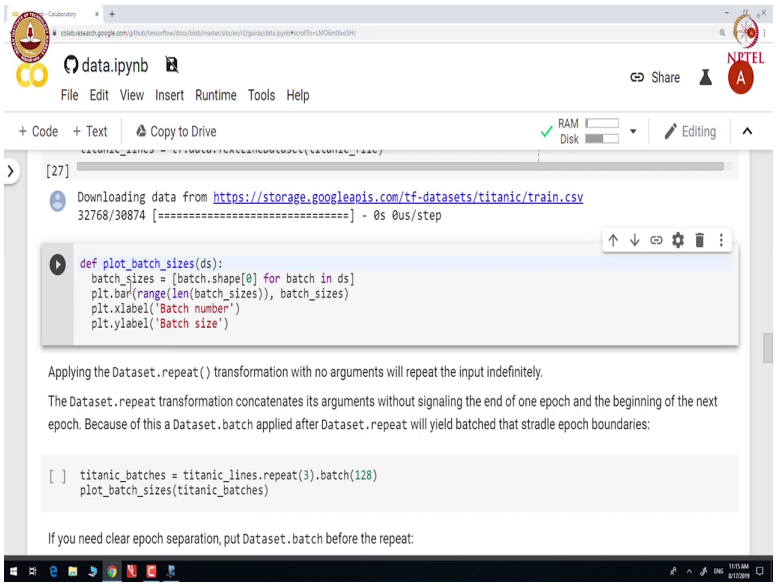
(Refer Slide Time: 22:31)



Let us look at some other training workflows with respect to the datasets. When we train neural network or any other machine learning model, we make multiple pass over the dataset. One complete pass over dataset is known as epoch. And in many batch grading descent, we use a small batch size to update the parameter value.

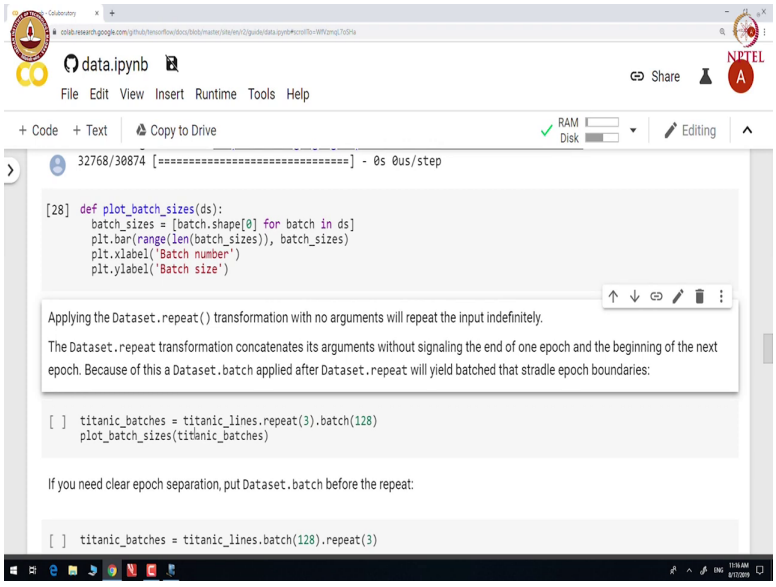
So, what do we need to do is we need to support the repeat transformation in the dataset. Dataset as a repeat transformation that enables us to iterate over a dataset in multiple epochs. Let us create a dataset that repeat its input for 3 epochs. We read we construct a dataset from a csv file.

(Refer Slide Time: 23:53)



We define a function to plot the batch size to understand the effect of repeat and any other transformations that will be applying.

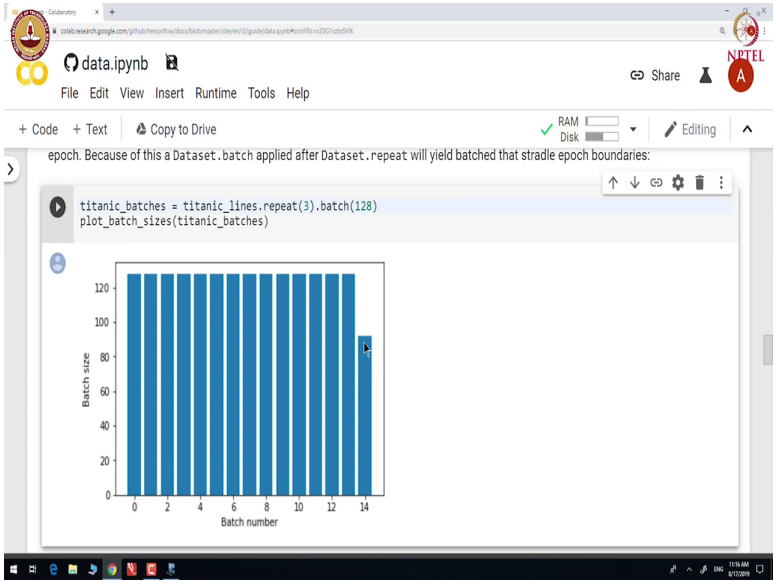
(Refer Slide Time: 24:17)



So, you apply repeat transformation with no argument to repeat the input infinitely. The repeat transformation concatenates its argument without signaling the end of an epoch and the beginning of the next epoch. Because of this a *dataset.batch* applied after *dataset.repeat* will yield batches that straddle epoch boundaries. So, let us repeat the

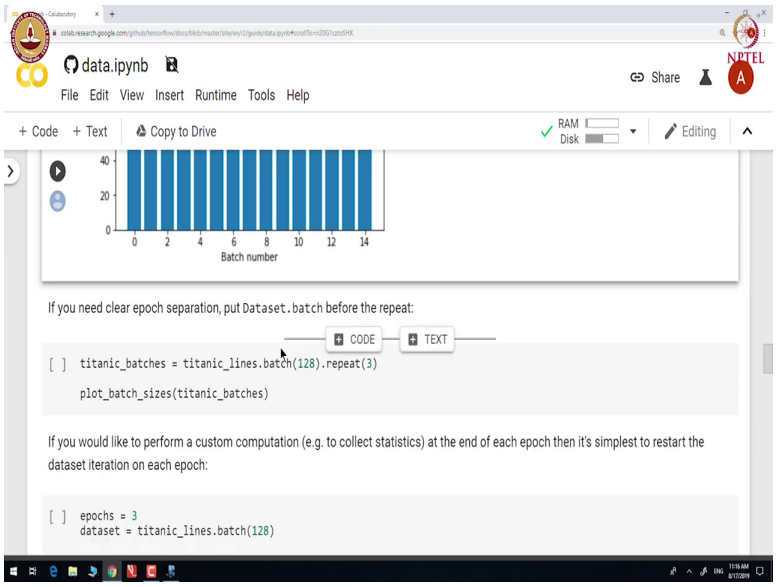
dataset 3 times and we process the dataset with a batch of 128, and let us plot the batch sizes.

(Refer Slide Time: 25:07)



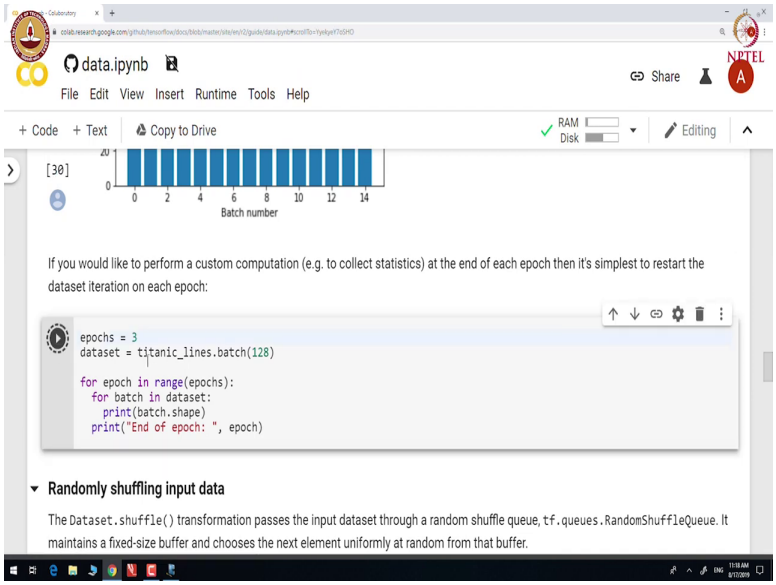
You can see that for different batches the batch size was constant which was 128 except the last batch. So, you can see that we applied batch after repeat and that causes the batches to straddle the epoch boundaries.

(Refer Slide Time: 25:31)



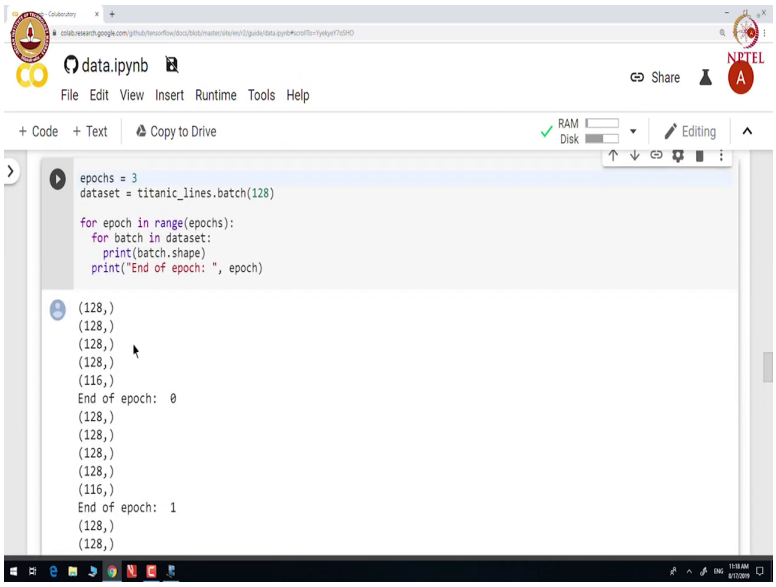
Now, let us apply repeat after batch and see what happens. Now, you can see that when we apply a batch, when you apply repeat after batch, we can see incomplete batches after every 4 batches. So, you can see that here the batches are not straddling the epoch boundaries when we apply batch before repeat. In this case, it helps us to clear epoch separations.

(Refer Slide Time: 26:37)



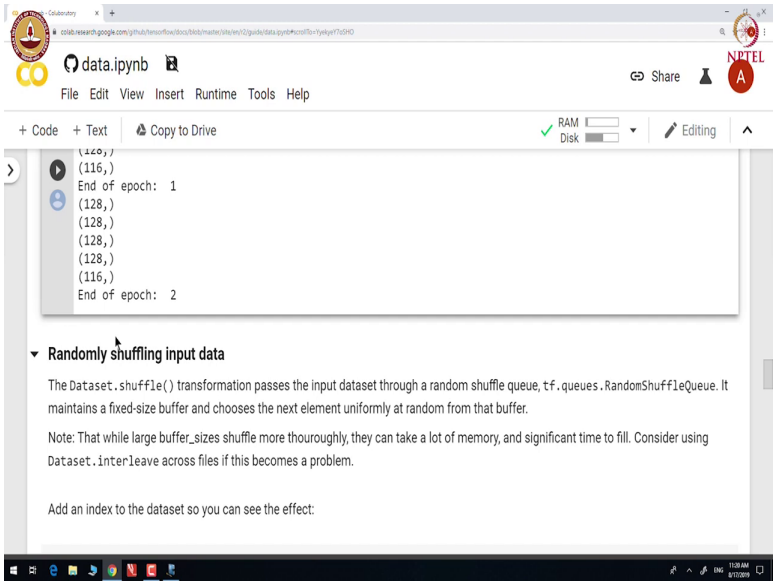
If you want to perform a custom computation at the end of each epoch, then it is simple to restart the dataset iteration on each epoch. So, let us say we want to print the shape of the batch and also print the epoch id at the end of the epoch. We essentially do that in two loops two for loops. The first for loop is for epoch, and within epoch we use batches on the dataset. So, here we define epoch to be 3, and we construct a dataset by batching the titanic_lines dataset into batches of 128.

(Refer Slide Time: 27:31)



There are first four batches with 128 examples in it, whereas the fifth batch has 116 examples. And you can see this happening epochs after the epochs.

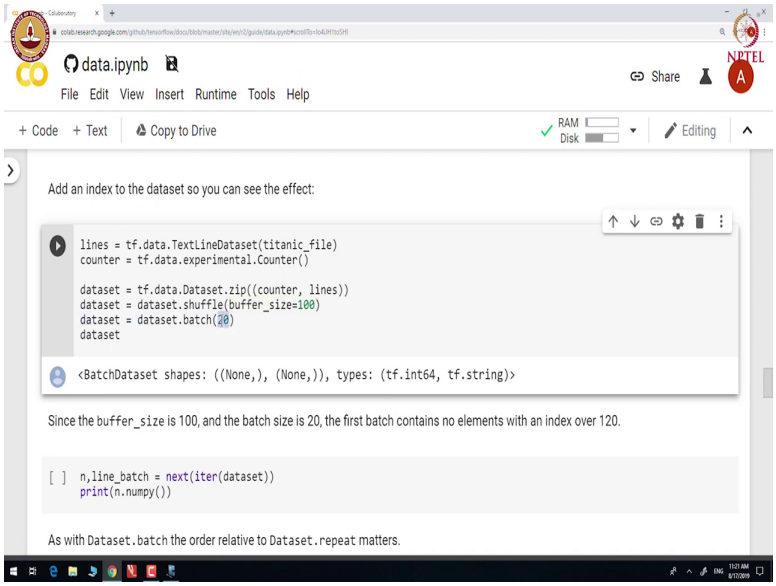
(Refer Slide Time: 27:47)



Other important transformation is shuffle. And we have seen in some of the earlier classes that is it that it is important to shuffle their training data to remove any systematic temporal biases that are present in the dataset. So, *dataset.shuffle* transformation helps us to shuffle the dataset. Shuffle maintains a fixed size buffer and chooses the next element

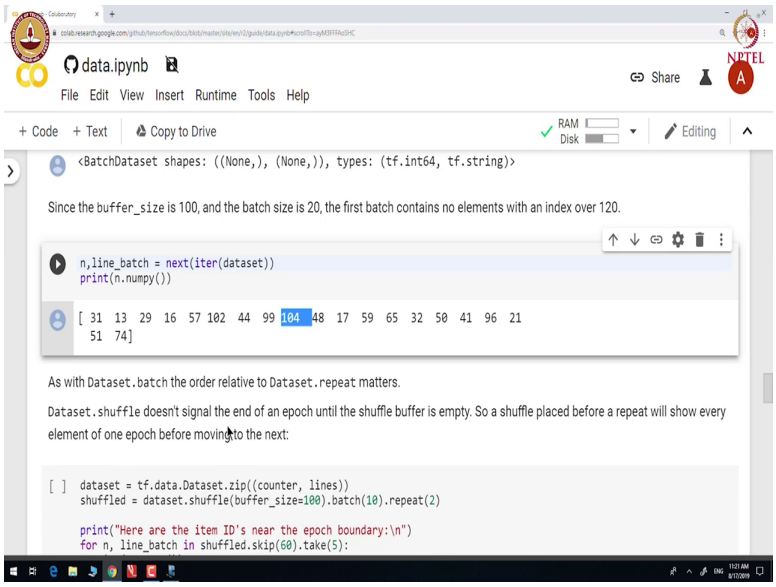
uniformly at random from that buffer. The large buffer sizes shuffle more thoroughly, but they take a lot of memory and significant time to fill. In such cases one can try in interleaved transformation across files which heals the similar kind of effect and shuffle transformation.

(Refer Slide Time: 29:13)



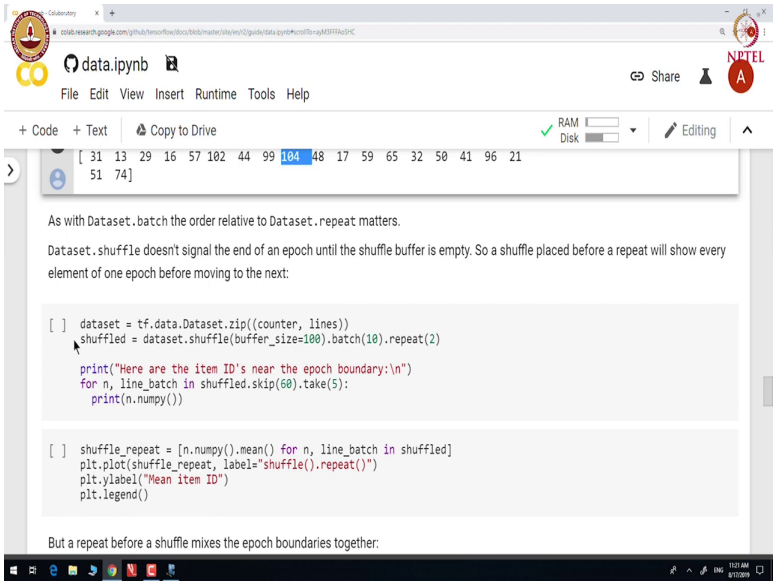
So, let us try to use the shuffle transformation on the titanic dataset. So since the buffer is 100 as you are defining it over here and the batch size of 10 and the batch size of 20, the first batch contains no element with an index or 120.

(Refer Slide Time: 29:47)



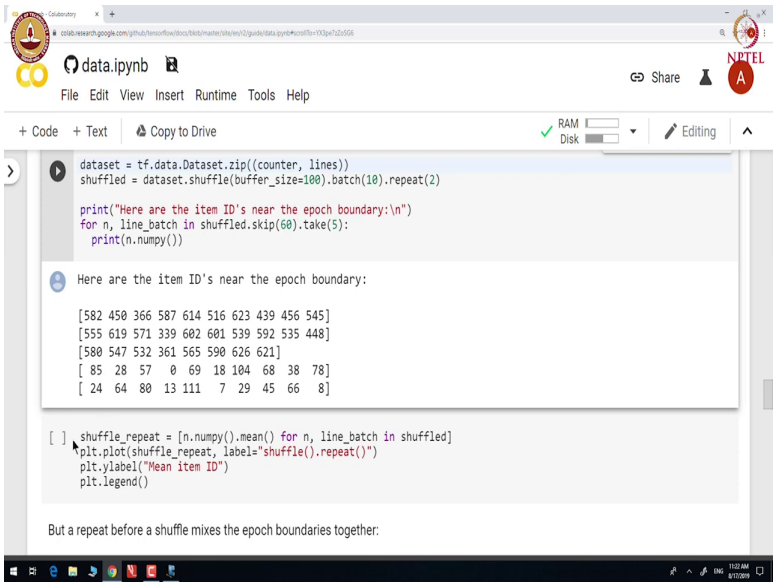
Let us check it out. Yes, you can see that the maximum value of the element in the first batch is 104 as with the batch transformation the order related to repeat maters for batch. The shuffle does not signal the end of the epoch until the shuffle buffer is empty. So, a shuffle placed before a repeat will show every element of one epoch before moving to the next.

(Refer Slide Time: 30:33)



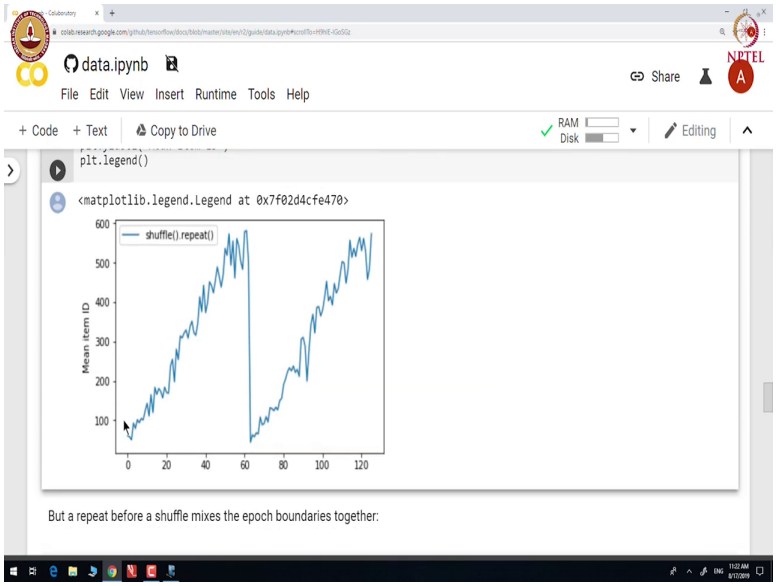
So, here you have placed shuffle before repeat and we can see the elements.

(Refer Slide Time: 30:43)

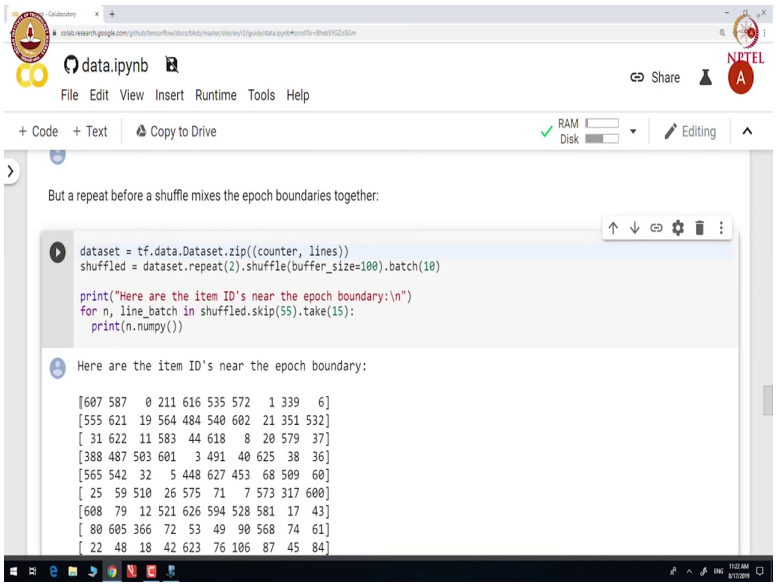


And let us look at the mean id of the element.

(Refer Slide Time: 30:53)

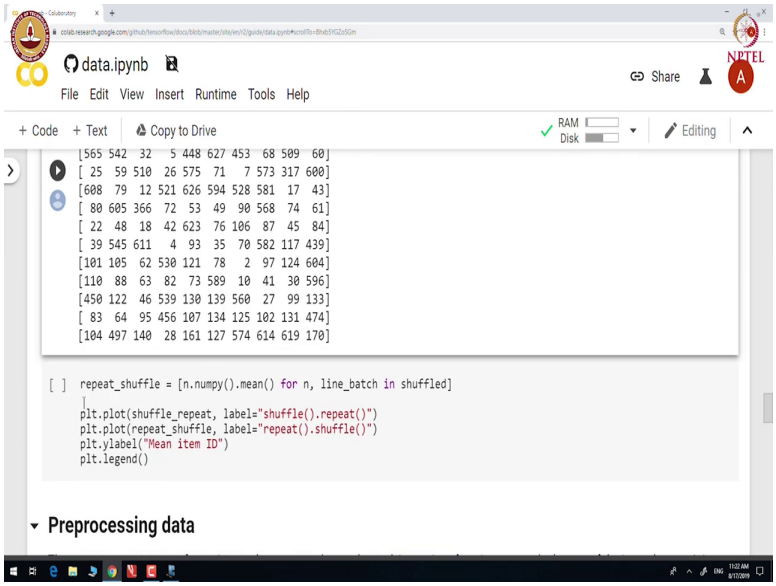


(Refer Slide Time: 30:57)



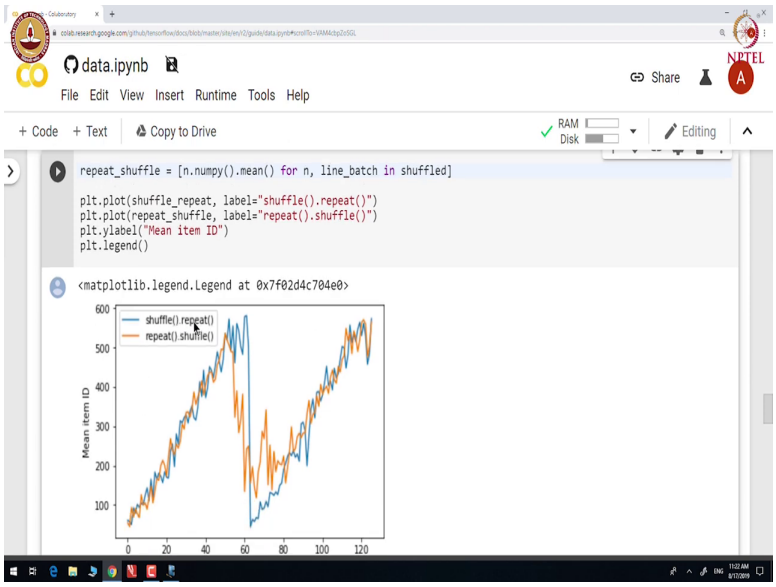
Now, we put shuffle after repeat and see what happens.

(Refer Slide Time: 31:07)



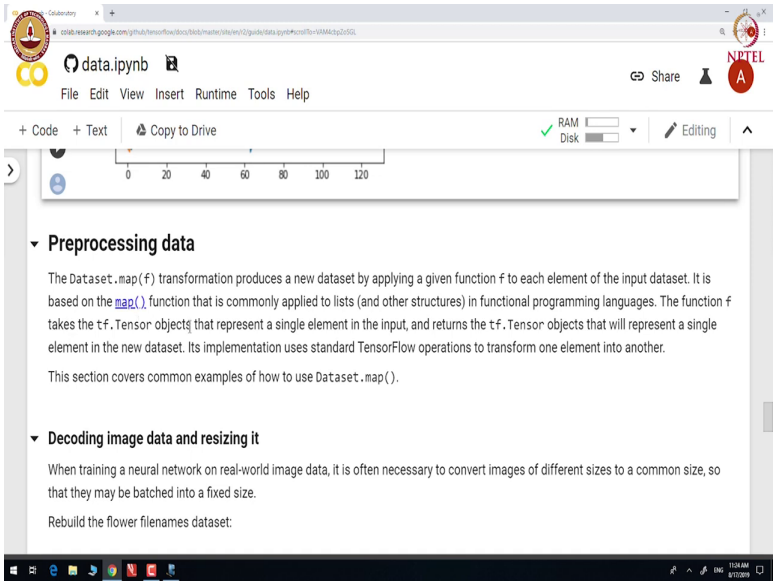
They are the item ids near the epoch boundaries.

(Refer Slide Time: 31:13)



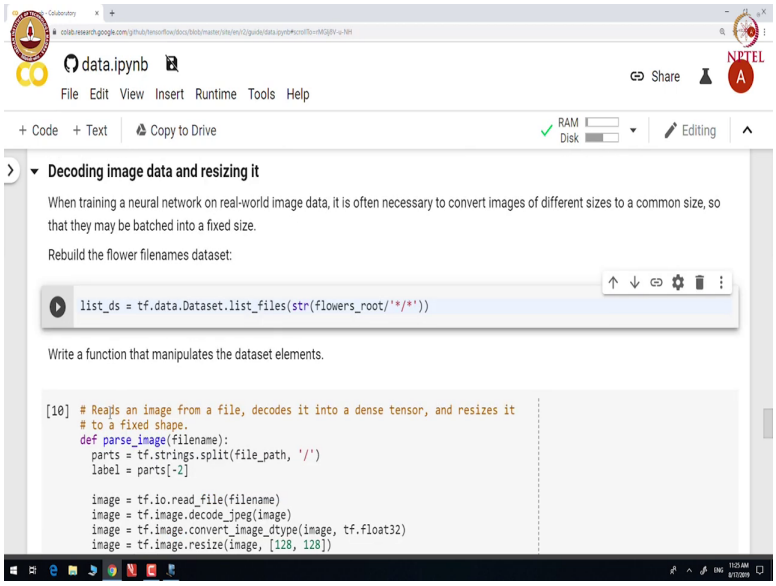
And if you plot two graphs where we compare shuffle before repeat and shuffle after repeat. So, you can see that shuffle before repeat make sure that each element in the epoch is presented to the training data, whereas shuffle after repeat does not give us that kind of guarantee. So, depending on your requirement you can either use shuffle before or after repeat.

(Refer Slide Time: 32:01)



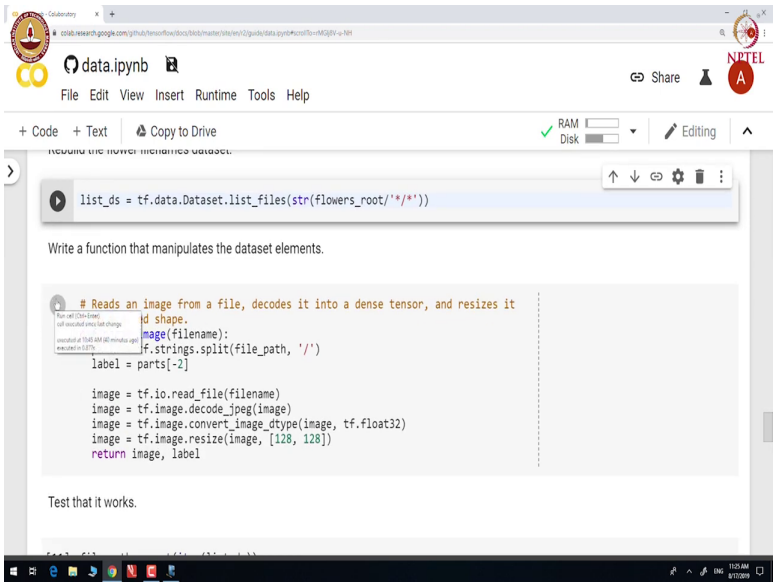
Another important step in data pipeline is pre-processing data, where we want to apply some function on each element of the dataset. So, there are things like normalization or applying some kind of transformation on each element becomes an important part of data pre-processing. So, we use a map transformation for applying a given function or transform each element of the input dataset. We can also use map function to apply arbitrary python function on each of the element of the dataset.

(Refer Slide Time: 33:07)



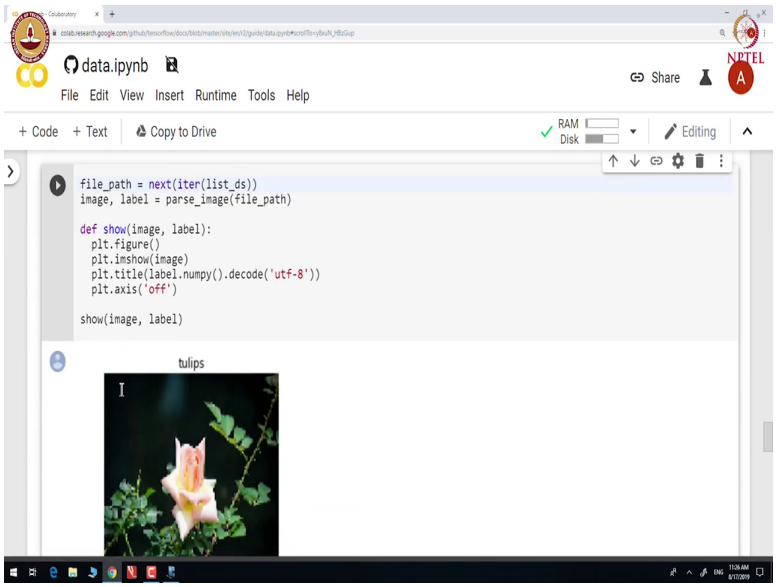
So, let us try to apply map on the image data. To begin with the images are in different sizes, we will convert them into a common size, so that they can be batched into a fixed sizes. Let us look at this transformation in the image data.

(Refer Slide Time: 33:37)



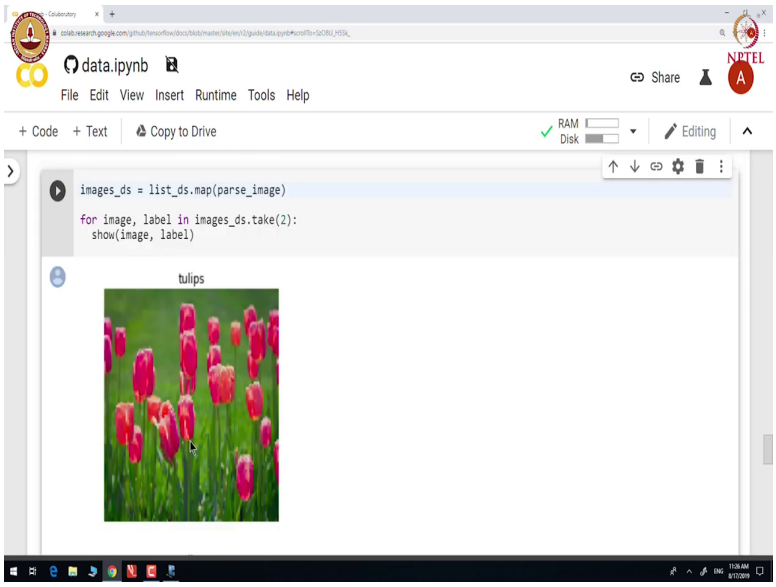
We read images from a file decodes into a dense tensor and resizes it to a fixed shape. So, we first find out the label of the image, we read the image from the file. We decode the jpeg. We convert the image into float and then we resize the image into 128 x 128 tensor.

(Refer Slide Time: 34:09)



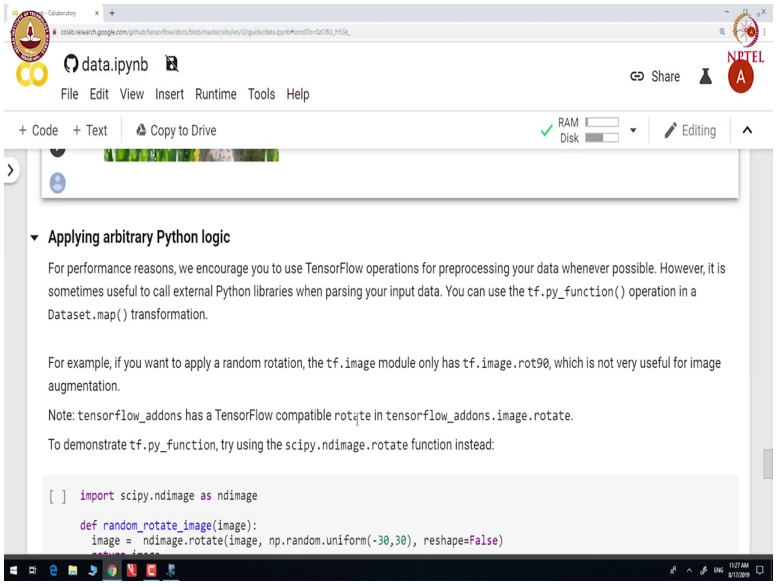
Let us apply it on the first image and see the result. We use *imshow()* command to plot the image. So, this is the first image that is converted into 128 x 128 tensor.

(Refer Slide Time: 34:39)



We apply it on a couple of more images, and you can see that the transformation converts each image into the same size which is 128 x 128.

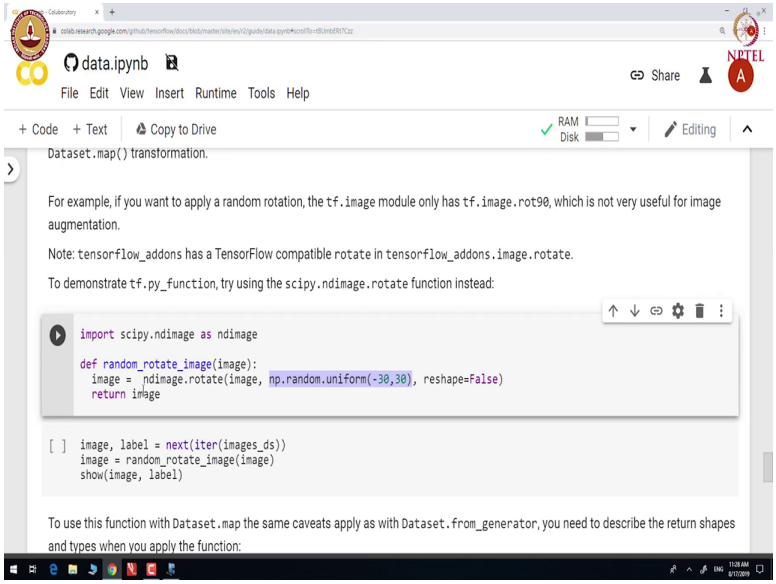
(Refer Slide Time: 34:57)



We can also apply arbitrary python logic using *tf.py_function*. Note that for performance reason, it is better to use TensorFlow operations for preprocessing the data whenever possible, but sometimes it is useful to call external python libraries and this is where

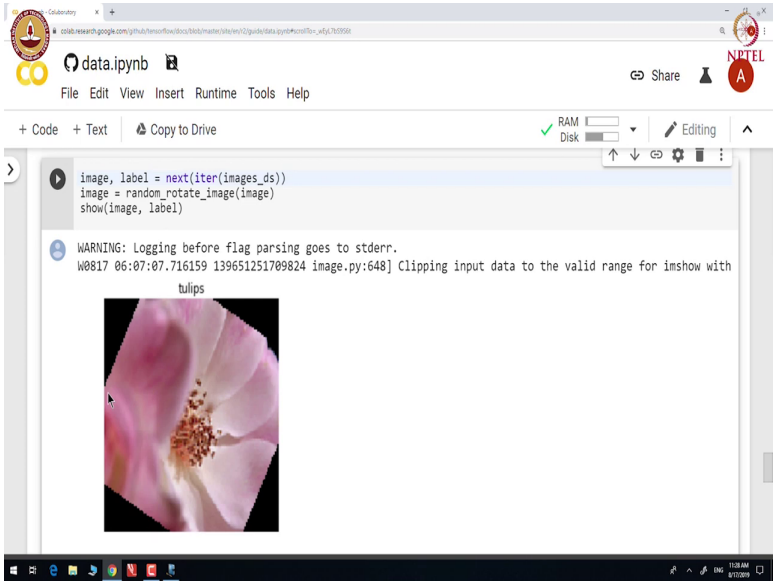
tf.py_function helps us to perform map transformation. Let us look at a concrete example for this. So, here you want the use a rotate function from *scipy* library.

(Refer Slide Time: 35:49)



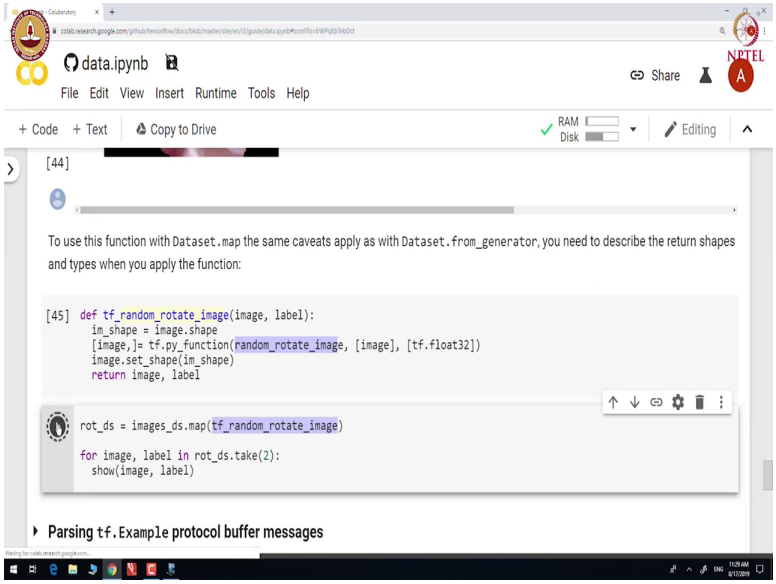
So, first we import the scipy.ndimage library, and we will be using the rotate function. And the image is rotated using the rotate function where we provide the image you provide the angle to rotate which is decided randomly through a uniform distribution any angle between -30 to 30, and it returns the image.

(Refer Slide Time: 36:25)



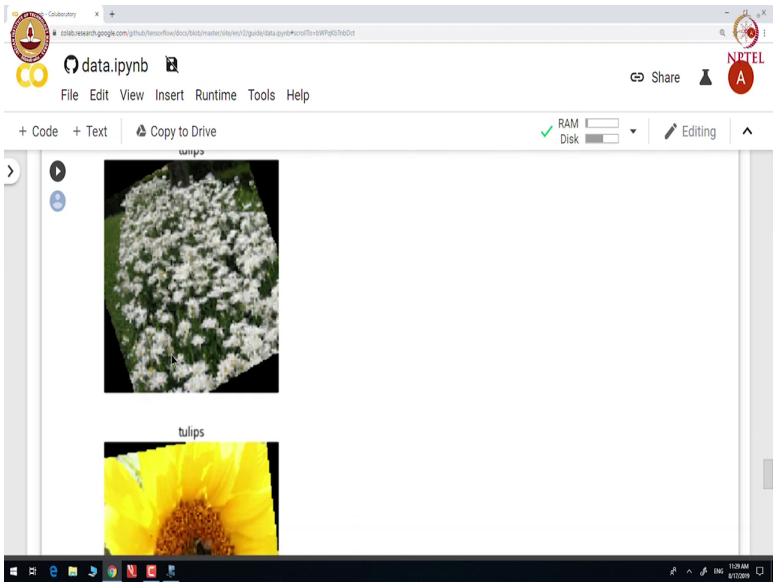
Let us apply it on the first image and see the result. You can see that this particular tulips flower is rotated by and you can see the rotated image of the tulip flower.

(Refer Slide Time: 36:41)



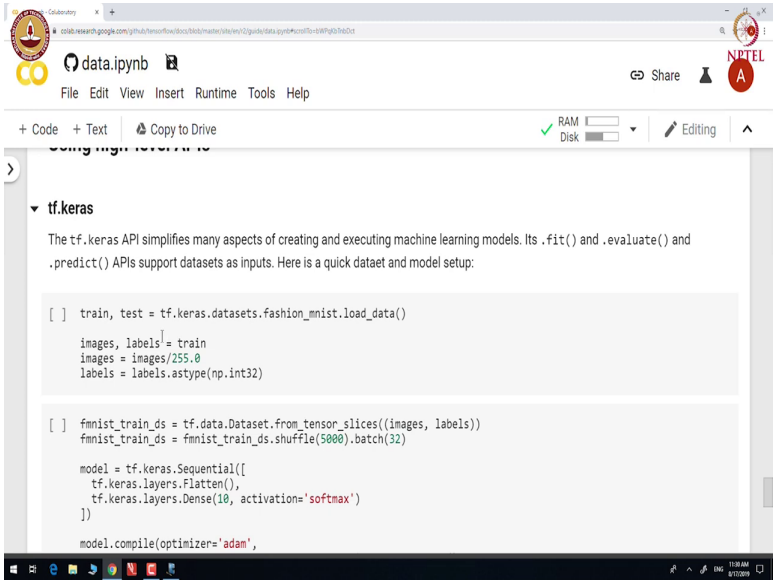
Let us use its function with dataset.map. So, you can see that we used *tf.py_function* to wrap the *random_rotate_image* function. And we call this particular *tf_random_rotate* image function that in turn wraps the rotation function through *py_function*.

(Refer Slide Time: 37:25)



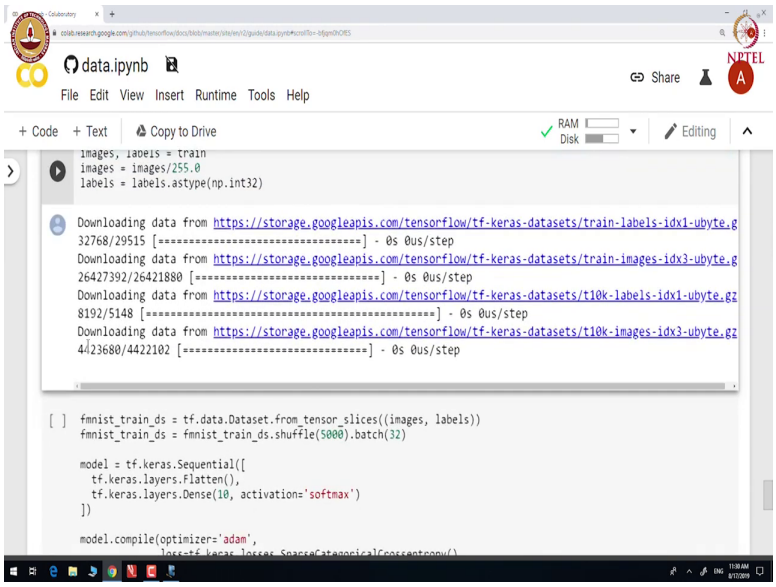
So, you can see that both the images are rotated with different angles, chosen based on a uniform distribution between - 30 to 30.

(Refer Slide Time: 37:47)



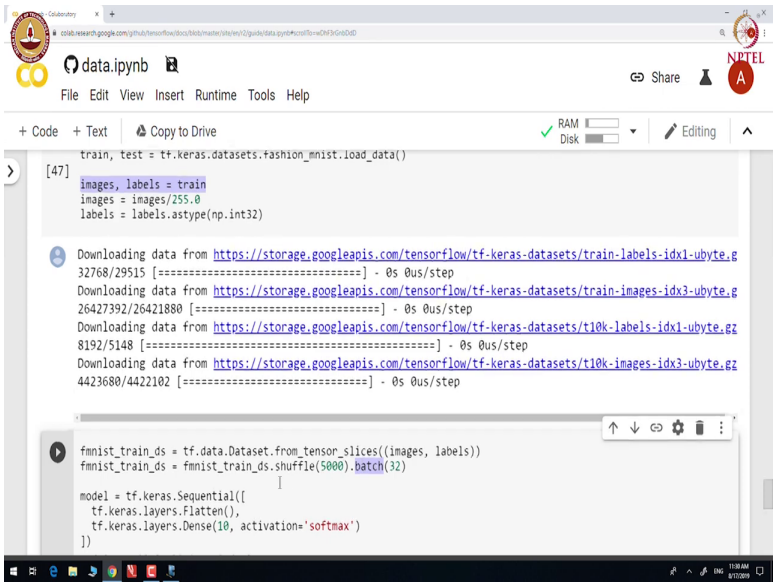
Let us see how to use the data sets with high level APIs. There are two high level APIs. One is *tf.keras* and *tf.estimator*, we will first see how to use the data API with *tf.keras*. So, let us use a Fashion-MNIST dataset.

(Refer Slide Time: 38:15)



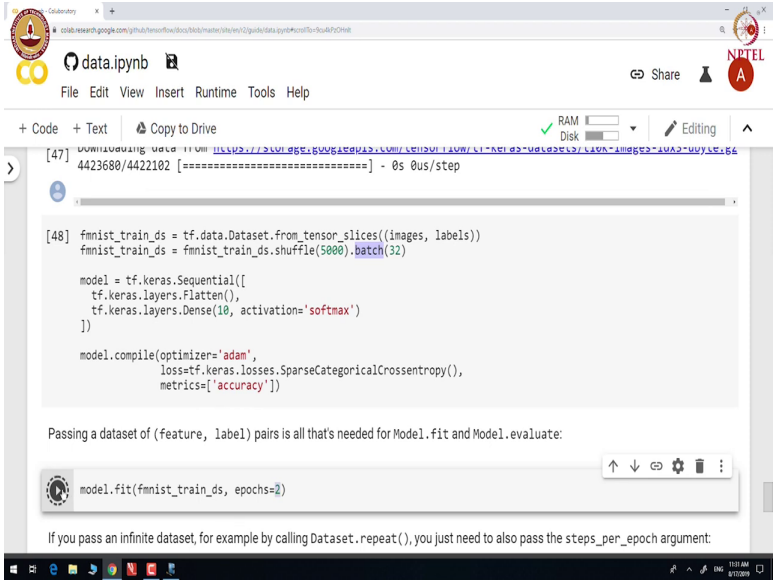
And build a sequential model or a neural network model through *tf.keras.sequential*.

(Refer Slide Time: 38:31)



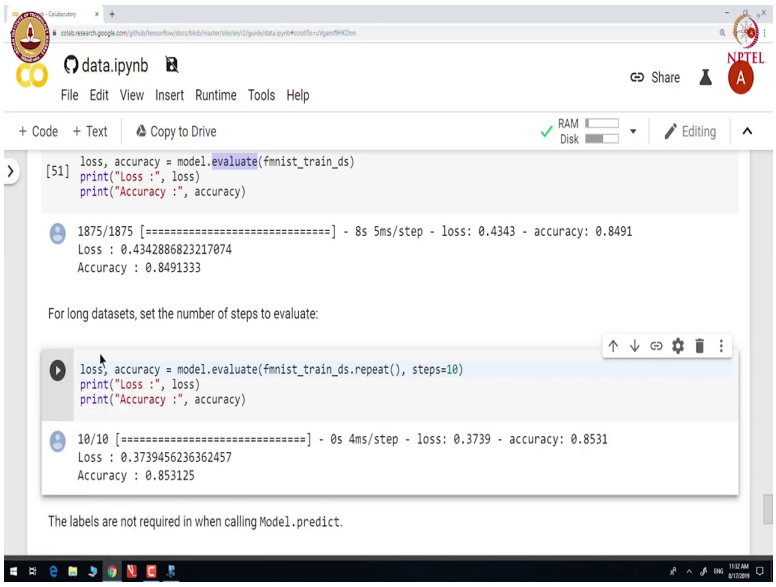
And here will use we construct a dataset object for tensor slices of images and labels which are from the training set. Then we shuffle with buffer size of 5000 and batch it into a batch of 32 examples.

(Refer Slide Time: 39:07)



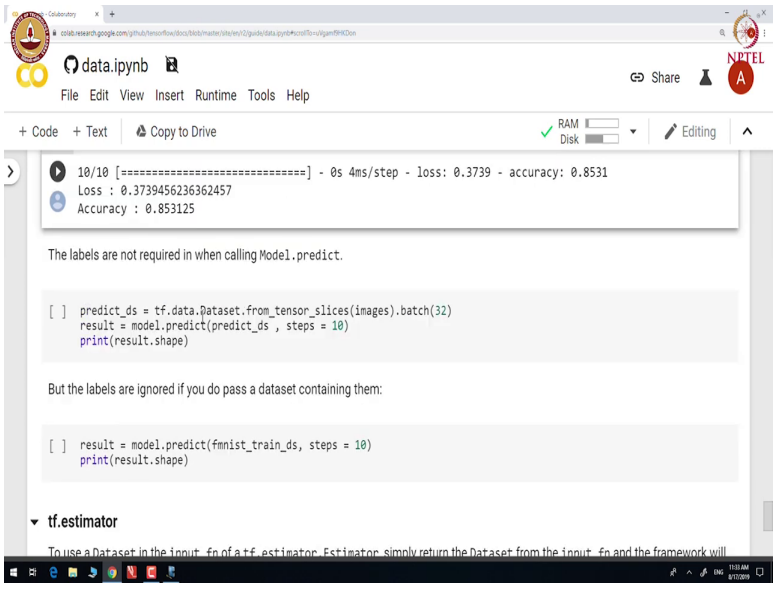
And we pass the training dataset into the fit function and provide epochs for which you want to train the model.

(Refer Slide Time: 39:33)



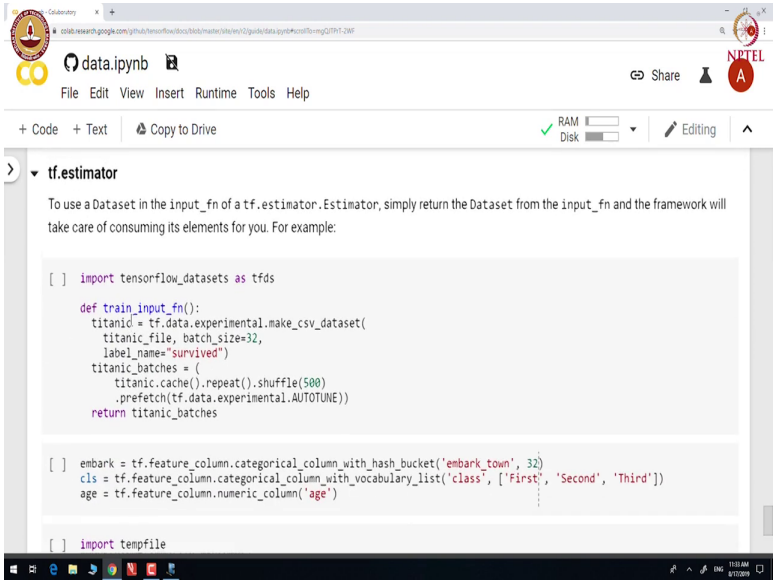
If we pass an infinite dataset by calling the repeat without any arguments, we need to pass steps per epoch along with repeat where we do not specify any argument for repeat. Here if we want to evaluate accuracy on the training set, we pass the training dataset to the evaluate() function. If a dataset is big, we set number of steps to evaluate. Here we said number of steps to 10. So, we get estimation of performance of the model on the training set on a sample.

(Refer Slide Time: 40:35)



So, the labels are not required while calling *model.predict()* even if we pass a dataset containing label, the labels are ignored by the predict() function.

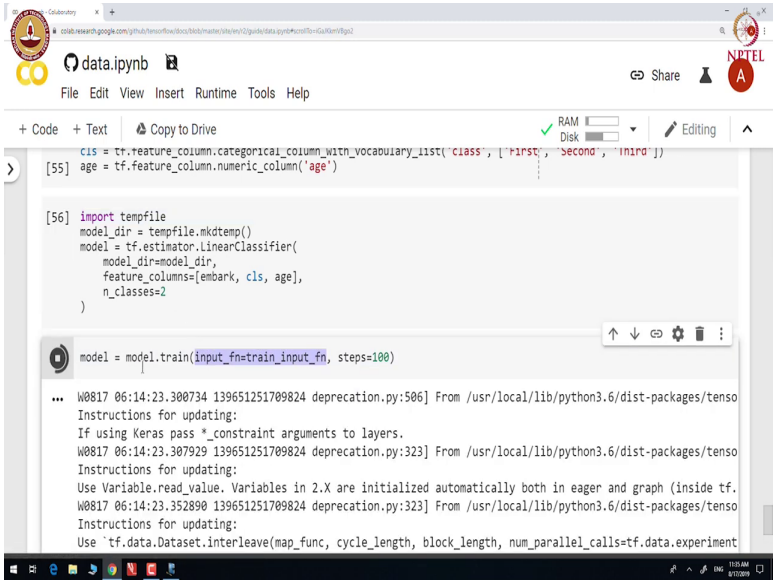
(Refer Slide Time: 40:53)



In case of *tf.estimator*, we need to define *input_fn()* that returns a dataset object, and then the framework will take care of consuming its element for you. So, if you wanted to give titanic dataset as input to *tf.estimator.estimator*, we define *train_input_fn()*. We recreated titanic dataset and perform transformations like repeat and shuffle.

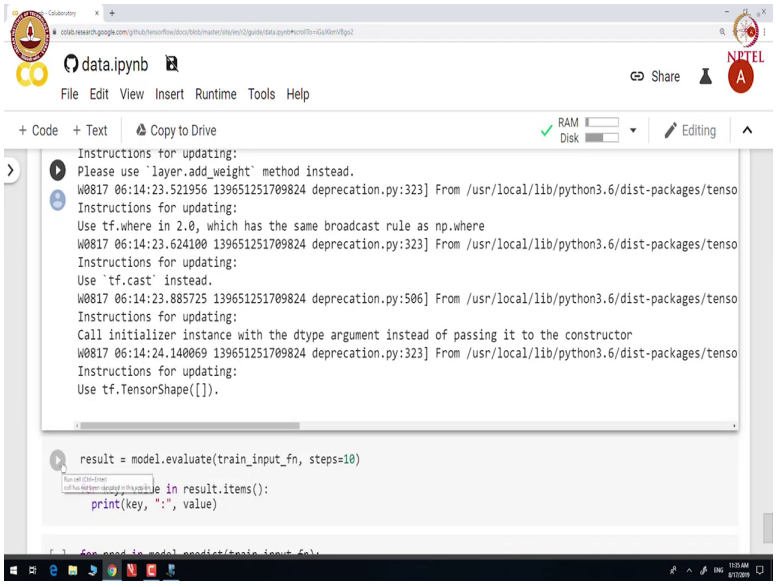
And we also specify the pre-fetch, so that the batch is fetched before time, so that the training is not stalled. In addition to that, you have to convert the non-numerical columns into numerical columns. For example, categorical columns how to be converted into numbers either using hash buckets or a vocabulary list. And then in the *train()* function of the estimator, we specify the *input()* function.

(Refer Slide Time: 42:13)

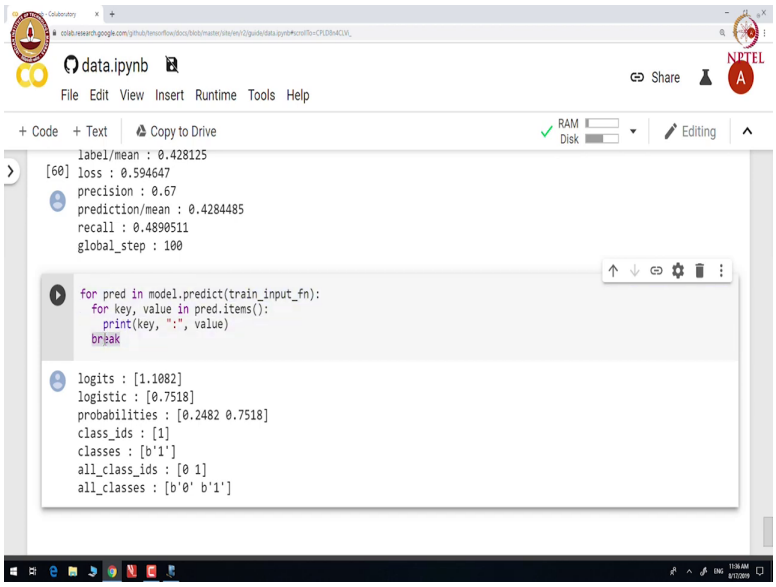


We can also specify the *input()* function in the evaluation in the *evaluate()* function to get the evaluation results.

(Refer Slide Time: 42:27)



(Refer Slide Time: 42:45)



We can get the predictions for each of the example in the training by running a *predict()* function on the estimator by specifying the training *input()* function which consumes one element at a time and gives the predictions for each of the element. Here you print the prediction for the first element after which you break.

So, in this session, we studied how to build input data pipelines with *tf.data* API. In the next session you will learn how to construct *tf.dataset* from different formats like csv, text data and image data.