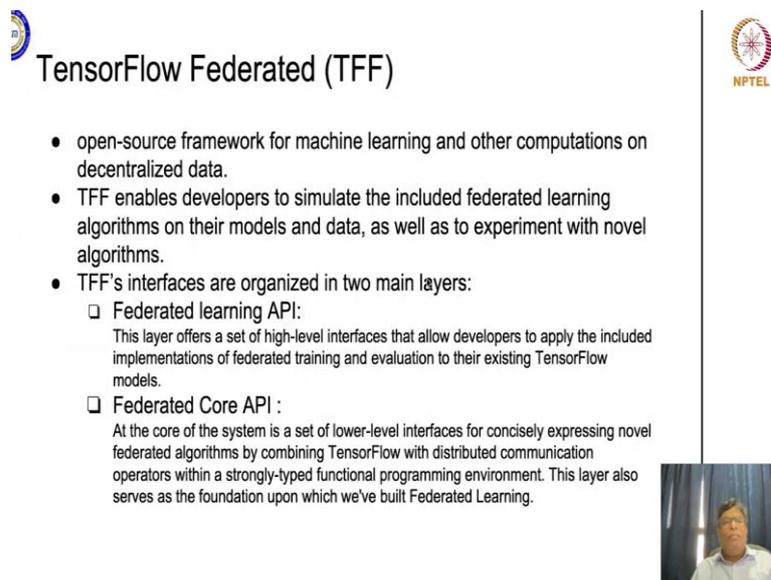


Applied Accelerated Artificial Intelligence
Prof. Satyadhyan Chickerur
School of Computer Science and Engineering
KLE Technological University
Indian Institute of Technology, Palakkad

Lecture - 59
Applied AI: Healthcare (Federated Learning, AI Assisted Annotation)
Session I - Part - 2

(Refer Slide Time: 00:16)



The slide features the TensorFlow Federated (TFF) logo on the left and the NPTEL logo on the right. The main text is centered and describes the framework's purpose and structure. A small video inset of the presenter is located in the bottom right corner of the slide area.

TensorFlow Federated (TFF)

- open-source framework for machine learning and other computations on decentralized data.
- TFF enables developers to simulate the included federated learning algorithms on their models and data, as well as to experiment with novel algorithms.
- TFF's interfaces are organized in two main layers:
 - Federated learning API:
This layer offers a set of high-level interfaces that allow developers to apply the included implementations of federated training and evaluation to their existing TensorFlow models.
 - Federated Core API :
At the core of the system is a set of lower-level interfaces for concisely expressing novel federated algorithms by combining TensorFlow with distributed communication operators within a strongly-typed functional programming environment. This layer also serves as the foundation upon which we've built Federated Learning.

There is something which is called as TensorFlow federated. This is an open source framework for machine learning and other computations on decentralized data. So, you are talking of TensorFlow federated, wherein you are using the concept of federated learning and you are using TensorFlow along with it.

Now, TensorFlow Federated or TFF is going to enable developers to simulate the federated learning algorithms on their models and data as well as to experiment with novel algorithms. And TFF inferences are organized in two main layers ok. So, the idea is that this TensorFlow federated when you talk about their interfaces are organized in two main layers, one is called as the federated learning API and another one is called as federated core API.

Now, this basically means these application programming interfaces of TFF gives you two levels of sophistication. This means the federated learning API is going to give you a

set of high-level interfaces that allow the developers to apply the implementation of federated learning which basically means you can do federated training and evaluation to any of the existing TensorFlow models.

Whereas federated core API is the low-level interface for expressing these federated algorithms by mixing TensorFlow with distributed communication operators. So, see try understanding that we have told the difference between distributed deep learning and federated deep learning to some extent.

Now, in case of distributed deep learning you are trying to distribute either your data or model, in both these cases your data is actually with somebody else. In this scenario what you want is, you do not want the data to go to somebody else, but the model one strain could be used by other people right.

So, this is what is the difference between the distributed deep learning and federated deep learning and since federated deep learning also involves distribution. So, there has to be some communication operation which should happen ok.

And that communication operation between the various nodes ok has to be as a API and it basically would be something called as federated core API ok. So, today we will do hands on TensorFlow federated and we will try to see four examples of it on Google Colab and tomorrow we will actually go to NVIDIA FLARE which I told.

(Refer Slide Time: 03:46)



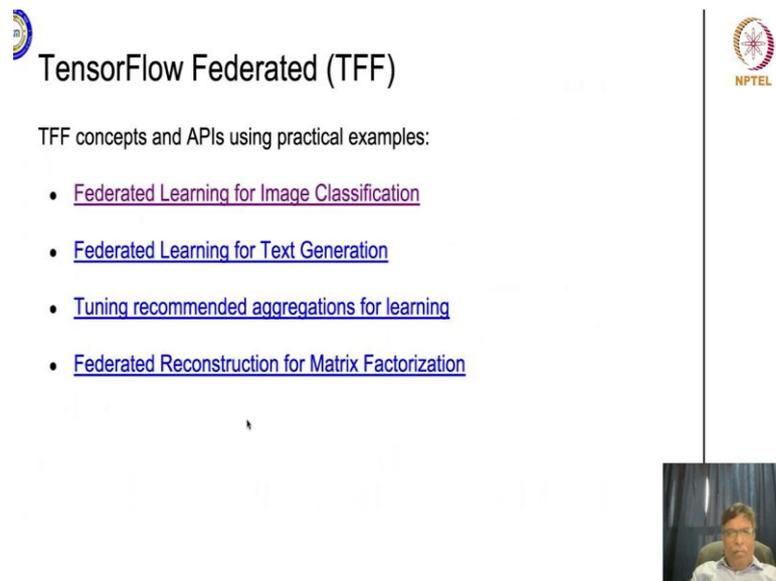
NVIDIA FLARE :

- NVIDIA FLARE™ (NVIDIA Federated Learning Application Runtime Environment) is a domain-agnostic, open-source, and extensible SDK for Federated Learning.
- It allows researchers and data scientists to adapt existing ML/DL workflow to a federated paradigm and enables platform developers to build a secure, privacy-preserving offering for a distributed multi-party collaboration.



I thought that we can discuss what NVIDIA FLARE is, but tomorrow itself when we discuss this implementation that time if I take it that would be more better right. Otherwise, I was thinking of covering this theory also today, but since we have got some demos which you can do hands on on.

(Refer Slide Time: 04:11)



 TensorFlow Federated (TFF)

TFF concepts and APIs using practical examples:

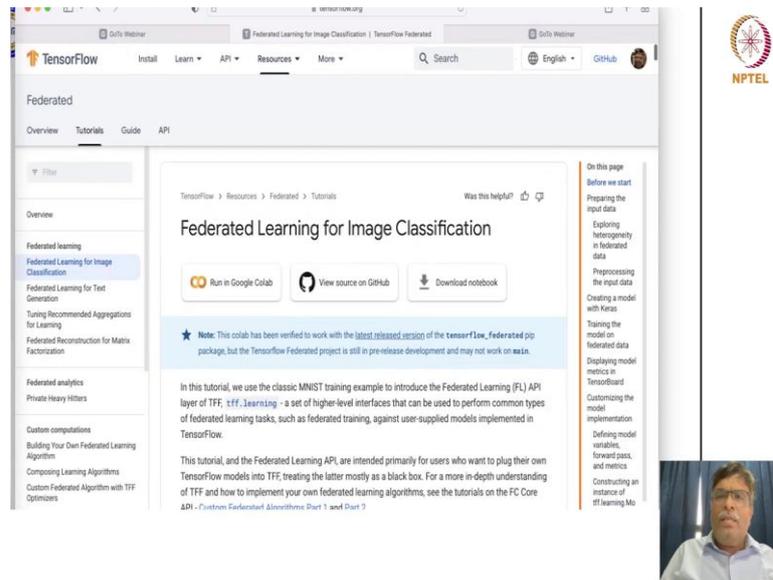
- [Federated Learning for Image Classification](#)
- [Federated Learning for Text Generation](#)
- [Tuning recommended aggregations for learning](#)
- [Federated Reconstruction for Matrix Factorization](#)





So, I will just go to the hands on portion for today and we will try to understand that how TensorFlow federated ok could be used for image classification, text generation ok and federated reconstruction for something called as matrix factorization. So, based on the time ok I will start with image classification and then to text generation. If the time permits I will try running through these two also, otherwise this link would be shared with you so that you can do some hands on on your own right. So, let us try to go to federated learning for image classification.

(Refer Slide Time: 04:54)



TensorFlow Federated

Federated Learning for Image Classification

Run in Google Colab | View source on GitHub | Download notebook

Note: This colab has been verified to work with the latest released version of the tensorflow_federated pip package, but the TensorFlow Federated project is still in pre-release development and may not work on main.

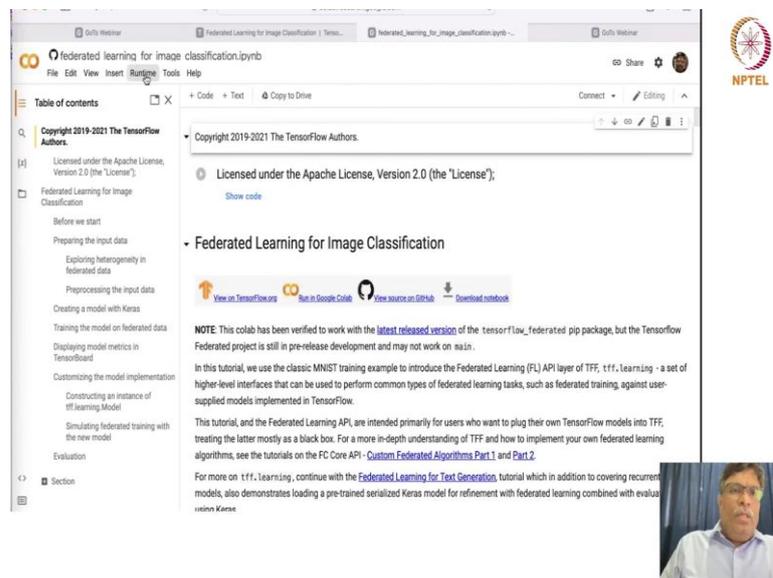
In this tutorial, we use the classic MNIST training example to introduce the Federated Learning (FL) API layer of TFF, `tff.learning` - a set of higher-level interfaces that can be used to perform common types of federated learning tasks, such as federated training, against user-supplied models implemented in TensorFlow.

This tutorial, and the Federated Learning API, are intended primarily for users who want to plug their own TensorFlow models into TFF, treating the latter mostly as a black box. For a more in-depth understanding of TFF and how to implement your own federated learning algorithms, see the tutorials on the FC Core API - [Custom Federated Algorithms Part 1](#) and [Part 2](#).

On this page:
Before we start
Preparing the input data
Exploring heterogeneity in federated data
Preprocessing the input data
Creating a model with Keras
Training the model on federated data
Displaying model metrics in TensorBoard
Customizing the model implementation
Defining model variables, forward pass, and metrics
Constructing an instance of `tf.learning.Model`

So, so let us try to understand the context and then let us try to do this ah.

(Refer Slide Time: 05:18)



federated_learning_for_image_classification.ipynb

Federated Learning for Image Classification

View on TensorFlow.org | Run in Google Colab | View source on GitHub | Download notebook

NOTE: This colab has been verified to work with the latest released version of the tensorflow_federated pip package, but the TensorFlow Federated project is still in pre-release development and may not work on main.

In this tutorial, we use the classic MNIST training example to introduce the Federated Learning (FL) API layer of TFF, `tff.learning` - a set of higher-level interfaces that can be used to perform common types of federated learning tasks, such as federated training, against user-supplied models implemented in TensorFlow.

This tutorial, and the Federated Learning API, are intended primarily for users who want to plug their own TensorFlow models into TFF, treating the latter mostly as a black box. For a more in-depth understanding of TFF and how to implement your own federated learning algorithms, see the tutorials on the FC Core API - [Custom Federated Algorithms Part 1](#) and [Part 2](#).

For more on `tff.learning`, continue with the [Federated Learning for Text Generation](#) tutorial which in addition to covering recurrent models, also demonstrates loading a pre-trained serialized Keras model for refinement with federated learning combined with evaluation on Keras.

(Refer Slide Time: 05:20)

The screenshot shows a Google Colab notebook interface. The title bar reads 'federated_learning_for_image_classification.ipynb'. The left sidebar contains a 'Table of contents' with sections like 'Before we start', 'Preparing the input data', 'Exploring heterogeneity in federated data', 'Preprocessing the input data', 'Creating a model with Keras', 'Training the model on federated data', 'Displaying model metrics in TensorBoard', 'Customizing the model implementation', 'Constructing an instance of tff.learning.Model', 'Simulating federated training with the new model', and 'Evaluation'. The main content area shows the beginning of the notebook text, including a copyright notice for TensorFlow Authors and a license notice: 'Licensed under the Apache License, Version 2.0 (the "License");'. A small video inset of a man is visible in the bottom right corner.

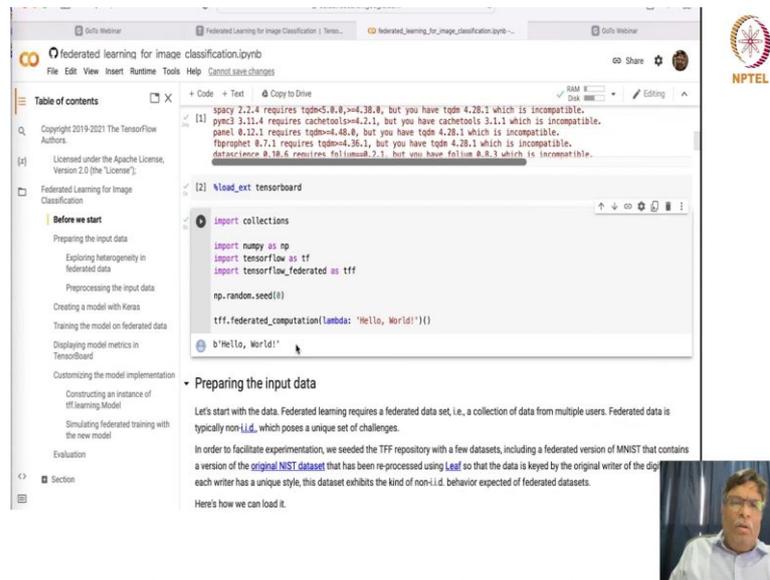
So, let me just change the run time.

(Refer Slide Time: 05:37)

This screenshot shows the same Google Colab notebook as before, but with a 'Notebook settings' dialog box open in the center. The dialog has a title bar 'Notebook settings' and contains the following options: 'Hardware accelerator' is set to 'GPU'; 'Background execution' is checked; and there is a checkbox for 'Omit code cell output when saving this notebook' which is currently unchecked. There are 'Cancel' and 'Save' buttons at the bottom of the dialog. A small video inset of a man is visible in the bottom right corner.

So, here I will do this, do this and then I save this and then let us try to understand this.

(Refer Slide Time: 07:03)



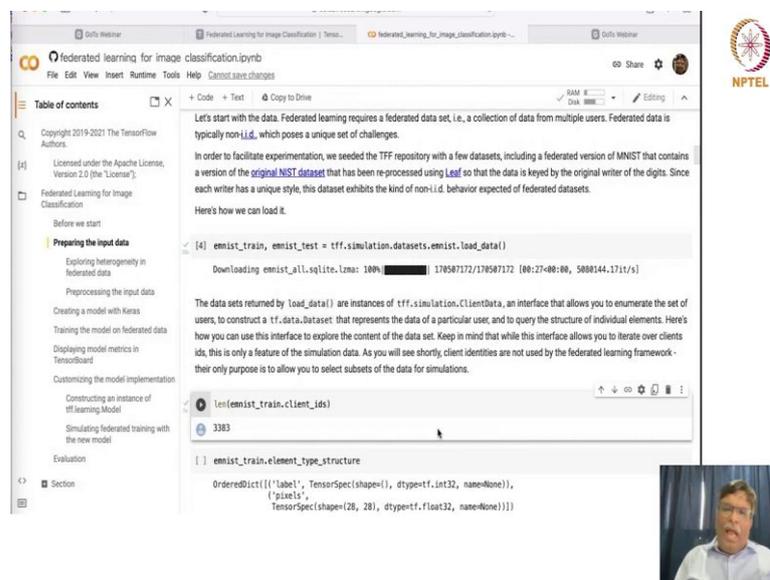
The screenshot shows a Jupyter Notebook interface. The code cell contains the following Python code:

```
import collections
import numpy as np
import tensorflow as tf
import tensorflow_federated as tff
np.random.seed()
tff.federated_computation(lambda: 'Hello, World!')()
```

The output of the cell is 'b'Hello, World!'. The notebook also displays a table of contents on the left and a section titled 'Preparing the input data' with explanatory text about federated data sets.

Then you are trying to import in various things like tensorflow federated as tff and this computation of trying to just do some 'Hello World' type of stuff. So, that you know everything is fine. So, this we have got it like this.

(Refer Slide Time: 07:18)



The screenshot shows a Jupyter Notebook interface. The code cell contains the following Python code:

```
emnist_train, mnist_test = tff.simulation.datasets.emnist.load_data()
len(emnist_train.client_ids)
```

The output of the cell is 3383. The notebook also displays a table of contents on the left and a section titled 'Preparing the input data' with explanatory text about federated data sets.

So, now let us try to start with the data right. So, you are supposed to be working with something called as federated data. So, the data has to be you know split and it has to be shared. So, we are trying to work on a Colab wherein you should have the feel of trying to work with multiple users right. So, the federated data ok is typically a non-IID right.

So, when you talk of non-IID it is Independent Identical Distributed type of a random variable basically ok or random data.

So, this is in terms of probabilities when you talk of IID type of data set right. So, now, we they have actually kept some TFF repository with few data sets and we are talking of federated version of MNIST ok. That contains original MNIST data set and they have reprocessed it using this leaf so that the data is keyed by the original writer of the digits.

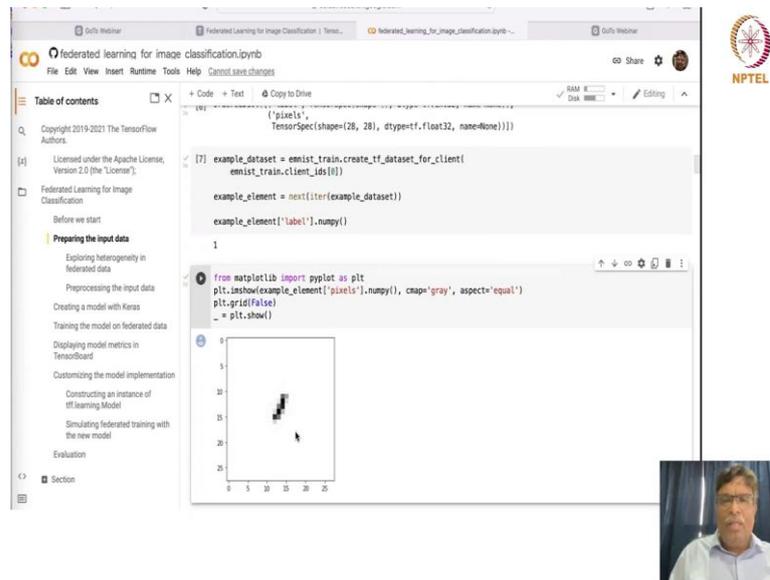
And since each writer has a unique style, the data set exhibits the kind of non-IID behavior expected of federated data sets. See the idea of having a non-IID type of a behavior is that it should be not too much of independent identical distributed random variables, but it should be like somewhat similar, right. You cannot have totally very non IID type of behavior.

So, what you can do is, once we go here we can actually load the tensorflow federated data set which is mnist here ok. So, it is it will take some time does not matter, but yeah. So, now, ok by the time its loading just let me tell you like how the simulation is done. So, here you have got something called as client data right. So, this client data basically is for a particular user ok some portion of this data set ok.

And to query the structure of individual elements you use this function ok. So, the idea is that. So, client identities are not used by federated learning framework, the only purpose is to allow you to select subsets of the data for simulation. So, please try understanding that once you are able to know the client's identity right then you can do many things right.

But here the idea is the interface which is being developed or used ok is going to actually iterate over the client ids ok. And you are trying to just get some information about the feature of the simulation data right. So, that is what you are going to see here right.

(Refer Slide Time: 10:28)



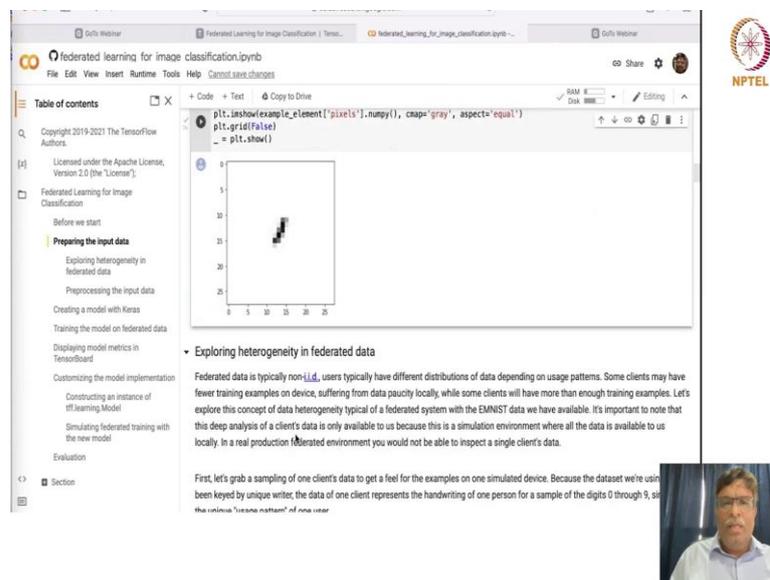
The screenshot shows a Jupyter Notebook interface with the following content:

- Table of contents:** A sidebar on the left lists sections like 'Preparing the input data', 'Exploring heterogeneity in federated data', etc.
- Code cell:** Contains Python code to create a dataset and iterate over it. The code is:

```
('pixels',  
    TensorSpec(shape=(28, 28), dtype=tf.float32, name=None)))  
[7] example_dataset = emnist_train.create_tf_dataset_for_client(  
    emnist_train.client_ids[8])  
  
example_element = next(iter(example_dataset))  
  
example_element['label'].numpy()  
  
1  
  
from matplotlib import pyplot as plt  
plt.imshow(example_element['pixels'].numpy(), cmap='gray', aspect='equal')  
plt.grid(False)  
_ = plt.show()
```
- Figure:** A 28x28 grayscale plot showing a handwritten digit '1' on a grid.
- NPTEL logo:** Located in the top right corner.
- Video feed:** A small inset video of a man speaking in the bottom right corner.

So, this is your data set. So, let us try to understand, let us try to see example ok for number 1 this is a data which is basically found out right.

(Refer Slide Time: 10:48)



The screenshot shows a Jupyter Notebook interface with the following content:

- Table of contents:** Similar to the previous slide, but with 'Exploring heterogeneity in federated data' expanded.
- Code cell:** Contains Python code to plot a handwritten digit '1':

```
plt.imshow(example_element['pixels'].numpy(), cmap='gray', aspect='equal')  
_ = plt.show()
```
- Text:** A section titled 'Exploring heterogeneity in federated data' with the following text:

Federated data is typically non-IID, users typically have different distributions of data depending on usage patterns. Some clients may have fewer training examples on device, suffering from data paucity locally, while some clients will have more than enough training examples. Let's explore this concept of data heterogeneity typical of a federated system with the EMNIST data we have available. It's important to note that this deep analysis of a client's data is only available to us because this is a simulation environment where all the data is available to us locally. In a real production federated environment you would not be able to inspect a single client's data.

First, let's grab a sampling of one client's data to get a feel for the examples on one simulated device. Because the dataset we're using has been keyed by unique writer, the data of one client represents the handwriting of one person for a sample of the digits 0 through 9, so the following "instance numbers" are from one user.
- Figure:** A 28x28 grayscale plot showing a handwritten digit '1' on a grid.
- NPTEL logo:** Located in the top right corner.
- Video feed:** A small inset video of a man speaking in the bottom right corner.

So, this is just like how federated data set is right. So, federated data is typically non-IID; that means, users typically have different distributions of data depending on the usage pattern. This is very random right probabilistic. So, the idea is that you will have clients, who will have fewer training examples on device; you will have data paucity locally while some clients will have more than enough training examples.

But these when you say that there are only fewer training examples, those training examples would be really very nice ok maybe in case which would going to improve your performance sometimes very better ok, this may happen. So, data heterogeneity is required for improving your system. So, you get that.

(Refer Slide Time: 11:37)

NPTEL

So, let us try to understand that for one client ok. This is how the spread of data is, this is just the example ok. So, you will have so many 1's ok, so many 0's and then you have 5 written in these fashion, 7 in this so like this.

(Refer Slide Time: 11:55)

NPTEL

So, for each client right you will have got some different types of frequencies right, for each of these digits ok.

(Refer Slide Time: 12:13)

The screenshot displays a Jupyter Notebook interface. On the left, a table of contents lists sections such as 'Exploring heterogeneity in federated data', 'Preprocessing the input data', 'Creating a model with Keras', 'Training the model on federated data', 'Displaying model metrics in TensorBoard', 'Customizing the model implementation', 'Constructing an instance of fflLearning Model', 'Simulating federated training with the new model', and 'Evaluation'. The main area shows a code editor with the following Python code:

```
[ ] # Each client has different mean images, meaning each client will be nudging  
# the model in their own directions locally.  
  
for i in range(5):  
    client_dataset = emnist_train.create_tf_dataset_for_client(  
        emnist_train.client_ids[i])  
    plot_data = collections.defaultdict(list)  
    for example in client_dataset:  
        plot_data[example['label']].append(example['pixels'].numpy())  
    f = plt.figure(1, figsize=(12, 5))  
    f.suptitle("Client #1's Mean Image Per Label".format(i))
```

Below the code, three bar charts are displayed, each showing the frequency of digits 0 through 9 for a specific client. The y-axis for each chart ranges from 0 to 12. The bars are color-coded by digit, and the distribution of frequencies varies across the three clients, illustrating heterogeneity in the data.

So, now here you will get client 0, client 1, client 2, client 3, client 4 and up to the client 5. So, each of these clients will have different numbers ok for 0, 1, 2, 3 up till 9 digits ok. So, this basically (Refer Time: 13:06) approximately 11 values ok for digit 9, this client 0 has got about 8 values or 8 types of digit 9 right something like this.

So, now, once we get this, we can visualize the mean image per client for each MNIST label right. So; that means, that we can find out the mean of each pixel values for all of the user's examples for one label right.

(Refer Slide Time: 13:38)

The screenshot shows a Jupyter Notebook titled "federated_learning_for_image_classification.ipynb". The code cell contains the following Python code:

```
[ ]: f = plt.figure(1, figsize=(10, 5))
      f.suptitle("Client #1's Mean Image Per Label", format(1))
      for j in range(10):
          mean_img = np.mean(plot_data[j], 0)
          plt.subplot(2, 5, j+1)
          plt.imshow(mean_img.reshape((28, 28)))
          plt.axis('off')
```

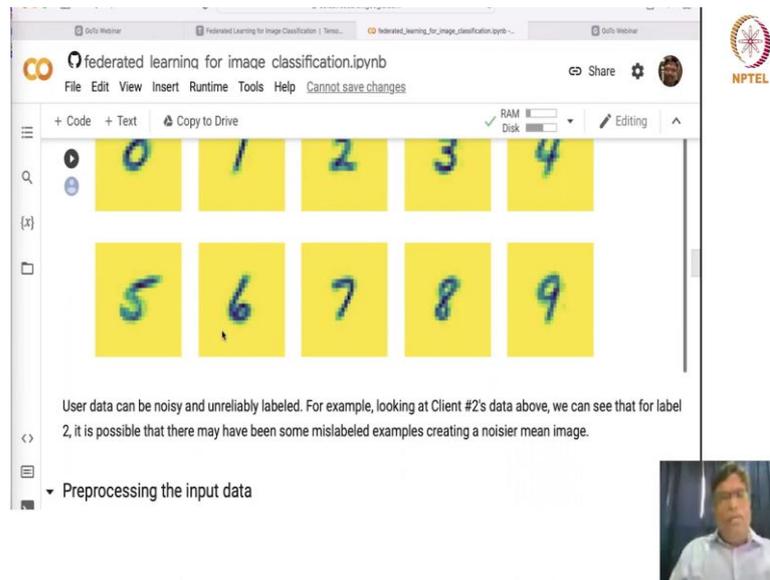
The output of the code is a 3x5 grid of mean images for labels 0-9. The first row shows labels 0-4, the second row shows labels 5-9, and the third row shows labels 0-4. The images are yellow with black digits. The caption below the grid reads "Client #1's Mean Image Per Label".

In the bottom right corner, there is a small video inset showing a man with glasses and a light blue shirt speaking.

So, each client is going to have different mean images, meaning each clients will be nudging the model in their own directions locally. Because when your mean image changes right and when you are talking of something like a model and assume that you are using CNN.

So, the mean image and the actual image is which is going to tell you the direction right of I suppose I need to zoom a bit if possible ok. So, let me close this so that you know, yeah. So see here that when you are talking of client 0's mean image per label this is how it is, client 1's this is how it is client 2 this is how it is. So, if you see this these are all not same mean images right ok.

(Refer Slide Time: 14:34)



federated learning for image classification.ipynb

File Edit View Insert Runtime Tools Help Cannot save changes

+ Code + Text Copy to Drive RAM Disk Editing

0 1 2 3 4

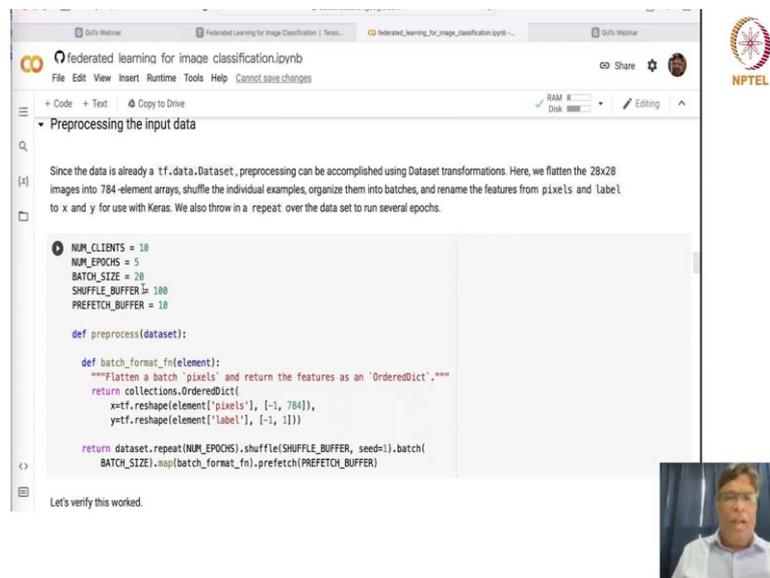
5 6 7 8 9

User data can be noisy and unreliable. For example, looking at Client #2's data above, we can see that for label 2, it is possible that there may have been some mislabeled examples creating a noisier mean image.

Preprocessing the input data

So, this basically means your data will be noisy, it will be unreliably labelled ok.

(Refer Slide Time: 14:38)



federated learning for image classification.ipynb

File Edit View Insert Runtime Tools Help Cannot save changes

+ Code + Text Copy to Drive RAM Disk Editing

Preprocessing the input data

Since the data is already a `tf.data.Dataset`, preprocessing can be accomplished using Dataset transformations. Here, we flatten the 28x28 images into 784-element arrays, shuffle the individual examples, organize them into batches, and rename the features from `pixels` and `label` to `x` and `y` for use with Keras. We also throw in a `repeat` over the data set to run several epochs.

```
NUM_CLIENTS = 10
NUM_EPOCHS = 5
BATCH_SIZE = 28
SHUFFLE_BUFFER = 100
PREFETCH_BUFFER = 10

def preprocess(dataset):

    def batch_format_fn(element):
        """Flatten a batch 'pixels' and return the features as an 'OrderedDict'."""
        return collections.OrderedDict(
            x=tf.reshape(element['pixels'], [-1, 784]),
            y=tf.reshape(element['label'], [-1, 1]))

    return dataset.repeat(NUM_EPOCHS).shuffle(SHUFFLE_BUFFER, seed=1).batch(
        BATCH_SIZE).map(batch_format_fn).prefetch(PREFETCH_BUFFER)
```

Let's verify this worked.

ah And then the mean image right is going to be affected. So, this is how it is going to happen ok.

(Refer Slide Time: 14:57)

The screenshot shows a Jupyter Notebook titled "federated learning for image classification.ipynb". The main content area displays a grid of handwritten digits (5, 6, 7, 8, 9) and a caption "Client #1's Mean Image Per Label" with a corresponding grid of digits (0, 1, 2, 3, 4). Below the grid, there is a text box that reads: "User data can be noisy and unreliably labeled. For example, looking at Client #2's data above, we can see that for label 2, it is possible that there may have been some mislabeled examples creating a noisier mean image." A small video inset in the bottom right corner shows a man speaking.

So, now so the idea is that you need to pre process that image. So, let us say that you have got 10 number of clients and your epoch size is 5, batch size is 20 you have some shuffle buffer right and all of that. So, let us try to understand that you can pre-process ok for data set transformation ok. So, that you do and then once you have pre-processed it ok.

(Refer Slide Time: 15:28)

The screenshot shows a Jupyter Notebook with the following code and output:

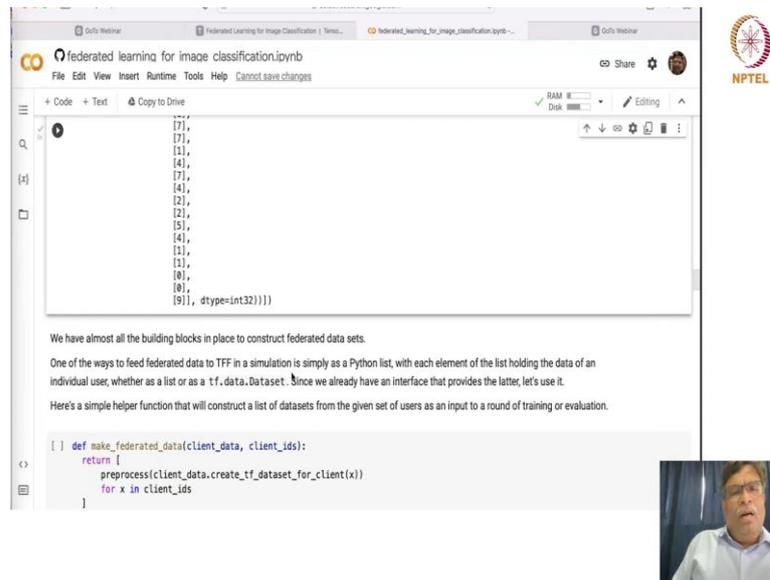
```
preprocessed_example_dataset = preprocess(example_dataset)
sample_batch = tf.nest.map_structure(lambda x: x.numpy(),
next(iter(preprocessed_example_dataset)))
sample_batch
```

OrderedDict([('x', array([[1., 1., 1., ..., 1., 1., 1.],
[1., 1., 1., ..., 1., 1., 1.],
[1., 1., 1., ..., 1., 1., 1.],
...,
[1., 1., 1., ..., 1., 1., 1.],
[1., 1., 1., ..., 1., 1., 1.],
[1., 1., 1., ..., 1., 1., 1.],
[1., 1., 1., ..., 1., 1., 1.], dtype=float32)],
('y', array([[2],
[5],
[7],
[7],
[7],
[7],
[1],
[4],
[7],
[4],
[2]]))

A small video inset in the bottom right corner shows a man speaking.

So, you basically are in a position, ok.

(Refer Slide Time: 15:37)



federated learning for image classification.ipynb

```
[[7],  
 [7],  
 [1],  
 [4],  
 [7],  
 [4],  
 [2],  
 [2],  
 [2],  
 [5],  
 [4],  
 [1],  
 [1],  
 [0],  
 [0],  
 [9]], dtype=int32]]
```

We have almost all the building blocks in place to construct federated data sets.

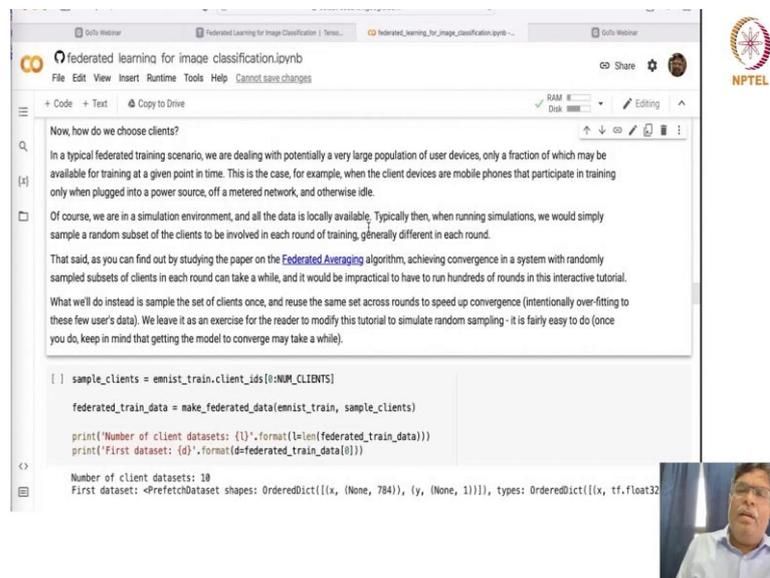
One of the ways to feed federated data to TFF in a simulation is simply as a Python list, with each element of the list holding the data of an individual user, whether as a list or as a `tf.data.Dataset`. Since we already have an interface that provides the latter, let's use it.

Here's a simple helper function that will construct a list of datasets from the given set of users as an input to a round of training or evaluation.

```
[ ] def make_federated_data(client_data, client_ids):  
    return [  
        preprocess(client_data.create_tf_dataset_for_client(x)  
                    for x in client_ids  
    ]
```

To get a initiation of how do you construct the federated data sets, right.

(Refer Slide Time: 15:54)



federated learning for image classification.ipynb

Now, how do we choose clients?

In a typical federated training scenario, we are dealing with potentially a very large population of user devices, only a fraction of which may be available for training at a given point in time. This is the case, for example, when the client devices are mobile phones that participate in training only when plugged into a power source, off a metered network, and otherwise idle.

Of course, we are in a simulation environment, and all the data is locally available. Typically then, when running simulations, we would simply sample a random subset of the clients to be involved in each round of training, generally different in each round.

That said, as you can find out by studying the paper on the [Federated Averaging](#) algorithm, achieving convergence in a system with randomly sampled subsets of clients in each round can take a while, and it would be impractical to have to run hundreds of rounds in this interactive tutorial.

What we'll do instead is sample the set of clients once, and reuse the same set across rounds to speed up convergence (intentionally over-fitting to these few user's data). We leave it as an exercise for the reader to modify this tutorial to simulate random sampling - it is fairly easy to do (once you do, keep in mind that getting the model to converge may take a while).

```
[ ] sample_clients = emnist_train.client_ids[:NUM_CLIENTS]  
  
federated_train_data = make_federated_data(emnist_train, sample_clients)  
  
print('Number of client datasets: {}'.format(len(federated_train_data)))  
print('First dataset: {}'.format(d=federated_train_data[0]))
```

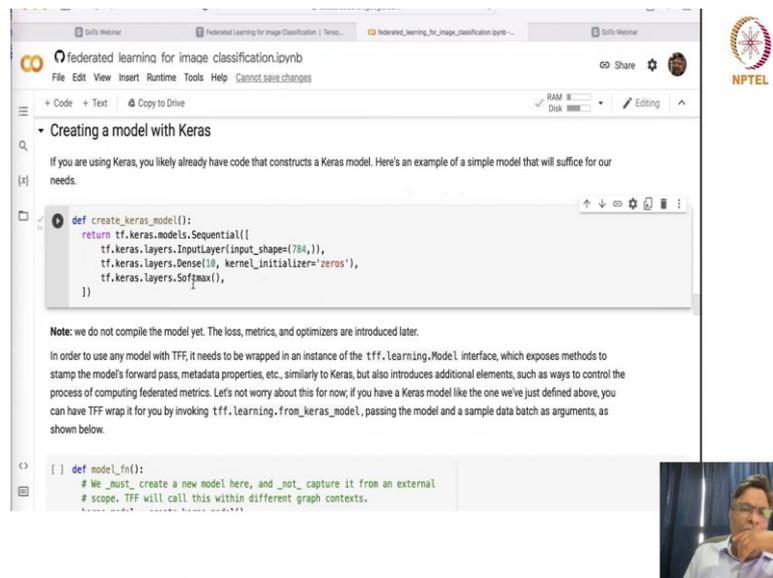
Number of client datasets: 10
First dataset: <PrefetchDataset shapes: OrderedDict([(x, (None, 784)), (y, (None, 1))]), types: OrderedDict([(x, tf.float32

In the sense these are all federated data sets right, each of these clients have their own data and it has to have its own mean, it has to have everything ok. And then there is some specific simple Colab TFF the function which gives you a federated data thing right. So, this function is going to help you construct a list of data sets from the given set of users as an input to a round of training on evaluation.

So, the idea here is, see you have got now this data which is split among various clients right. But out of all of these you are supposed to get the data ok, from the given set of users as a input to a round of training or evaluation right. Now how do we choose the clients? Ok.

So, generally in a typical federated training scenario we are dealing with a potentially large population this, this, this only fraction of which may be available for training at a particular time, right. So, the idea is when the client devices are mobile phones that participate in training only when plugged into a power source of a metered network and otherwise this. So, the idea is we are in a simulation environment and all the data is locally available here, right.

(Refer Slide Time: 17:09)



The screenshot shows a Jupyter Notebook interface with the following content:

```
def create_keras_model():
    return tf.keras.models.Sequential([
        tf.keras.layers.InputLayer(input_shape=(784,)),
        tf.keras.layers.Dense(10, kernel_initializer='zeros'),
        tf.keras.layers.Softmax(),
    ])
```

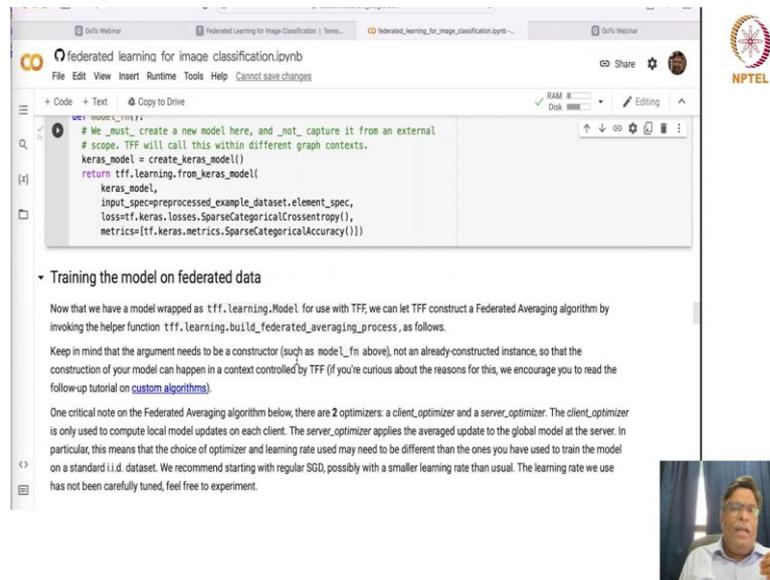
Note: we do not compile the model yet. The loss, metrics, and optimizers are introduced later.

In order to use any model with TFF, it needs to be wrapped in an instance of the `tff.LearningModel` interface, which exposes methods to stamp the model's forward pass, metadata properties, etc., similarly to Keras, but also introduces additional elements, such as ways to control the process of computing federated metrics. Let's not worry about this for now; if you have a Keras model like the one we've just defined above, you can have TFF wrap it for you by invoking `tff.learning.from_keras_model`, passing the model and a sample data batch as arguments, as shown below.

```
def model_fn():
    # We must create a new model here, and not capture it from an external
    # scope. TFF will call this within different graph contexts.
```

So, the idea is you can talk of something called as federated averaging algorithm in this paper, we are not going into this. But the basic idea is that you are going to now generate ok, the client data set right. So, there are 10 clients and each of these clients will have certain data with you ok. Now, we will have to develop our own model right and this is a KERAS model.

(Refer Slide Time: 17:48)



The screenshot shows a Jupyter Notebook interface with the following content:

```
def model_fn():  
    # We _must_ create a new model here, and _not_ capture it from an external  
    # scope. TFF will call this within different graph contexts.  
    keras_model = create_keras_model()  
    return tff.learning.from_keras_model(  
        keras_model,  
        input_spec=preprocessed_example_dataset.element_spec,  
        loss=tf.keras.losses.SparseCategoricalCrossentropy(),  
        metrics=[tf.keras.metrics.SparseCategoricalAccuracy()])
```

Training the model on federated data

Now that we have a model wrapped as `tff.learning.Model` for use with TFF, we can let TFF construct a Federated Averaging algorithm by invoking the helper function `tff.learning.build_federated_averaging_process`, as follows.

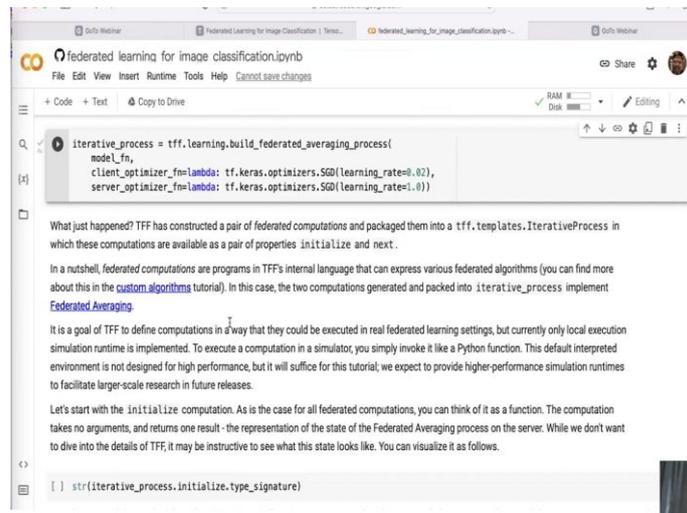
Keep in mind that the argument needs to be a constructor (such as `model_fn` above), not an already-constructed instance, so that the construction of your model can happen in a context controlled by TFF (if you're curious about the reasons for this, we encourage you to read the follow-up tutorial on [custom algorithms](#)).

One critical note on the Federated Averaging algorithm below, there are 2 optimizers: a `client_optimizer` and a `server_optimizer`. The `client_optimizer` is only used to compute local model updates on each client. The `server_optimizer` applies the averaged update to the global model at the server. In particular, this means that the choice of optimizer and learning rate used may need to be different than the ones you have used to train the model on a standard i.i.d. dataset. We recommend starting with regular SGD, possibly with a smaller learning rate than usual. The learning rate we use has not been carefully tuned, feel free to experiment.

So, I am not going into the details, I am just telling you that we are creating some model ok. This is a some training model ok and now when you develop some model here ok you have to use that model with the federated data right. So, you are supposed to interface this model ok in such a way that it is going to actually give you ok certain matrices or matrix which are federated in terms right.

So, we are going to actually use this TFF learning from this model passing the model and a sample data batches as argument. So, the basic idea is you are now trying to actually develop a model, interface it in such a manner that you can use that federated data which you have got ok for training your model ok.

(Refer Slide Time: 18:46)



```
iterative_process = tff.learning.build_federated_averaging_process(
    model_fn,
    client_optimizer_fn=lambda: tf.keras.optimizers.SGD(learning_rate=0.02),
    server_optimizer_fn=lambda: tf.keras.optimizers.SGD(learning_rate=1.0))
```

What just happened? TFF has constructed a pair of federated computations and packaged them into a `tff.templates.IterativeProcess` in which these computations are available as a pair of properties `initialize` and `next`.

In a nutshell, federated computations are programs in TFF's internal language that can express various federated algorithms (you can find more about this in the [custom algorithms](#) tutorial). In this case, the two computations generated and packed into `iterative_process` implement [Federated Averaging](#).

It is a goal of TFF to define computations in a way that they could be executed in real federated learning settings, but currently only local execution simulation runtime is implemented. To execute a computation in a simulator, you simply invoke it like a Python function. This default interpreted environment is not designed for high performance, but it will suffice for this tutorial; we expect to provide higher-performance simulation runtimes to facilitate larger-scale research in future releases.

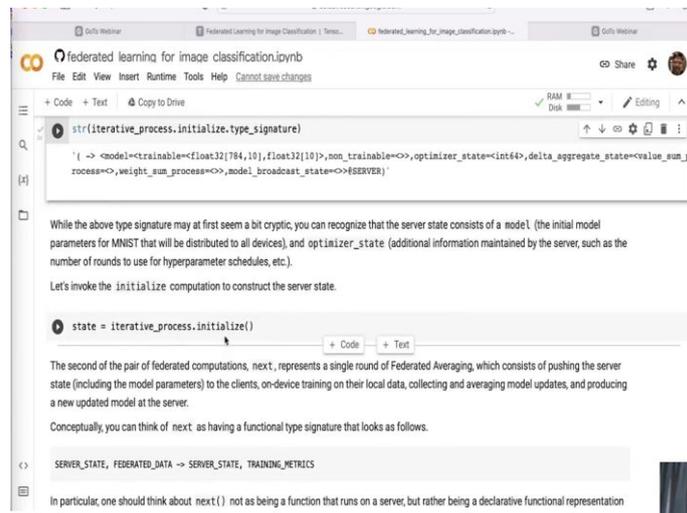
Let's start with the `initialize` computation. As is the case for all federated computations, you can think of it as a function. The computation takes no arguments, and returns one result - the representation of the state of the Federated Averaging process on the server. While we don't want to dive into the details of TFF, it may be instructive to see what this state looks like. You can visualize it as follows.

```
str(iterative_process.initialize.type_signature)
```



So, now since you have got this model wrapped as a TFF learning model ok for the use with TFF we can let TFF construct a federated averaging algorithms by invoking the helper function ok which basically does the averaging right. So, you are going to actually develop a model now ok, which is a iterative process do this and then ok.

(Refer Slide Time: 19:28)



```
str(iterative_process.initialize.type_signature)
```

```
('() -> (<model=<trainable=<float32[784, 10], float32[10]>, non_trainable=<>>, optimizer_state=<int64>, delta_aggregate_state=<value_sum_p<br>rocess=<>, weight_sum_process=<>>, model_broadcast_state=<>>#SERVER)')
```

While the above type signature may at first seem a bit cryptic, you can recognize that the server state consists of a `model` (the initial model parameters for MNIST that will be distributed to all devices), and `optimizer_state` (additional information maintained by the server, such as the number of rounds to use for hyperparameter schedules, etc.).

Let's invoke the `initialize` computation to construct the server state.

```
state = iterative_process.initialize()
```

The second of the pair of federated computations, `next`, represents a single round of Federated Averaging, which consists of pushing the server state (including the model parameters) to the clients, on-device training on their local data, collecting and averaging model updates, and producing a new updated model at the server.

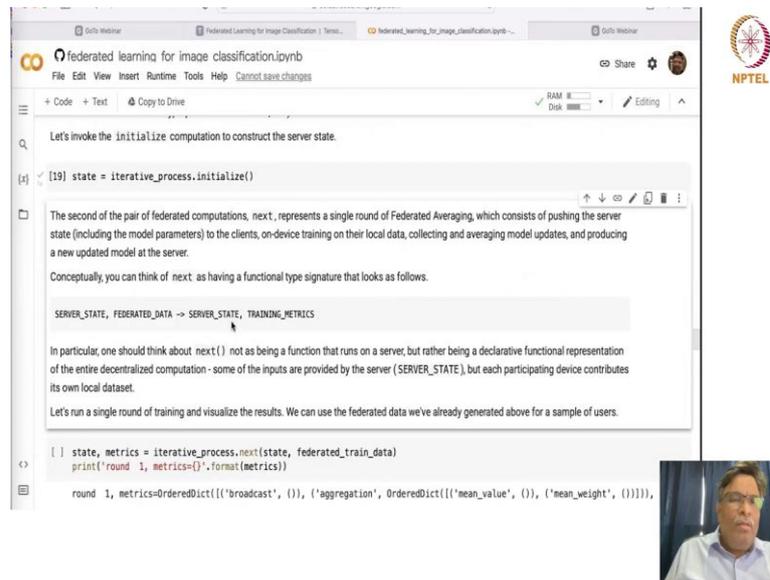
Conceptually, you can think of `next` as having a functional type signature that looks as follows.

```
SERVER_STATE, FEDERATED_DATA -> SERVER_STATE, TRAINING_METRICS
```

In particular, one should think about `next()` not as being a function that runs on a server, but rather being a declarative functional representation



(Refer Slide Time: 19:50)



The screenshot shows a Jupyter Notebook interface with the following content:

Let's invoke the `initialize` computation to construct the server state.

```
[19] state = iterative_process.initialize()
```

The second of the pair of federated computations, `next`, represents a single round of Federated Averaging, which consists of pushing the server state (including the model parameters) to the clients, on-device training on their local data, collecting and averaging model updates, and producing a new updated model at the server.

Conceptually, you can think of `next` as having a functional type signature that looks as follows.

```
SERVER_STATE, FEDERATED_DATA -> SERVER_STATE, TRAINING_METRICS
```

In particular, one should think about `next()` not as being a function that runs on a server, but rather being a declarative functional representation of the entire decentralized computation - some of the inputs are provided by the server (`SERVER_STATE`), but each participating device contributes its own local dataset.

Let's run a single round of training and visualize the results. We can use the federated data we've already generated above for a sample of users.

```
[ ] state, metrics = iterative_process.next(state, federated_train_data)
print('round 1, metrics={}'.format(metrics))

round 1, metrics=OrderedDict([('broadcast', {}), ('aggregation', OrderedDict([('mean_value', {}), ('mean_weight', {})]))])
```

Now, the idea here is we should understand here that this type of a signature what we are trying to see here. Because you are trying to actually generate iterative process, you are trying to develop a model which is trainable ok and you are trying to see that how many of this portion ok what data you are using for training, what data you are not going to use for training and then you will have to aggregate this information ok.

And then broadcast it to the server, what does it mean is that you will have a server state ok which basically is the initial model right. You would have a centralized trained MNIST model ok. And that model will be distributed to all the devices that is called as a server state initial model right so that is a model.

And then optimizer state which basically means this additional information which is maintained by the server which will include various parameters, hyper parameters and all of that ok is something which basically is the state of a server.

So, you are having a server state which talks about the basic model and then there is a optimizer state for the server which tells about like what are the various hyper parameters being used, how many epochs, how many rounds, how many iterations and so on and so forth.

So now, the idea is you need to initialize this process of training. Now once you are able to do it, once you have done a single iteration ok which is single round of iteration then

you will go in for a single round of federated averaging ok. So, when you do this averaging what effectively happens is you are pushing the server state to the client. Then you do this on device training on the local data, collect it back and then do averaging ok on that model which you updated the server right.

So, your server state is going to change with the data which it has used. So, your server state comma federated data changes to server state with training metrics. Because your federated data is not going to be shared, it is only the training matrix which is going to be shared right. So, this is how it is going to happen.

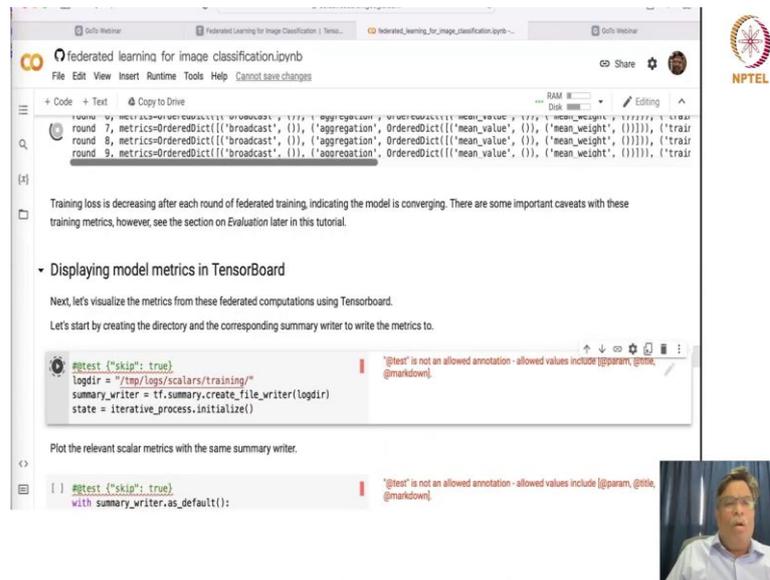
(Refer Slide Time: 22:31)

```
NUM_ROUNDS = 11
for round_num in range(2, NUM_ROUNDS):
    state, metrics = iterative_process.next(state, federated_train_data)
    print('round {}:'.format(round_num, metrics))

...
round 2, metrics=OrderedDict([('broadcast', {}), ('aggregation', OrderedDict([('mean_value', {}), ('mean_weight', {})])), ('train
round 3, metrics=OrderedDict([('broadcast', {}), ('aggregation', OrderedDict([('mean_value', {}), ('mean_weight', {})])), ('train
round 4, metrics=OrderedDict([('broadcast', {}), ('aggregation', OrderedDict([('mean_value', {}), ('mean_weight', {})])), ('train
round 5, metrics=OrderedDict([('broadcast', {}), ('aggregation', OrderedDict([('mean_value', {}), ('mean_weight', {})])), ('train
round 6, metrics=OrderedDict([('broadcast', {}), ('aggregation', OrderedDict([('mean_value', {}), ('mean_weight', {})])), ('train
```

So, let us try to understand this in a bit maybe once you do a hands on and read through this we will be able to understand it better, but at least a gist of how basically it happens ok is what we are trying to tell you right. So, now, you do so many rounds of it. So, many iterations ok, then this is how the matrix is going to change ok. So, mean weight, this, that, aggregation you are doing broadcasting all of this.

(Refer Slide Time: 23:02)



The screenshot shows a Jupyter Notebook titled "federated learning for image classification.ipynb". The code cell contains the following Python code:

```
round 7, metrics=OrderedDict([('broadcast', {}), ('aggregation', OrderedDict([('mean_value', {}), ('mean_weight', {})]))], ('train', {}))
round 8, metrics=OrderedDict([('broadcast', {}), ('aggregation', OrderedDict([('mean_value', {}), ('mean_weight', {})]))], ('train', {}))
round 9, metrics=OrderedDict([('broadcast', {}), ('aggregation', OrderedDict([('mean_value', {}), ('mean_weight', {})]))], ('train', {}))
```

Below the code, there is a text block explaining that training loss is decreasing and providing instructions on displaying model metrics in TensorBoard. The code cell below that shows the creation of a summary writer:

```
@test {"skip": true}
logdir = "/tmp/logs/scalars/training/"
summary_writer = tf.summary.create_file_writer(logdir)
state = iterative_process.initialize()
```

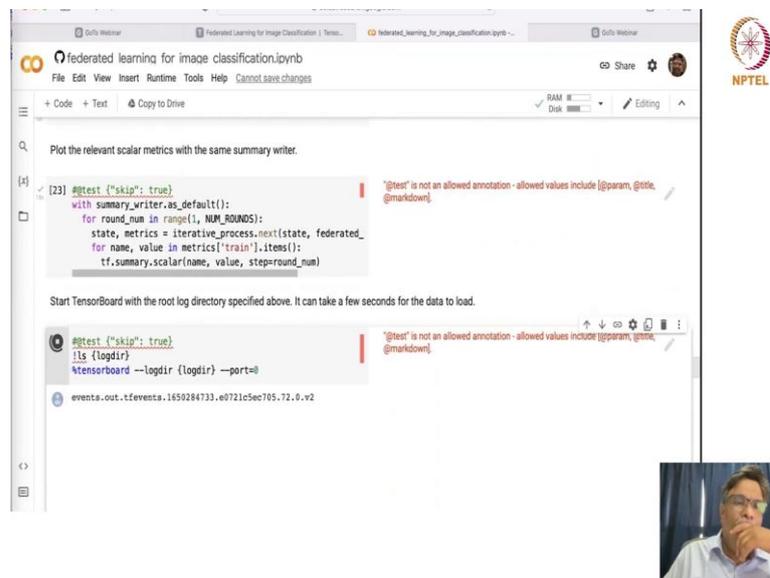
Another code cell shows the use of the summary writer:

```
@test {"skip": true}
with summary_writer.as_default():
```

The NPTEL logo is visible in the top right corner. A small video inset of a man is in the bottom right corner.

And then you can display this model matrix on the tensor board ok.

(Refer Slide Time: 23:07)



The screenshot shows a Jupyter Notebook with the following code cell:

```
[23] @test {"skip": true}
with summary_writer.as_default():
    for round_num in range(1, NUM_ROUNDS):
        state, metrics = iterative_process.next(state, federated_
        for name, value in metrics['train'].items():
            tf.summary.scalar(name, value, step=round_num)
```

Below the code, there is a text block instructing to start TensorBoard with the root log directory. The code cell below that shows the command to run TensorBoard:

```
@test {"skip": true}
!ls {logdir}
!tensorboard --logdir {logdir} --port=0
```

The output of the command is shown as:

```
events.out.tfevents.1659284733.e0721c5ec705.72.0.v2
```

The NPTEL logo is visible in the top right corner. A small video inset of a man is in the bottom right corner.

So, here the tensor board we will see in the end right, because since you are working on Colab. So, again I do not want to go now into tensor board log in and everything, but some thing here you will be able to see ok.

(Refer Slide Time: 23:36)

federated learning for image classification.ipynb

```
[ ] | #!test ("skip": true)
# Uncomment and run this cell to clean your directory of old
# future graphs from this directory. We don't run it by defau
# you do a "Runtime > Run all" you don't lose your results.

# !rm -R /tmp/logs/scalars/*
```

In order to view evaluation metrics the same way, you can create a separate eval folder, like "logs/scalars/eval", to write to TensorBoard.

Customizing the model implementation

Keras is the **recommended high-level model API for TensorFlow**, and we encourage using Keras models (via `tff.Learning.from_keras_model`) in TFF whenever possible.

However, `tff.Learning` provides a lower-level model interface, `tff.Learning.Model`, that exposes the minimal functionality necessary for using a model for federated learning. Directly implementing this interface (possibly still using building blocks like `tf.keras.layers`) allows for maximum customization without modifying the internals of the federated learning algorithms.

So let's do it all over again from scratch.

Defining model variables, forward pass, and metrics

(Refer Slide Time: 23:37)

federated learning for image classification.ipynb

Customizing the model implementation

Keras is the **recommended high-level model API for TensorFlow**, and we encourage using Keras models (via `tff.Learning.from_keras_model`) in TFF whenever possible.

However, `tff.Learning` provides a lower-level model interface, `tff.Learning.Model`, that exposes the minimal functionality necessary for using a model for federated learning. Directly implementing this interface (possibly still using building blocks like `tf.keras.layers`) allows for maximum customization without **modifying** the internals of the federated learning algorithms.

So let's do it all over again from scratch.

Defining model variables, forward pass, and metrics

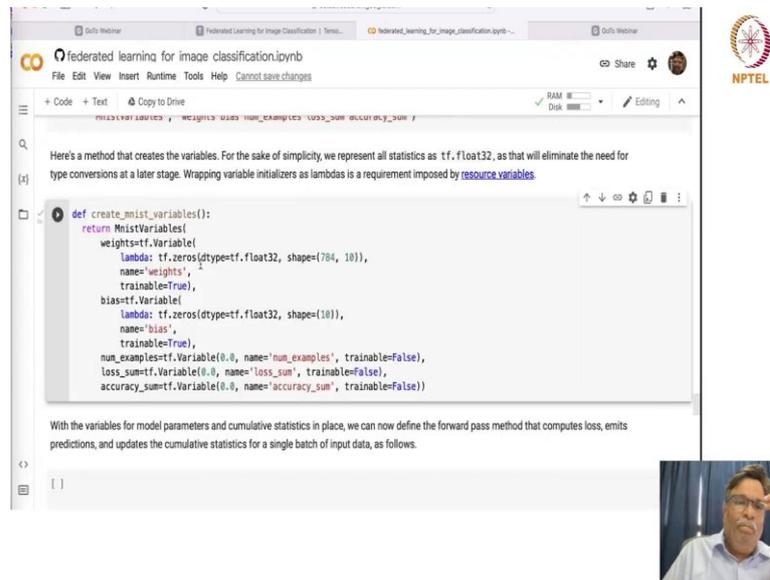
The first step is to identify the TensorFlow variables we're going to work with. In order to make the following code more legible, let's define a data structure to represent the entire set. This will include variables such as `weights` and `bias` that we will train, as well as variables that will hold various cumulative statistics and counters we will update during training, such as `loss_sum`, `accuracy_sum`, and `num_examples`.

```
[25] MnistVariables = collections.namedtuple(
    'MnistVariables', 'weights bias num_examples loss_sum accuracy_sum')
```

Here's a method that creates the variables. For the sake of simplicity, we represent all statistics as `tf.float32`, as that will eliminate the need for type conversions at a later stage. Wrapping variable initializers as lambdas is a requirement imposed by [resource variables](#).

So, you actually can go deep into it, a lower level model interface wherein you are trying to actually go at the variables level ok. That you can create variables that basically means you can actually customize your model implementation.

(Refer Slide Time: 24:00)



federated learning for image classification.ipynb

File Edit View Insert Runtime Tools Help Cannot save changes

+ Code + Text Copy to Drive RAM Disk Editing

Here's a method that creates the variables. For the sake of simplicity, we represent all statistics as `tf.float32`, as that will eliminate the need for type conversions at a later stage. Wrapping variable initializers as lambdas is a requirement imposed by `resource_variables`.

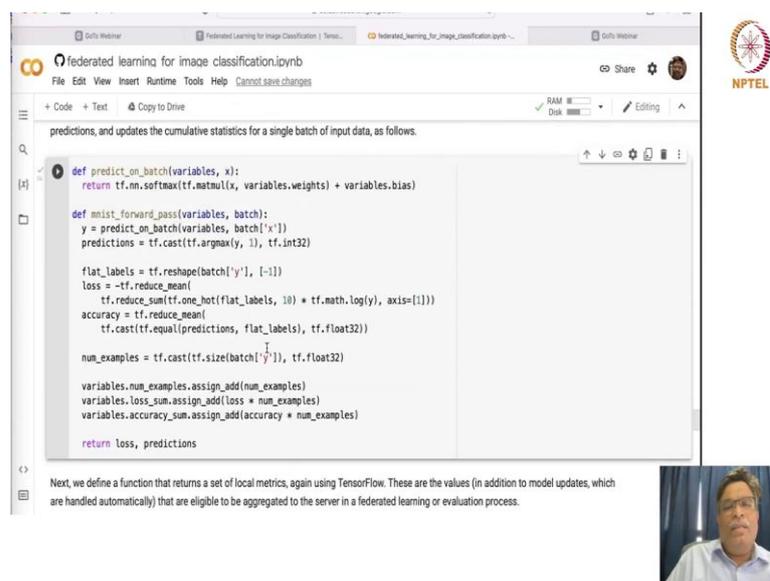
```
[+] def create_mnist_variables():
    return MnistVariables(
        weights=tf.Variable(
            lambda: tf.zeros(dtype=tf.float32, shape=(784, 10)),
            name='weights',
            trainable=True),
        bias=tf.Variable(
            lambda: tf.zeros(dtype=tf.float32, shape=(10)),
            name='bias',
            trainable=True),
        num_examples=tf.Variable(0.0, name='num_examples', trainable=False),
        loss_sum=tf.Variable(0.0, name='loss_sum', trainable=False),
        accuracy_sum=tf.Variable(0.0, name='accuracy_sum', trainable=False))
```

With the variables for model parameters and cumulative statistics in place, we can now define the forward pass method that computes loss, emits predictions, and updates the cumulative statistics for a single batch of input data, as follows.

[]



(Refer Slide Time: 24:00)



federated learning for image classification.ipynb

File Edit View Insert Runtime Tools Help Cannot save changes

+ Code + Text Copy to Drive RAM Disk Editing

predictions, and updates the cumulative statistics for a single batch of input data, as follows.

```
[+] def predict_on_batch(variables, x):
    return tf.nn.softmax(tf.matmul(x, variables.weights) + variables.bias)

def mnist_forward_pass(variables, batch):
    y = predict_on_batch(variables, batch['x'])
    predictions = tf.cast(tf.argmax(y, 1), tf.int32)

    flat_labels = tf.reshape(batch['y'], [-1])
    loss = -tf.reduce_mean(
        tf.reduce_sum(tf.one_hot(flat_labels, 10) * tf.math.log(y), axis=[1]))
    accuracy = tf.reduce_mean(
        tf.cast(tf.equal(predictions, flat_labels), tf.float32))

    num_examples = tf.cast(tf.size(batch['x']), tf.float32)

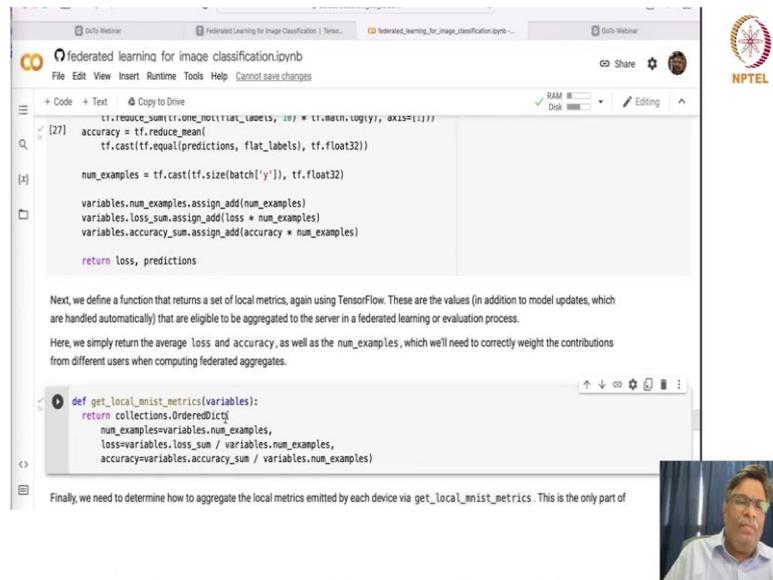
    variables.num_examples.assign_add(num_examples)
    variables.loss_sum.assign_add(loss * num_examples)
    variables.accuracy_sum.assign_add(accuracy * num_examples)

    return loss, predictions
```

Next, we define a function that returns a set of local metrics, again using TensorFlow. These are the values (in addition to model updates, which are handled automatically) that are eligible to be aggregated to the server in a federated learning or evaluation process.



(Refer Slide Time: 24:05)



federated learning for image classification.ipynb

```
tf.reduce_sum(tf.one_hot(tf.argmax(logits, axis=-1), num_classes)) / tf.reduce_sum(tf.ones_like(logits))
```

[27] accuracy = tf.reduce_mean(tf.cast(tf.equal(predictions, flat_labels), tf.float32))

```
num_examples = tf.cast(tf.size(batch['y']), tf.float32)
```

```
variables.num_examples.assign_add(num_examples)
```

```
variables.loss_sum.assign_add(loss * num_examples)
```

```
variables.accuracy_sum.assign_add(accuracy * num_examples)
```

```
return loss, predictions
```

Next, we define a function that returns a set of local metrics, again using TensorFlow. These are the values (in addition to model updates, which are handled automatically) that are eligible to be aggregated to the server in a federated learning or evaluation process.

Here, we simply return the average loss and accuracy, as well as the num_examples, which we'll need to correctly weight the contributions from different users when computing federated aggregates.

```
def get_local_metrics(variables):
```

```
    return collections.OrderedDict({
```

```
        'num_examples': variables.num_examples,
```

```
        'loss': variables.loss_sum / variables.num_examples,
```

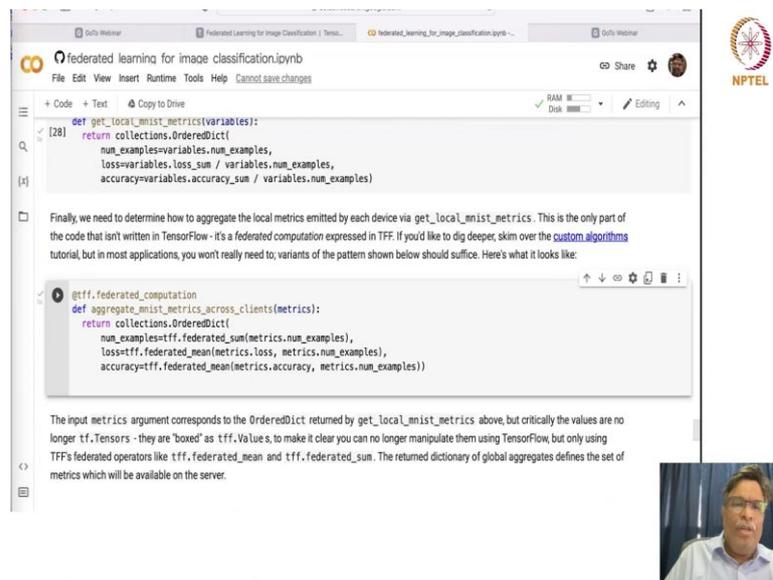
```
        'accuracy': variables.accuracy_sum / variables.num_examples
```

```
    })
```

Finally, we need to determine how to aggregate the local metrics emitted by each device via get_local_metrics. This is the only part of



(Refer Slide Time: 24:09)



federated learning for image classification.ipynb

```
def get_local_metrics(variables):
```

```
    return collections.OrderedDict({
```

```
        'num_examples': variables.num_examples,
```

```
        'loss': variables.loss_sum / variables.num_examples,
```

```
        'accuracy': variables.accuracy_sum / variables.num_examples
```

```
    })
```

Finally, we need to determine how to aggregate the local metrics emitted by each device via get_local_metrics. This is the only part of the code that isn't written in TensorFlow - it's a federated computation expressed in TFF. If you'd like to dig deeper, skim over the [custom algorithms](#) tutorial, but in most applications, you won't really need to, variants of the pattern shown below should suffice. Here's what it looks like:

```
@tff.federated_computation
```

```
def aggregate_metrics_across_clients(metrics):
```

```
    return collections.OrderedDict({
```

```
        'num_examples': tff.federated_sum(metrics.num_examples),
```

```
        'loss': tff.federated_mean(metrics.loss, metrics.num_examples),
```

```
        'accuracy': tff.federated_mean(metrics.accuracy, metrics.num_examples)
```

```
    })
```

The input metrics argument corresponds to the OrderedDict returned by get_local_metrics above, but critically the values are no longer tf.Tensors - they are "boxed" as tff.Value s, to make it clear you can no longer manipulate them using TensorFlow, but only using TFF's federated operators like tff.federated_mean and tff.federated_sum. The returned dictionary of global aggregates defines the set of metrics which will be available on the server.



(Refer Slide Time: 24:14)

```

from typing import Callable, List, OrderedDict

class MnistModel(tff.Learning.Model):
    def __init__(self):
        self._variables = create_mnist_variables()

    @property
    def trainable_variables(self):
        return [self._variables.weights, self._variables.bias]

    @property
    def non_trainable_variables(self):
        return []

```

And then you try to collect the statistics, local metrics, do the federated computations and then you can use ok again another model representation, ok. Instead of the general KERAS model which we did.

(Refer Slide Time: 24:39)

```

def local_variables(self):
    return [
        self._variables.num_examples, self._variables.loss_sum,
        self._variables.accuracy_sum
    ]

@property
def input_spec(self):
    return collections.OrderedDict(
        x=tf.TensorSpec([None, 784], tf.float32),
        y=tf.TensorSpec([None, 1], tf.int32))

@tf.function
def predict_on_batch(self, x, training=True):
    del training
    return predict_on_batch(self._variables, x)

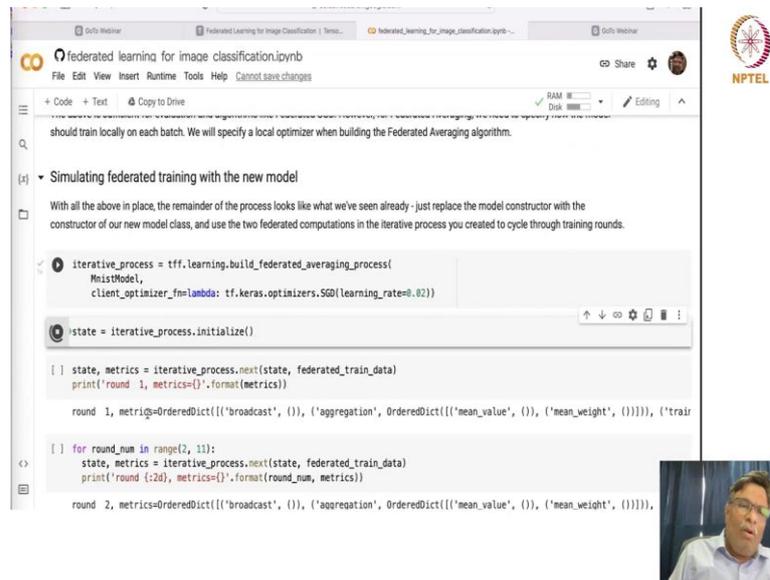
@tf.function
def forward_pass(self, batch, training=True):
    del training
    loss, predictions = mnist_forward_pass(self._variables, batch)
    num_examples = tf.shape(batch['x'])[0]
    return tff.learning.BatchOutput(
        loss=loss, predictions=predictions, num_examples=num_examples)

@tf.function
def report_local_outputs(self):
    return get_local_mnist_metrics(self._variables)

```

So, what we are trying to do is you can work at different levels of complexities right.

(Refer Slide Time: 24:46)



```
should train locally on each batch. We will specify a local optimizer when building the Federated Averaging algorithm.
```

Simulating federated training with the new model

With all the above in place, the remainder of the process looks like what we've seen already - just replace the model constructor with the constructor of our new model class, and use the two federated computations in the iterative process you created to cycle through training rounds.

```
iterative_process = tff.learning.build_federated_averaging_process(
    MnistModel,
    client_optimizer_fn=Lambda(lambda: tf.keras.optimizers.SGD(learning_rate=0.02))

state = iterative_process.initialize()

state, metrics = iterative_process.next(state, federated_train_data)
print('round 1, metrics={}'.format(metrics))

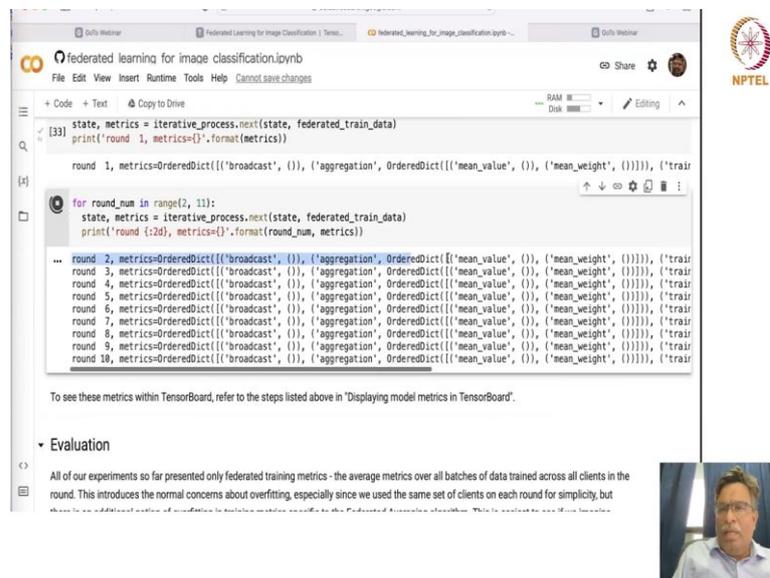
round 1, metrics=OrderedDict([('broadcast', {}), ('aggregation', OrderedDict([('mean_value', {}), ('mean_weight', {})]), ('train

for round_num in range(2, 11):
    state, metrics = iterative_process.next(state, federated_train_data)
    print('round {}:2d, metrics={}'.format(round_num, metrics))

round 2, metrics=OrderedDict([('broadcast', {}), ('aggregation', OrderedDict([('mean_value', {}), ('mean_weight', {})]), ('train
```

And you can make your own model instead of using a standard TFF KERAS model ok.

(Refer Slide Time: 24:50)



```
state, metrics = iterative_process.next(state, federated_train_data)
print('round 1, metrics={}'.format(metrics))

round 1, metrics=OrderedDict([('broadcast', {}), ('aggregation', OrderedDict([('mean_value', {}), ('mean_weight', {})]), ('train

for round_num in range(2, 11):
    state, metrics = iterative_process.next(state, federated_train_data)
    print('round {}:2d, metrics={}'.format(round_num, metrics))

... round 2, metrics=OrderedDict([('broadcast', {}), ('aggregation', OrderedDict([('mean_value', {}), ('mean_weight', {})]), ('train
round 3, metrics=OrderedDict([('broadcast', {}), ('aggregation', OrderedDict([('mean_value', {}), ('mean_weight', {})]), ('train
round 4, metrics=OrderedDict([('broadcast', {}), ('aggregation', OrderedDict([('mean_value', {}), ('mean_weight', {})]), ('train
round 5, metrics=OrderedDict([('broadcast', {}), ('aggregation', OrderedDict([('mean_value', {}), ('mean_weight', {})]), ('train
round 6, metrics=OrderedDict([('broadcast', {}), ('aggregation', OrderedDict([('mean_value', {}), ('mean_weight', {})]), ('train
round 7, metrics=OrderedDict([('broadcast', {}), ('aggregation', OrderedDict([('mean_value', {}), ('mean_weight', {})]), ('train
round 8, metrics=OrderedDict([('broadcast', {}), ('aggregation', OrderedDict([('mean_value', {}), ('mean_weight', {})]), ('train
round 9, metrics=OrderedDict([('broadcast', {}), ('aggregation', OrderedDict([('mean_value', {}), ('mean_weight', {})]), ('train
round 10, metrics=OrderedDict([('broadcast', {}), ('aggregation', OrderedDict([('mean_value', {}), ('mean_weight', {})]), ('train

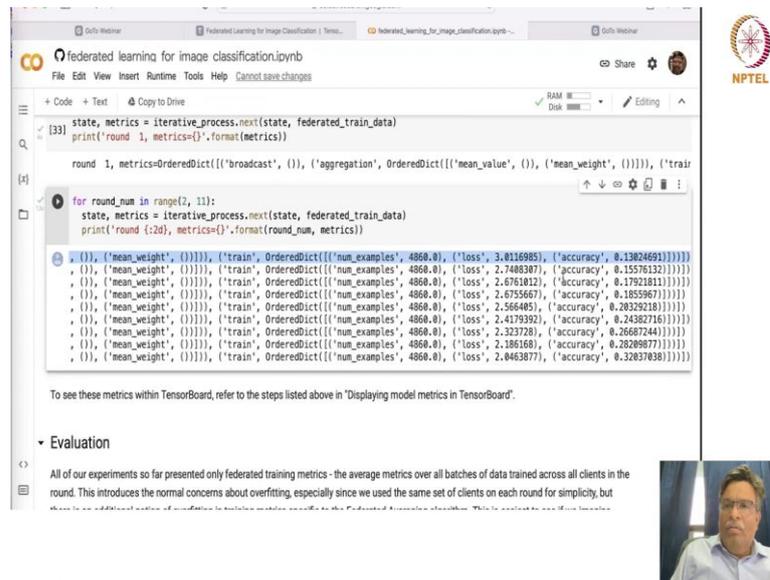
To see these metrics within TensorBoard, refer to the steps listed above in 'Displaying model metrics in TensorBoard'.
```

Evaluation

All of our experiments so far presented only federated training metrics - the average metrics over all batches of data trained across all clients in the round. This introduces the normal concerns about overfitting, especially since we used the same set of clients on each round for simplicity, but there is an additional concern of overfitting to the specific clients used in the Federated Averaging algorithm. This is addressed in the next slide.

So, after training right here right, you are going to actually get after each round these matrices ok, these are metrics right.

(Refer Slide Time: 25:13)



```
state, metrics = iterative_process.next(state, federated_train_data)
print('round 1, metrics={}'.format(metrics))

round 1, metrics=OrderedDict([('broadcast', {}), ('aggregation', OrderedDict([('mean_value', {}), ('mean_weight', {})]))])

for round_num in range(2, 11):
    state, metrics = iterative_process.next(state, federated_train_data)
    print('round {}:2d, metrics={}'.format(round_num, metrics))

[0.0], ('mean_weight', {}), ('train', OrderedDict([('num_examples', 4868.0), ('loss', 3.0116985), ('accuracy', 0.13824691)]))
, (), ('mean_weight', {}), ('train', OrderedDict([('num_examples', 4868.0), ('loss', 2.7488387), ('accuracy', 0.15576132)]))
, (), ('mean_weight', {}), ('train', OrderedDict([('num_examples', 4868.0), ('loss', 2.6761812), ('accuracy', 0.17921811)]))
, (), ('mean_weight', {}), ('train', OrderedDict([('num_examples', 4868.0), ('loss', 2.6755667), ('accuracy', 0.1855967)]))
, (), ('mean_weight', {}), ('train', OrderedDict([('num_examples', 4868.0), ('loss', 2.566485), ('accuracy', 0.28329218)]))
, (), ('mean_weight', {}), ('train', OrderedDict([('num_examples', 4868.0), ('loss', 2.4179392), ('accuracy', 0.24382716)]))
, (), ('mean_weight', {}), ('train', OrderedDict([('num_examples', 4868.0), ('loss', 2.323728), ('accuracy', 0.26687244)]))
, (), ('mean_weight', {}), ('train', OrderedDict([('num_examples', 4868.0), ('loss', 2.186168), ('accuracy', 0.28289877)]))
, (), ('mean_weight', {}), ('train', OrderedDict([('num_examples', 4868.0), ('loss', 2.0463877), ('accuracy', 0.32837838)]))
```

To see these metrics within TensorBoard, refer to the steps listed above in 'Displaying model metrics in TensorBoard'.

Evaluation

All of our experiments so far presented only federated training metrics - the average metrics over all batches of data trained across all clients in the round. This introduces the normal concerns about overfitting, especially since we used the same set of clients on each round for simplicity, but there is an additional notion of overfitting in training metrics specific to the Federated Averaging algorithm. This is easiest to see if we imagine each client had a single batch of data, and we train on that batch for many iterations (epochs). In this case, the local model will quickly exactly fit to that one batch, and so the local accuracy metric we average will approach 1.0. Thus, these training metrics can be taken as a sign that training is progressing, but not much more.

To perform evaluation on federated data, you can construct another `federated computation` designed for just this purpose, using the `tff.learning.build_federated_evaluation` function, and passing in your model constructor as an argument. Note that unlike with Federated Averaging, where we've used `MnistTrainableModel`, it suffices to pass the `MnistModel`. Evaluation doesn't perform gradient descent, and there's no need to construct optimizers.

For experimentation and research, when a centralized `test` dataset is available, [Federated Learning for Text Generation](#) demonstrates another evaluation option: taking the trained weights from federated learning, applying them to a standard Keras model, and then simply calling `tf.keras.models.Model.evaluate()` on a centralized dataset.

```
[ ] evaluation = tff.learning.build_federated_evaluation(MnistModel)
```

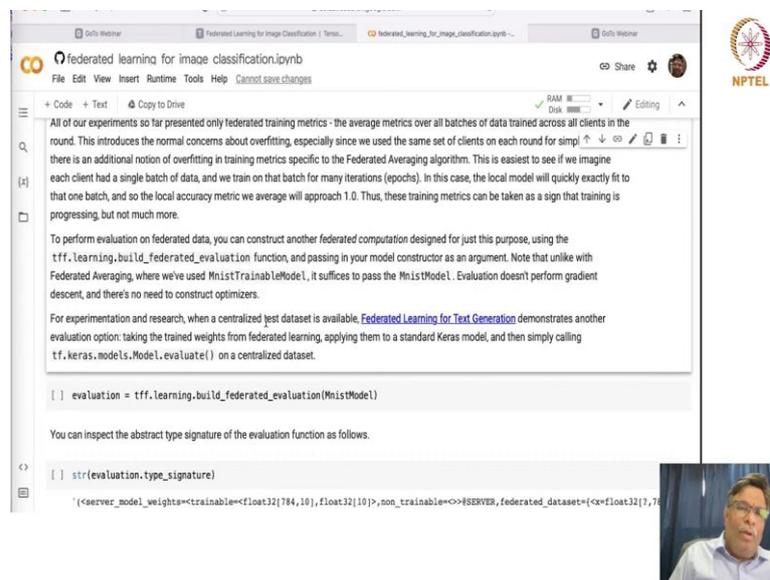
You can inspect the abstract type signature of the evaluation function as follows.

```
[ ] str(evaluation.type_signature)

'(<server_model_weights=<trainable=<float32[784,10],float32[10]>>,non_trainable=<>>SERVER,federated_dataset=<=<float32[7,7
```

So, the details about this you can refer or we will tell you what it is, but the basic idea is this is how these metrics would be shared, right.

(Refer Slide Time: 25:22)



```
[ ] evaluation = tff.learning.build_federated_evaluation(MnistModel)
```

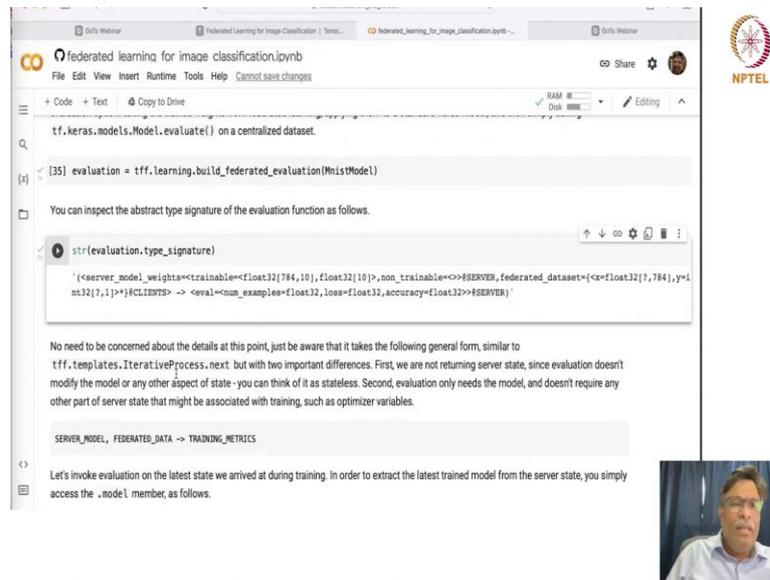
You can inspect the abstract type signature of the evaluation function as follows.

```
[ ] str(evaluation.type_signature)

'(<server_model_weights=<trainable=<float32[784,10],float32[10]>>,non_trainable=<>>SERVER,federated_dataset=<=<float32[7,7
```

So, the idea is that once you are able to train, get those metrics back then you will have to evaluate right which is basically evaluation. Then you will have to come up with TFF learning build federated evaluation function ok.

(Refer Slide Time: 25:39)



The screenshot shows a Jupyter Notebook titled "federated learning for image classification.ipynb". The code cell contains the following:

```
tff.keras.models.Model.evaluate() on a centralized dataset.  
  
[35] evaluation = tff.learning.build_federated_evaluation(MnistModel)
```

Below the code, there is explanatory text: "You can inspect the abstract type signature of the evaluation function as follows." This is followed by a code cell that prints the type signature:

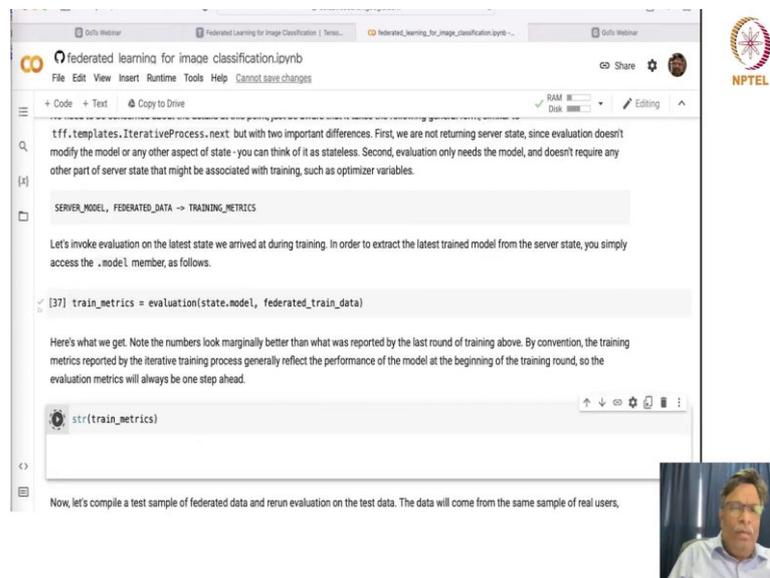
```
str(evaluation.type_signature)  
'(<server_model_weights=<trainable=<float32[784,10],float32[10]>,non_trainable=<>>SERVER,federated_dataset=<=<float32[7,784],y=1  
nt32[7,1]>*>CLIENTS> -> <eval=<num_examples=<float32,loss=<float32,accuracy=<float32>>SERVER)'
```

Further text explains that the evaluation function is stateless and only needs the model. A variable mapping is shown: SERVER_MODEL, FEDERATED_DATA -> TRAINING_METRICS. The final text says: "Let's invoke evaluation on the latest state we arrived at during training. In order to extract the latest trained model from the server state, you simply access the .model member, as follows."

A small video inset in the bottom right corner shows a man with glasses speaking.

So, you can evaluate these evaluation function ok on the central data set.

(Refer Slide Time: 25:50)



The screenshot shows the same Jupyter Notebook. The code cell now includes:

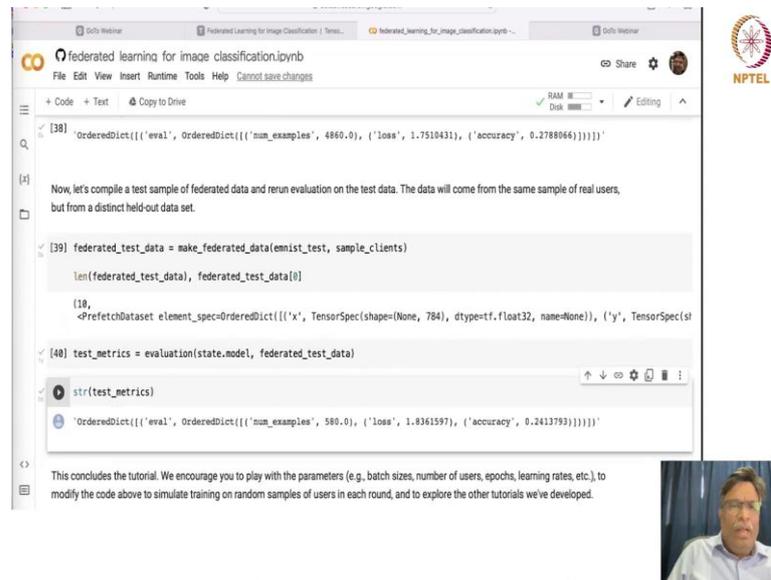
```
train_metrics = evaluation(state.model, federated_train_data
```

The text below explains that the training metrics are better than the last round. A variable mapping is shown: SERVER_MODEL, FEDERATED_DATA -> TRAINING_METRICS. The final text says: "Now, let's compile a test sample of federated data and run evaluation on the test data. The data will come from the same sample of real users."

A small video inset in the bottom right corner shows the same man speaking.

And then evaluation you do it.

(Refer Slide Time: 25:56)



The screenshot shows a Jupyter Notebook interface with the following content:

```
[38] 'OrderedDict({'eval': OrderedDict({'num_examples': 4860.0), ('loss', 1.7510431), ('accuracy', 0.2788966)}))'
```

Now, let's compile a test sample of federated data and rerun evaluation on the test data. The data will come from the same sample of real users, but from a distinct held-out data set.

```
[39] federated_test_data = make_federated_data(emnist_test, sample_clients)
len(federated_test_data), federated_test_data[0]
(18,
 <PrefetchDataset element_spec=OrderedDict({'x': TensorSpec(shape=(None, 784), dtype=tf.float32, name=None)), ('y', TensorSpec(st
[40] test_metrics = evaluation(state.model, federated_test_data)
str(test_metrics)
'OrderedDict({'eval': OrderedDict({'num_examples': 586.0), ('loss', 1.8361597), ('accuracy', 0.2413793)}))'
```

This concludes the tutorial. We encourage you to play with the parameters (e.g., batch sizes, number of users, epochs, learning rates, etc.) to modify the code above to simulate training on random samples of users in each round, and to explore the other tutorials we've developed.

The NPTEL logo is visible in the top right corner. A small video inset in the bottom right shows a man speaking.

So, you can test the metrics on the data and then you can see that number of examples, the loss and the accuracy. So, the idea is these are a sum of and average federated learning right algorithm which you use will be combination of all of this accuracies here all of the losses here all of the number of examples here ok.

And that would be using some model right which is there in the paper you can just refer to that right then you will be able to average it and then you can change these parameters as it has been it is there in this notebook of changing batch sizes number of users epoch learning rate all those hyper parameters you can change.

The basic idea is that so one of the evaluations would require the model one of the evaluations will not require the model ok and it will not require any server state which will not require any optimizer variable. So, you will understand this that here this particular type of what to say server state information what to say exchange right is just for training metrics. Whereas, there is another type wherein here you are going to actually get ok so many of these things.

Wherein your server state and this changes and you get server state along with variables as well right apart from metrics right. So, these are certain things which is there. So, I had actually three more examples. So, maybe once I cover tomorrow better approach using something called as NVIDIA FLARE maybe we will do this or we will continue this and extend that maybe by another 15-20 minutes to cover that as well right.

So, that is what I am thinking and yeah that is it. So, any questions if you have you can ask me or you can try other three of the labs which are given here tonight and maybe tomorrow maybe we can do it a bit fast right so that you know we can cover other things as well yeah. So, that is it from my side.