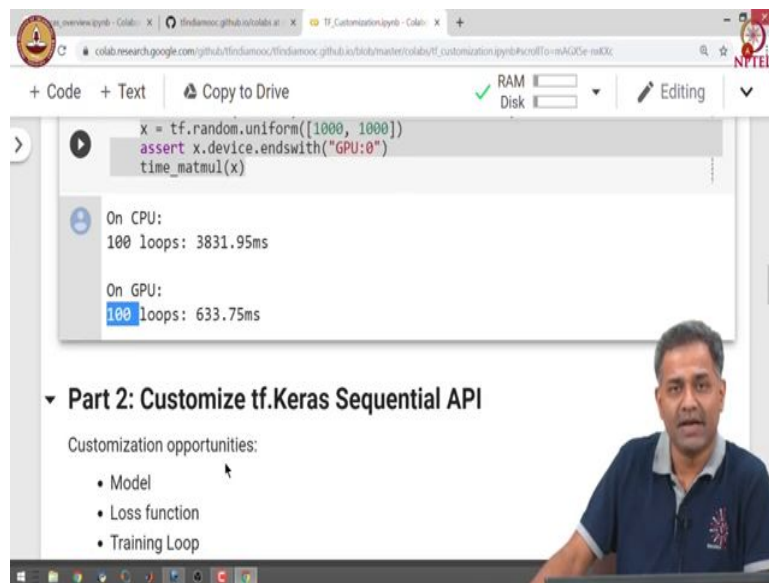


Practical Machine Learning with Tensorflow
Dr. Ashish Tendulkar
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture - 36
Customizing tf.keras – Part 1

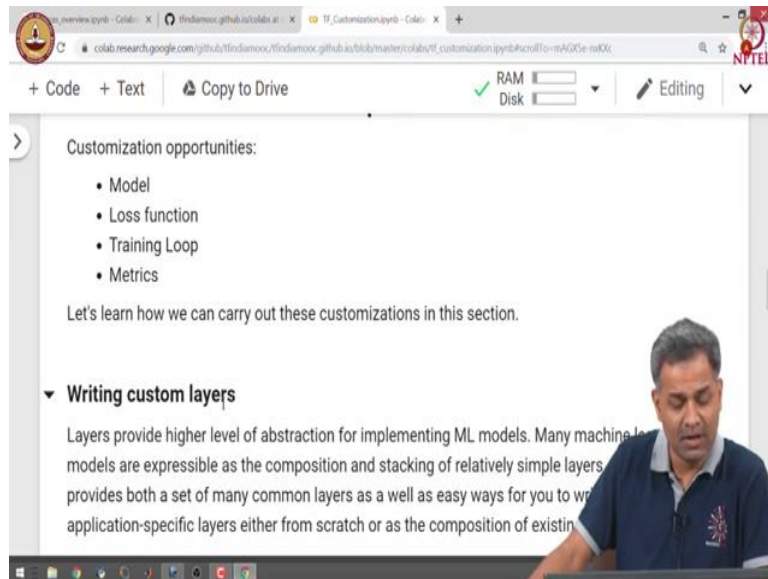
We just learnt how to perform a specific operation on the device that we intend to use.

(Refer Slide Time: 00:20)



Let us move on to understand how to customize tf.keras sequential API.

(Refer Slide Time: 00:28)



The screenshot shows a video lecture interface. The main content is a slide titled "Customization opportunities:" with a bulleted list: Model, Loss function, Training Loop, and Metrics. Below the list, it says "Let's learn how we can carry out these customizations in this section." A section titled "Writing custom layers" is expanded, showing text about layers providing abstraction for ML models. A video of a man in a blue shirt is visible in the bottom right corner of the slide.

Customization opportunities:

- Model
- Loss function
- Training Loop
- Metrics

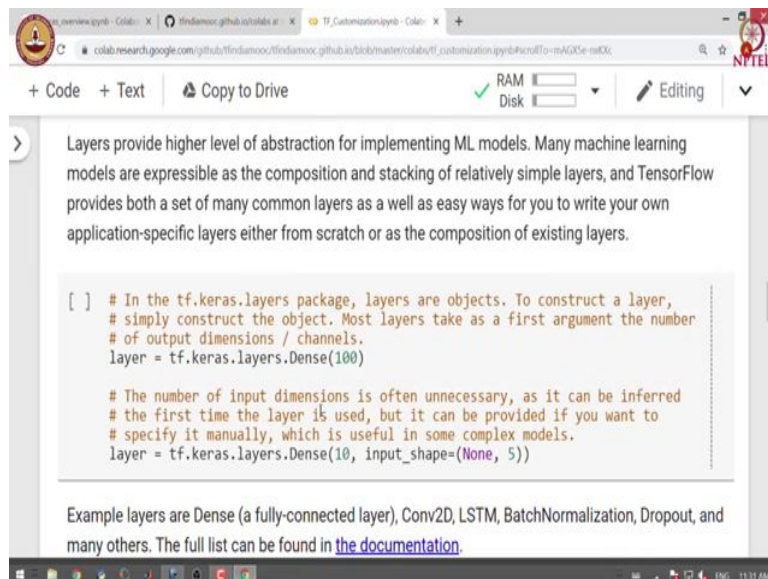
Let's learn how we can carry out these customizations in this section.

▼ Writing custom layers

Layers provide higher level of abstraction for implementing ML models. Many machine learning models are expressible as the composition and stacking of relatively simple layers, and TensorFlow provides both a set of many common layers as well as easy ways for you to write your own application-specific layers either from scratch or as the composition of existing layers.

There are customization opportunities in model, in writing a new loss function or training loop or using a new metric for measuring the performance of the model. Let us learn how to carry out this customization in this section.

(Refer Slide Time: 00:54)



The screenshot shows a video lecture interface. The main content is a slide titled "Layers provide higher level of abstraction for implementing ML models. Many machine learning models are expressible as the composition and stacking of relatively simple layers, and TensorFlow provides both a set of many common layers as well as easy ways for you to write your own application-specific layers either from scratch or as the composition of existing layers." Below the text is a code block showing how to create a Dense layer in TensorFlow. At the bottom, it lists example layers like Dense, Conv2D, LSTM, BatchNormalization, Dropout, and others, with a link to the documentation.

Layers provide higher level of abstraction for implementing ML models. Many machine learning models are expressible as the composition and stacking of relatively simple layers, and TensorFlow provides both a set of many common layers as well as easy ways for you to write your own application-specific layers either from scratch or as the composition of existing layers.

```
[ ] # In the tf.keras.layers package, layers are objects. To construct a layer,
# simply construct the object. Most layers take as a first argument the number
# of output dimensions / channels.
layer = tf.keras.layers.Dense(100)

# The number of input dimensions is often unnecessary, as it can be inferred
# the first time the layer is used, but it can be provided if you want to
# specify it manually, which is useful in some complex models.
layer = tf.keras.layers.Dense(10, input_shape=(None, 5))
```

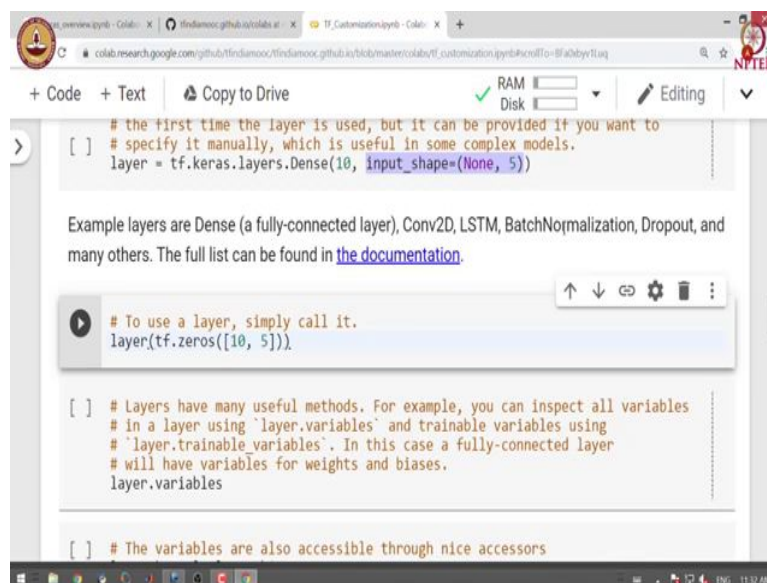
Example layers are Dense (a fully-connected layer), Conv2D, LSTM, BatchNormalization, Dropout, and many others. The full list can be found in [the documentation](#).

We will first study how to write custom layers. As you know layer provides high level abstraction for implementing machine learning models. Many machine learning models are expressed as composition and stacking of relatively simple layers. Tensor flow provides both

the set of many common layers as well as easy ways to write your own application specific layer either from scratch or as a composition of existing layers. We have already used layers in various machine learning operations.

These are typical statement that we have seen many times in this course. So, this particular statement defines a dense layer with 100 units. We can also specify sometimes the input shape along with the number of units in the layer.

(Refer Slide Time: 02:01)

A screenshot of a Google Colab notebook interface. The top bar shows the 'Code' tab selected, with options for 'Copy to Drive', 'RAM', 'Disk', and 'Editing'. The notebook content includes several code cells. The first cell defines a dense layer:

```
[ ] # the first time the layer is used, but it can be provided if you want to
[ ] # specify it manually, which is useful in some complex models.
layer = tf.keras.layers.Dense(10, input_shape=(None, 5))
```

 The second cell contains explanatory text: 'Example layers are Dense (a fully-connected layer), Conv2D, LSTM, BatchNormalization, Dropout, and many others. The full list can be found in [the documentation](#).' The third cell demonstrates using the layer:

```
[ ] # To use a layer, simply call it.
layer(tf.zeros([10, 5]))
```

 The fourth cell shows how to inspect layer variables:

```
[ ] # Layers have many useful methods. For example, you can inspect all variables
# in a layer using 'layer.variables' and trainable variables using
# 'layer.trainable_variables'. In this case a fully-connected layer
# will have variables for weights and biases.
layer.variables
```

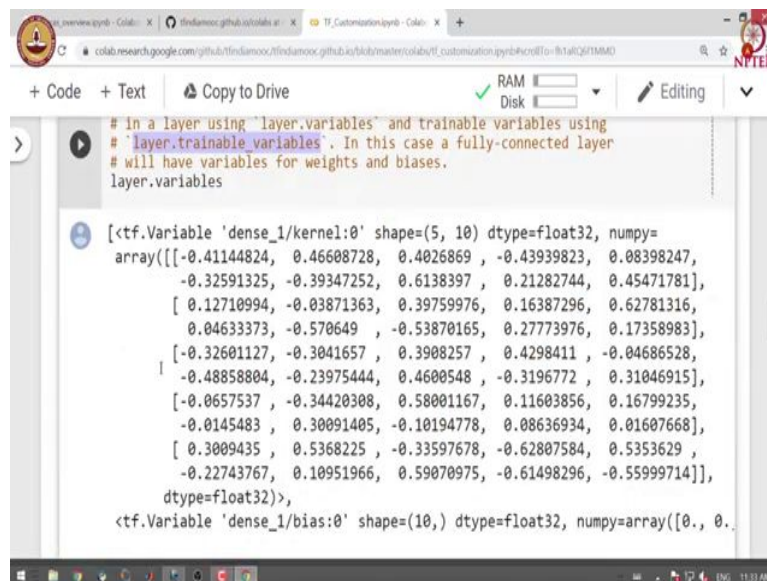
 The fifth cell shows an accessor:

```
[ ] # The variables are also accessible through nice accessors
```

So, there are layers like dense layer, conv 2D layer, LSTM, batch, normalization, dropout and many other layers are already defined by keras. In order to use layer, we simply call layer something like this. Here we call layer with a tensor which is a matrix which is 10x5 matrix of zeros. Layers have many useful methods, we can inspect all variables in the layer using `layer.variables`.

A trainable variables can be checked using `layer.trainable_variable`. The variables have weights and biases.

(Refer Slide Time: 03:23)

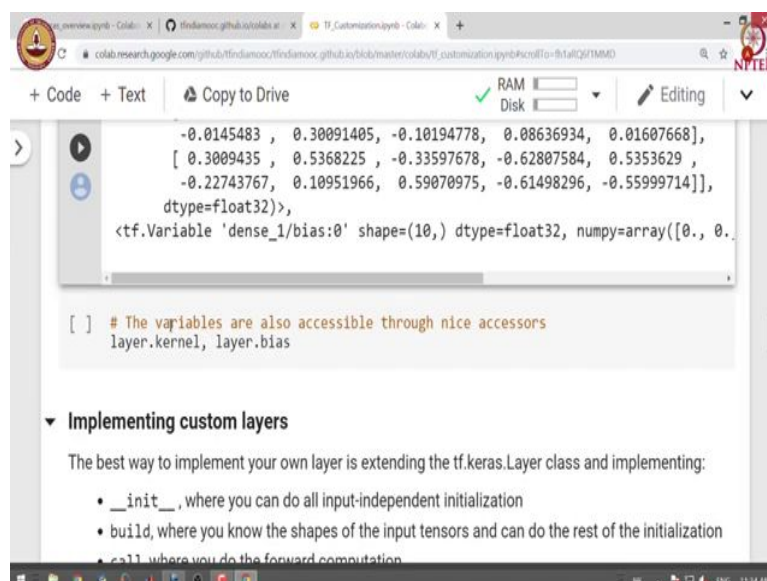


```
# In a layer using "layer.variables" and trainable variables using
# "layer.trainable_variables". In this case a fully-connected layer
# will have variables for weights and biases.
layer.variables

[<tf.Variable 'dense_1/kernel:0' shape=(5, 10) dtype=float32, numpy=
array([[ -0.41144824,  0.46608728,  0.4026869 , -0.43939823,  0.08398247,
        -0.32591325, -0.39347252,  0.6138397 ,  0.21282744,  0.45471781],
        [ 0.12710994, -0.03871363,  0.39759976,  0.16387296,  0.62781316,
         0.04633373, -0.570649 , -0.53870165,  0.27773976,  0.17358983],
        [-0.32601127, -0.3041657 ,  0.3908257 ,  0.4298411 , -0.04686528,
        -0.48858804, -0.23975444,  0.4600548 , -0.3196772 ,  0.31046915],
        [-0.0657537 , -0.34420308,  0.58001167,  0.11603856,  0.16799235,
        -0.0145483 ,  0.30091405, -0.10194778,  0.08636934,  0.01607668],
        [ 0.3009435 ,  0.5368225 , -0.33597678, -0.62807584,  0.5353629 ,
        -0.22743767,  0.10951966,  0.59070975, -0.61498296, -0.55999714]],
        dtype=float32)>,
 <tf.Variable 'dense_1/bias:0' shape=(10,) dtype=float32, numpy=array([0., 0.,
```

We can see that there are variables and biases. So, we have a variable tensor that shape of 5x10 and we have 10 biases.

(Refer Slide Time: 03:57)



```
[ ] # The variables are also accessible through nice accessors
layer.kernel, layer.bias
```

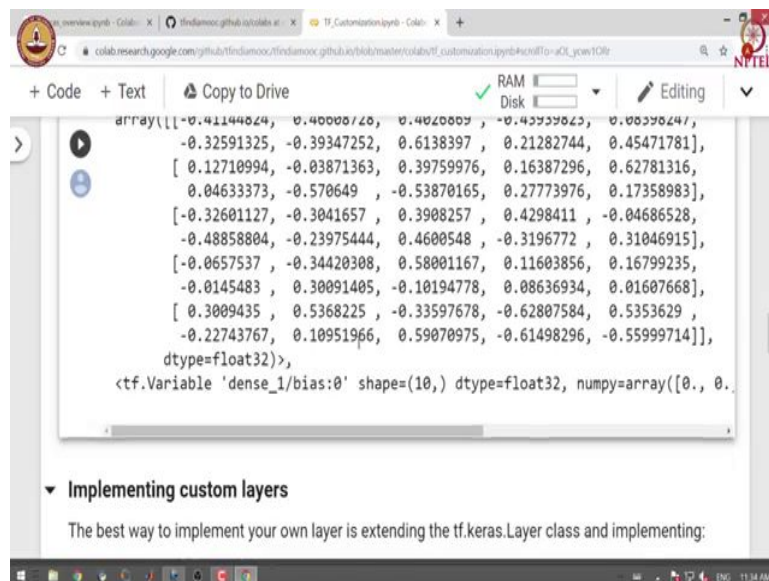
Implementing custom layers

The best way to implement your own layer is extending the `tf.keras.Layer` class and implementing:

- `__init__`, where you can do all input-independent initialization
- `build`, where you know the shapes of the input tensors and can do the rest of the initialization
- `call`, where you do the forward computation

We can also access variables separately; for example, all the variables can be assessed using `layer.kernel` and biases can be accessed with `layer.bias`.

(Refer Slide Time: 04:12)



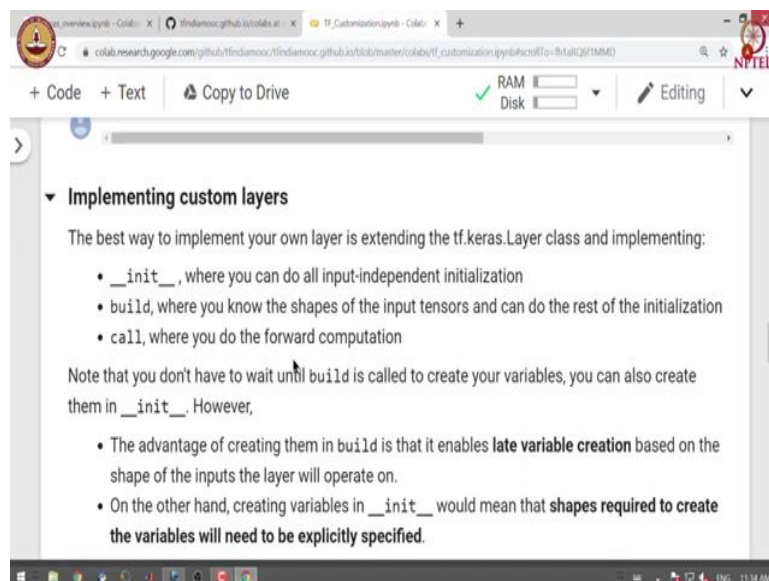
The screenshot shows a Jupyter Notebook interface with a code cell containing the following Python code:

```
array([[ -0.41144824,  0.40008720,  0.40200000, -0.43337043,  0.00330247,
        -0.32591325, -0.39347252,  0.6138397 ,  0.21282744,  0.45471781],
       [ 0.12710994, -0.03871363,  0.39759976,  0.16387296,  0.62781316,
         0.04633373, -0.570649 , -0.53870165,  0.27773976,  0.17358983],
       [-0.32601127, -0.3041657 ,  0.3908257 ,  0.4298411 , -0.04686528,
        -0.48858804, -0.23975444,  0.4600548 , -0.3196772 ,  0.31046915],
       [-0.0657537 , -0.34420308,  0.58001167,  0.11603856,  0.16799235,
        -0.0145483 ,  0.30091405, -0.10194778,  0.08636934,  0.01607668],
       [ 0.3009435 ,  0.5368225 , -0.33597678, -0.62807584,  0.5353629 ,
        -0.22743767,  0.10951966,  0.59070975, -0.61498296, -0.55999714]],
      dtype=float32)>,
<tf.Variable 'dense_1/bias:0' shape=(10,) dtype=float32, numpy=array([0., 0.,
```

Below the code cell, there is a section titled "Implementing custom layers" with the text: "The best way to implement your own layer is extending the tf.keras.Layer class and implementing:"

We can see that we get the same output. The difference is that when we use `.variables`, it gives us both kernel and bias. And we can separately ask for kernel and bias using `.kernel` and `.bias` accessors.

(Refer Slide Time: 04:38)



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

Implementing custom layers

The best way to implement your own layer is extending the `tf.keras.Layer` class and implementing:

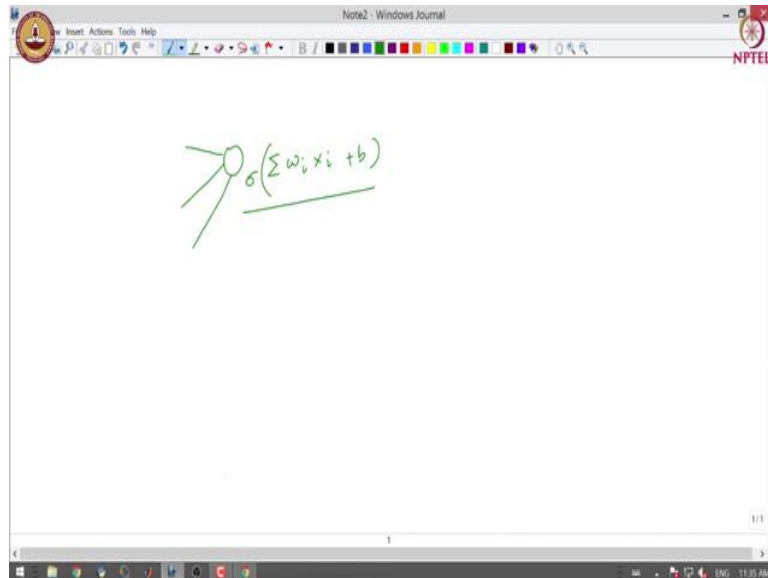
- `__init__`, where you can do all input-independent initialization
- `build`, where you know the shapes of the input tensors and can do the rest of the initialization
- `call`, where you do the forward computation

Note that you don't have to wait until `build` is called to create your variables, you can also create them in `__init__`. However,

- The advantage of creating them in `build` is that it enables **late variable creation** based on the shape of the inputs the layer will operate on.
- On the other hand, creating variables in `__init__` would mean that **shapes required to create the variables will need to be explicitly specified**.

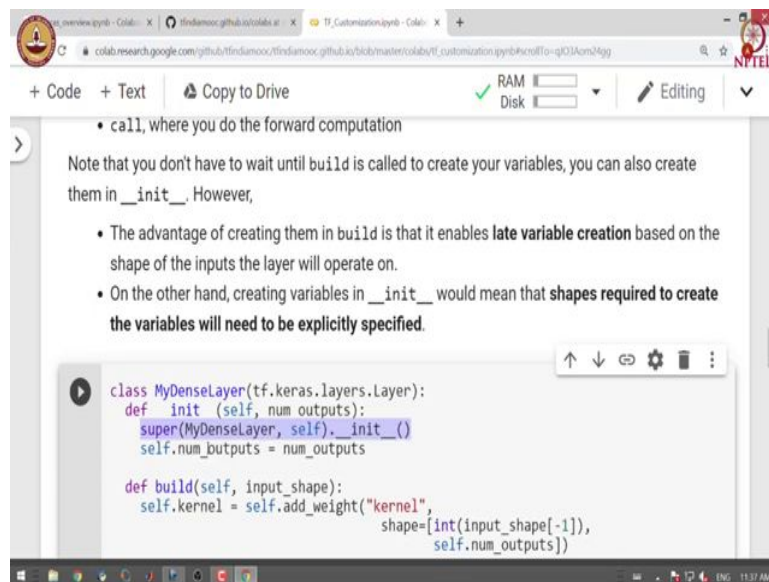
We can also implement our custom layer. In order to implement custom layer, we extend `tf.keras.layers` class and we implement the constructor. We implement `build` and a `call` function. The `call` function does the forward computations.

(Refer Slide Time: 05:19)



So, in order to give you an example, a unit in a dense layer takes input. So, essentially it does two operations; one is linear combination followed by non-linear activation. So, this is an example of a forward computation which is done in the `call` method.

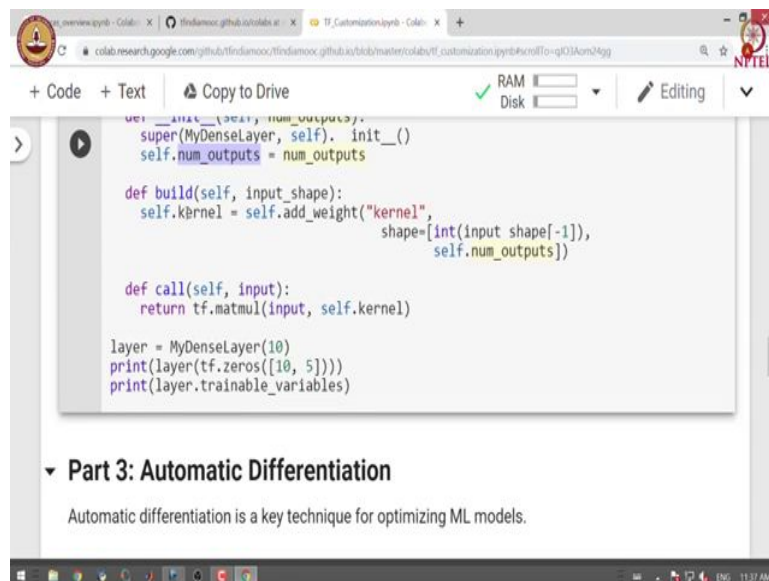
(Refer Slide Time: 05:52)



We do not have to wait until the build function is called to create variables, we can also create them in the constructor. However, the advantage of creating them in build is that it enables late variable creation based on the shape of input that layer will operate on. On the other hand creating variables in the constructor would mean that shape required to create the variable will have to be specified explicitly.

So, in this particular case we are defining MyDenseLayer which extends the layer class. And you can see that we have implemented three methods. One is the constructor; in constructor, we first call the constructor of the base class, we specified a number of outputs with a variable num_outputs.

(Refer Slide Time: 07:00)



```
def __init__(self, num_outputs):
    super(MyDenseLayer, self).__init__()
    self.num_outputs = num_outputs

def build(self, input_shape):
    self.kernel = self.add_weight("kernel",
                                   shape=[int(input_shape[-1]),
                                           self.num_outputs])

def call(self, input):
    return tf.matmul(input, self.kernel)

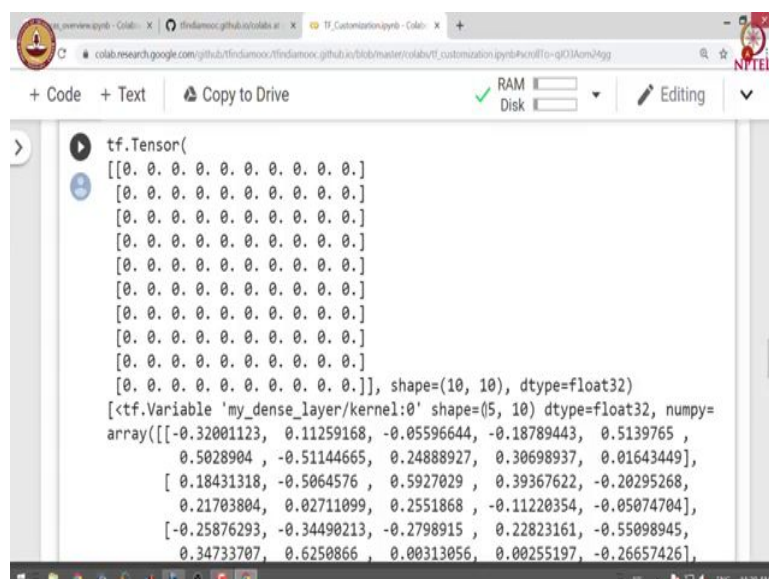
layer = MyDenseLayer(10)
print(layer(tf.zeros([10, 5])))
print(layer.trainable_variables)
```

▼ Part 3: Automatic Differentiation

Automatic differentiation is a key technique for optimizing ML models.

Then in the build stage, we specified the kernel. And in kernel, we have added weights in the kernel, which has the desired shape. And in the call function, we simply perform matrix multiplication of the input with the kernel or the weight vector. Here we define MyDenseLayer with 10 units; we pass the input shape of 10x5 to the layer. Let us print the layer and a trainable variables.

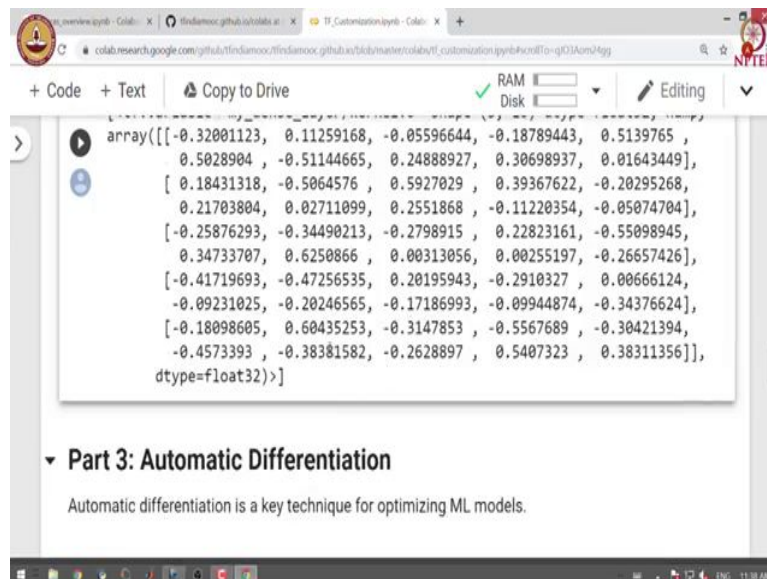
(Refer Slide Time: 07:53)



```
tf.Tensor(
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]], shape=(10, 10), dtype=float32)
[<tf.Variable 'my_dense_layer/kernel:0' shape=(5, 10) dtype=float32, numpy=
array([[-0.32001123,  0.11259168, -0.05596644, -0.18789443,  0.5139765 ,
        0.5028904 , -0.51144665,  0.24888927,  0.30698937,  0.01643449],
       [ 0.18431318, -0.5064576 ,  0.5927029 ,  0.39367622, -0.20295268,
        0.21703804,  0.02711099,  0.2551868 , -0.11220354, -0.05074704],
       [-0.25876293, -0.34490213, -0.2798915 ,  0.22823161, -0.55098945,
        0.34733707,  0.6250866 ,  0.00313056,  0.00255197, -0.26657426],
```


So, we can see that the layer was called with input shape of 10 cross 10.

(Refer Slide Time: 07:59)



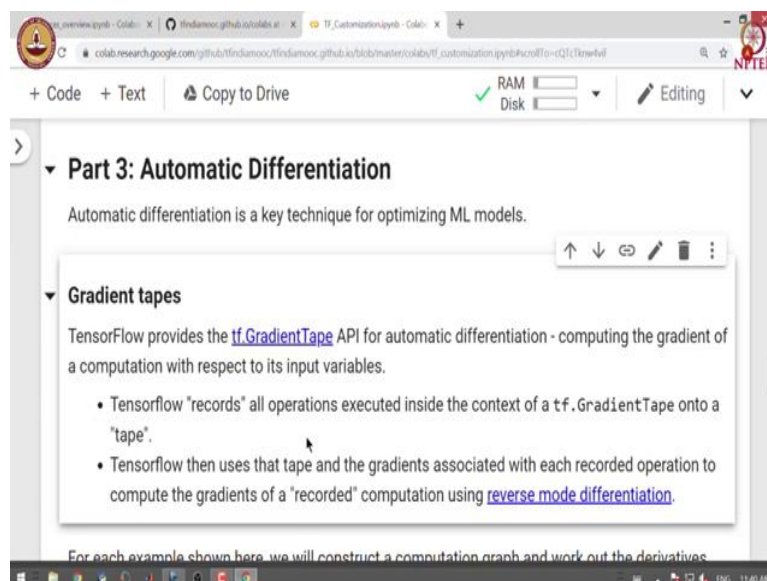
```
array([[-0.32001123,  0.11259168, -0.05596644, -0.18789443,  0.5139765 ,
        0.5028904 , -0.51144665,  0.24888927,  0.30698937,  0.01643449],
       [ 0.18431318, -0.5064576 ,  0.5927029 ,  0.39367622, -0.20295268,
        0.21703804,  0.02711099,  0.2551868 , -0.11220354, -0.05074704],
       [-0.25876293, -0.34490213, -0.2798915 ,  0.22823161, -0.55098945,
        0.34733707,  0.6250866 ,  0.00313056,  0.00255197, -0.26657426],
       [-0.41719693, -0.47256535,  0.20195943, -0.2910327 ,  0.00666124,
        -0.09231025, -0.20246565, -0.17186993, -0.09944874, -0.34376624],
       [-0.18098605,  0.60435253, -0.3147853 , -0.5567689 , -0.30421394,
        -0.4573393 , -0.38381582, -0.2628897 ,  0.5407323 ,  0.38311356]],
      dtype=float32)>]
```

Part 3: Automatic Differentiation

Automatic differentiation is a key technique for optimizing ML models.

You can see that your variable tensor by calling `.trainable_variables` on layer.

(Refer Slide Time: 08:29)



Part 3: Automatic Differentiation

Automatic differentiation is a key technique for optimizing ML models.

Gradient tapes

TensorFlow provides the [tf.GradientTape](#) API for automatic differentiation - computing the gradient of a computation with respect to its input variables.

- Tensorflow "records" all operations executed inside the context of a `tf.GradientTape` onto a "tape".
- Tensorflow then uses that tape and the gradients associated with each recorded operation to compute the gradients of a "recorded" computation using [reverse mode differentiation](#).

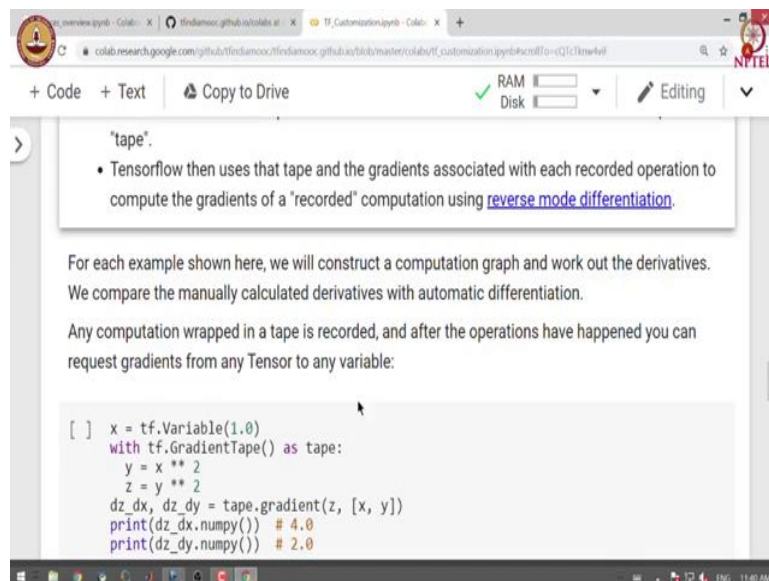
For each example shown here, we will construct a computation graph and work out the derivatives.

Having defined the custom layer; the next opportunity for customization lies in defining our own training loop. As you will remember from earlier classes; the main operation in training is to calculate the gradient and perform parameter update based on the gradient value.

Usually, these gradients are hand calculated, but tensorflow provides a way to automatically calculate these gradients based on the forward computation.

Tensorflow provides `tf.GradientTape` API for automatic differentiation. It computes the gradients of computation with respect to its own variable. Tensorflow records all the operations executed in the context of `tf.GradientTape` onto a tape.

(Refer Slide Time: 09:27)



The screenshot shows a Google Colab notebook interface. At the top, there are tabs for 'Overview', 'Code', and 'Text'. The 'Text' tab is active, displaying the following content:

"tape".

- Tensorflow then uses that tape and the gradients associated with each recorded operation to compute the gradients of a "recorded" computation using [reverse mode differentiation](#).

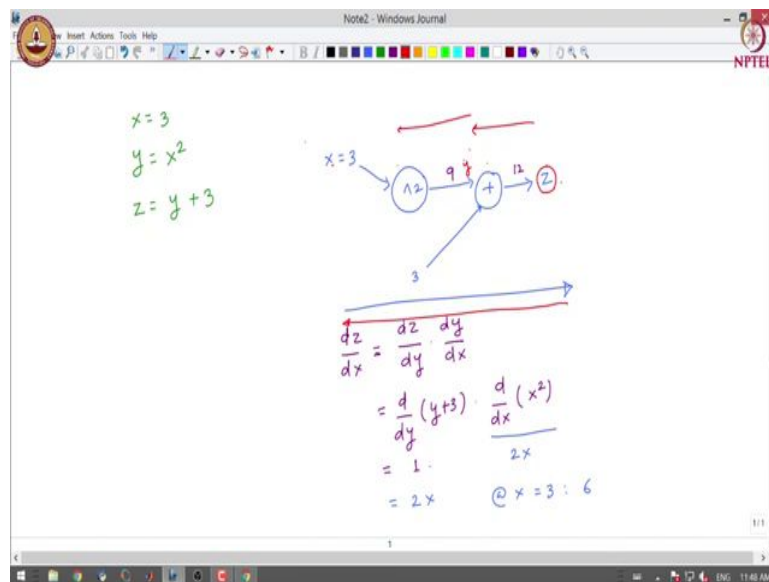
For each example shown here, we will construct a computation graph and work out the derivatives. We compare the manually calculated derivatives with automatic differentiation.

Any computation wrapped in a tape is recorded, and after the operations have happened you can request gradients from any Tensor to any variable:

```
[ ] x = tf.Variable(1.0)
    with tf.GradientTape() as tape:
        y = x ** 2
        z = y ** 2
    dz_dx, dz_dy = tape.gradient(z, [x, y])
    print(dz_dx.numpy()) # 4.0
    print(dz_dy.numpy()) # 2.0
```

Then it uses that tape and the gradients associated with each recorded operation to compute the gradients of a recorded computation using reverse mode differentiation. Let us take an example of how this particular thing operates.

(Refer Slide Time: 09:55)



So, let us say, we have $x=3$, $y=x^2$ and $z=y+3$. Let us use computation graph to represent the relationship between x , y and z .

The computation graph corresponding to the relationship between three variables x , y and z can be seen above. We perform forward computation, in the forward computation what we do is; we pass the values of the variable through this particular graph to obtain the value of z . It is concretely in this graph we set x to 3 , we raise the power of the value of x by 2 , we get the output of 9 over here. We add this 9 and 3 to get 12 at z .

Now, in machine learning we are interested in calculating the derivative of the loss that we normally see in the final step of neural network with respect to each of the input variables. So, we are modeling that particular situation with this toy example. In the context of this example we are interested in finding derivative of z with respect to x .

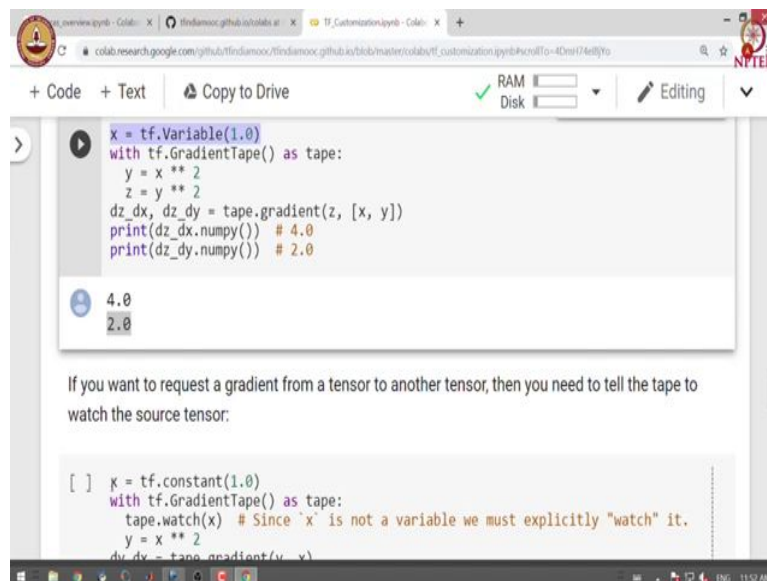
And in order to calculate this particular derivative, we use the chain rule of derivative; where we will take derivative of z with respect to y , which is known and derivative of y with respect to x . And with this chain rule of calculus, we calculate the derivative of z with respect to x . So, derivative of z with respect to x is calculated as the derivative of z with respect to y and derivative of y with respect to x .

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

So, we can now, we can see that $\frac{dz}{dy}$ is 1, because the derivative of y with respect to y is 1 and derivative of constant with respect to y is 0. From $\frac{dy}{dx}$ we get 2x. So, x equal to 3, this value this value is 6. So, this derivative computation is done in a reverse mode differentiation where, we start this variable we calculate the derivative of this with respect to y. And then we calculate the derivative of y with respect to x to obtain derivative of z with respect to x.

So, in the backward pass, we use reverse mode differentiation to calculate the derivative of z with respect to x. So, gradient tape is used to record the forward operations and then it uses the gradients associated with each recorded operations to compute the gradient of a recorded computation.

(Refer Slide Time: 15:28)



```

x = tf.Variable(1.0)
with tf.GradientTape() as tape:
    y = x ** 2
    z = y ** 2
dz_dx, dz_dy = tape.gradient(z, [x, y])
print(dz_dx.numpy()) # 4.0
print(dz_dy.numpy()) # 2.0

```

4.0
2.0

If you want to request a gradient from a tensor to another tensor, then you need to tell the tape to watch the source tensor:

```

[ ] x = tf.constant(1.0)
with tf.GradientTape() as tape:
    tape.watch(x) # Since 'x' is not a variable we must explicitly "watch" it.
    y = x ** 2
    dy_dx = tape.gradient(y, x)

```

Let us take a complete example; we define x to be a variable and in the context of gradient tape we perform two operations. First we raise the power of x by 2 and then we get y as a result and again we raise the power of y to 2 and we get z. So, here we have the following situation.

(Refer Slide Time: 15:57)

$x = 1$
 $y = x^2$
 $z = y^2$

$x=1 \rightarrow \text{circle with } x^2 \rightarrow \text{circle with } y^2 \rightarrow z$

$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$
 $= \frac{d}{dy}(y^2) \cdot \frac{d}{dx}(x^2)$
 $= 2y \cdot 2x$
 $= 2(x^2) \cdot 2x$
 $= 4x^3$
 Since $x=1$, $\frac{dz}{dx} = 4 \cdot 1^3 = 4$

$\frac{dz}{dy} = 2y = 2(x^2) = 2(1^2) = 2$

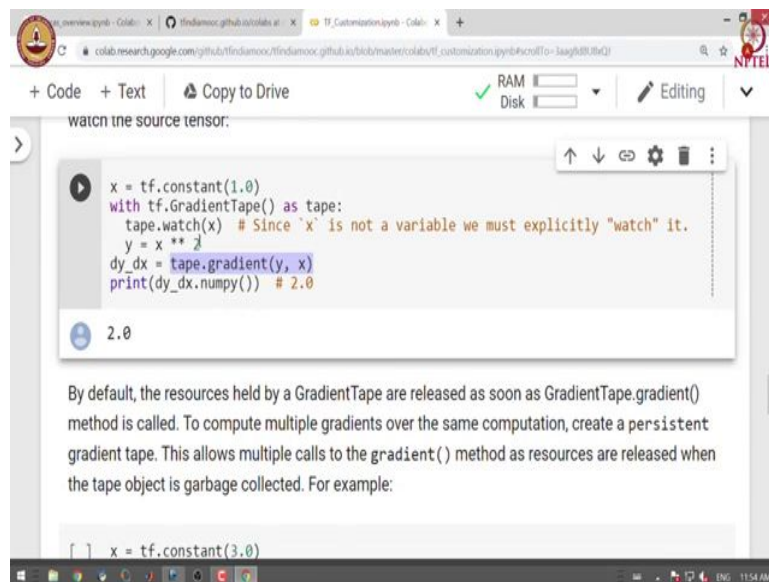
So, $x=1$, $y=x^2$ and $z=y^2$. So, we started with x equal to 1, we raise the power to 2, we get y we again apply the power operator to get the value of z is where we get y . So, this is the forward computation.

And we can calculate the derivative of the recorded computation with respect to the input variable with reverse mode differentiation. So, in this case $\frac{dz}{dx}$ is equal to $\frac{dz}{dy}$ into $\frac{dy}{dx}$, by the chain rule. So, $\frac{dz}{dy}$ is $2y$ and $\frac{dy}{dx}$ is $2x$. So, y as we know is x^2 . So, this derivative is $4x^3$.

So, you can see that, since x is equal to 1, we get $\frac{dz}{dx}$ to be 4 and $\frac{dz}{dy}$ to be 2. So, since x is equal to 1 $\frac{dz}{dx}$ is equal to $4x^3 = 4$. And $\frac{dz}{dy}$ is equal to $2y = 2x^2$ which is 2 into 1 squared is equal to 2. So, $\frac{dz}{dx}$ is 4 and $\frac{dz}{dy}$ is 2.

Let us run this to verify. We can see that $\frac{dz}{dx}$ is 4 and $\frac{dz}{dy}$ is 2. So, note that here x was defined as a variable. We can also request a gradient from a tensor to another tensor.

(Refer Slide Time: 19:12)



```
x = tf.constant(1.0)
with tf.GradientTape() as tape:
    tape.watch(x) # Since 'x' is not a variable we must explicitly "watch" it.
    y = x ** 2
dy_dx = tape.gradient(y, x)
print(dy_dx.numpy()) # 2.0
```

2.0

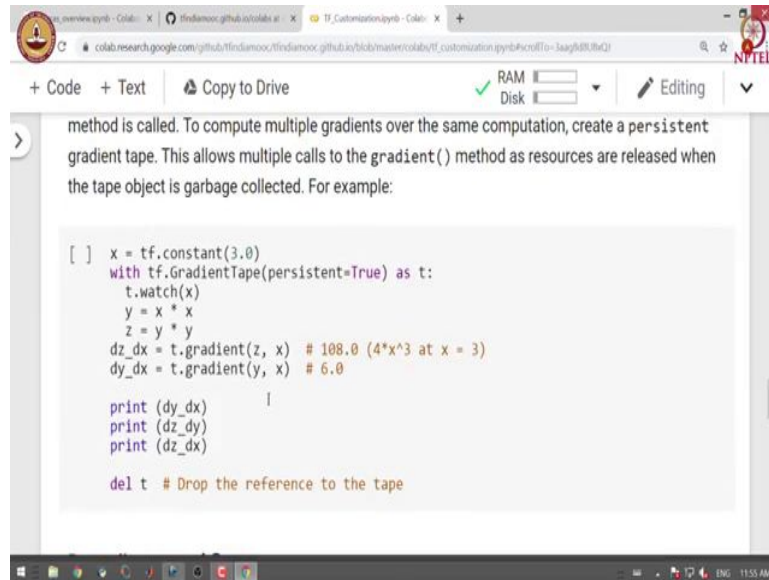
By default, the resources held by a GradientTape are released as soon as GradientTape.gradient() method is called. To compute multiple gradients over the same computation, create a persistent gradient tape. This allows multiple calls to the gradient() method as resources are released when the tape object is garbage collected. For example:

```
[ ] x = tf.constant(3.0)
```

Let us look at the example. Here x is a tensor and we raise the power of x by 2 to get y . And now we are interested in calculating derivative of y with respect to x . Note that x is a tensor and y is also a tensor.

Now, since you are interested in calculating derivative of y with respect to x , which is a tensor, we add `tape.watch` and add tensor x to the watch list. By doing this we are able to calculate the derivative of y which respect to x with the gradient tape. And we perform both this operation in the context of gradient tape and then calculate the gradient using `tape.gradient` method. Let us run it and check it out. And you know that y is x square, so derivative $\frac{dy}{dx}$ will be $2x$ and x is equal to 1 that is why the value that we see here is 2.

(Refer Slide Time: 20:31)



The screenshot shows a Jupyter Notebook interface with a code cell. The code defines a persistent gradient tape and computes gradients for a simple function. The output of the code is visible in the cell.

```
method is called. To compute multiple gradients over the same computation, create a persistent
gradient tape. This allows multiple calls to the gradient() method as resources are released when
the tape object is garbage collected. For example:

[ ] x = tf.constant(3.0)
    with tf.GradientTape(persistent=True) as t:
        t.watch(x)
        y = x * x
        z = y * y
        dz_dx = t.gradient(z, x) # 108.0 (4*x^3 at x = 3)
        dy_dx = t.gradient(y, x) # 6.0

    print(dy_dx)
    print(dz_dy)
    print(dz_dx)

    del t # Drop the reference to the tape
```

So, by default the resources held by gradient tape are released as soon as the gradient tape.gradient method is called. To compute multiple gradients over the same computation, we create a persistent gradient tape. We create a persistent gradient tape, this allows multiple calls to the gradient method as resources are released when the tape object is garbage collected. Let us look at the example of using a persistent tape.

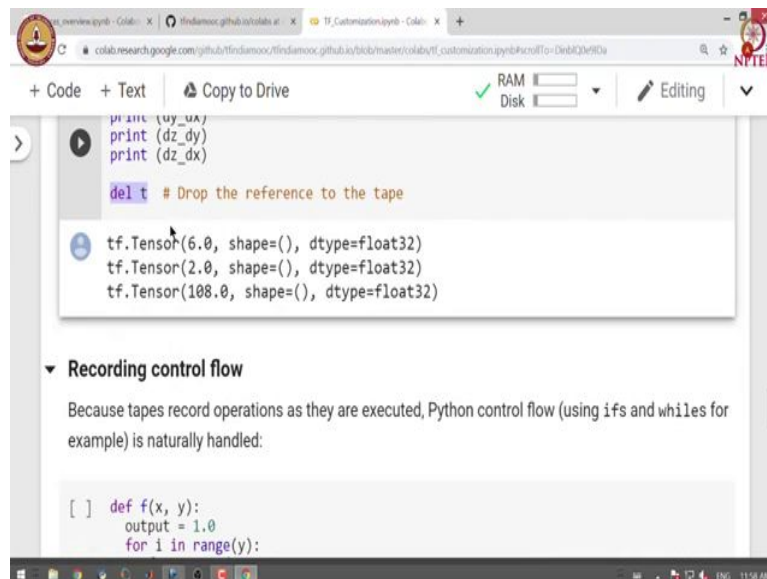
So, we simply add persistent equal to true to gradient tape method and then perform the forward computation in the context of this persistent gradient tape. Since, we are interested in calculating derivative of z and y with respect to x, we first add the tensor x to the watch list and then perform the remaining forward operations. So, here we obtained y by multiplying x to itself or in other words, by squaring the value of x and z is obtained by squaring the value of y.

Let us look at what is the gradient at x is equal to 3. So, we know that the gradient of z with respect to x is $4x^3$. So, that is why at value of $x=3$, $x^3=27$, and 27 time 4 gets us 108 as the value of gradient.

We know that the gradient of y with respect to x is essentially 2x that is why at x equal to 3, we get the value of 6. Let us print all the three gradients, which is the derivative of y with

respect to x , derivative of z with respect to y and derivative of z with respect to x . And finally, we delete the reference to the tape using `del` command.

(Refer Slide Time: 23:28)



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

```
print(dy_dx)
print(dz_dy)
print(dz_dx)

del t # Drop the reference to the tape
```

Below the code cell, the output is displayed:

```
tf.Tensor(6.0, shape=(), dtype=float32)
tf.Tensor(2.0, shape=(), dtype=float32)
tf.Tensor(108.0, shape=(), dtype=float32)
```

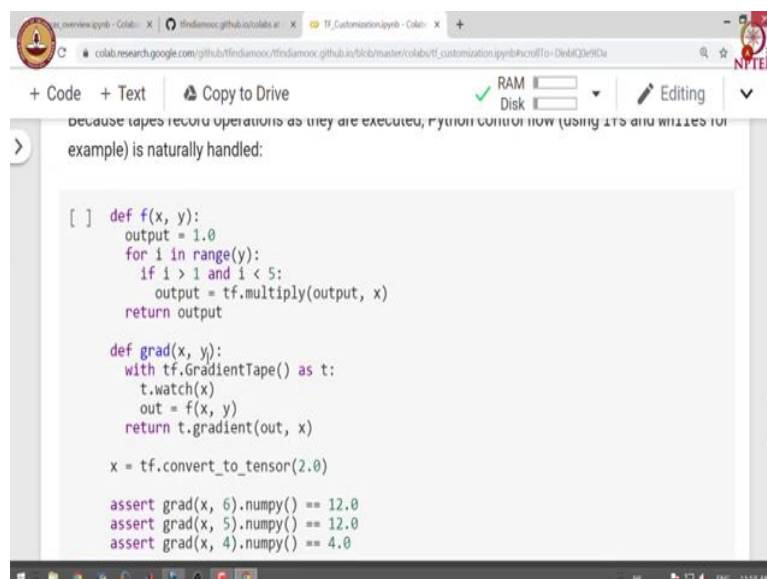
Below the output, there is a section titled "Recording control flow" with the following text:

Because tapes record operations as they are executed, Python control flow (using ifs and whiles for example) is naturally handled:

```
[ ] def f(x, y):
    output = 1.0
    for i in range(y):
```

So, we obtained results as per our expectations.

(Refer Slide Time: 23:35)



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

```
def f(x, y):
    output = 1.0
    for i in range(y):
        if i > 1 and i < 5:
            output = tf.multiply(output, x)
    return output

def grad(x, y):
    with tf.GradientTape() as t:
        t.watch(x)
        out = f(x, y)
    return t.gradient(out, x)

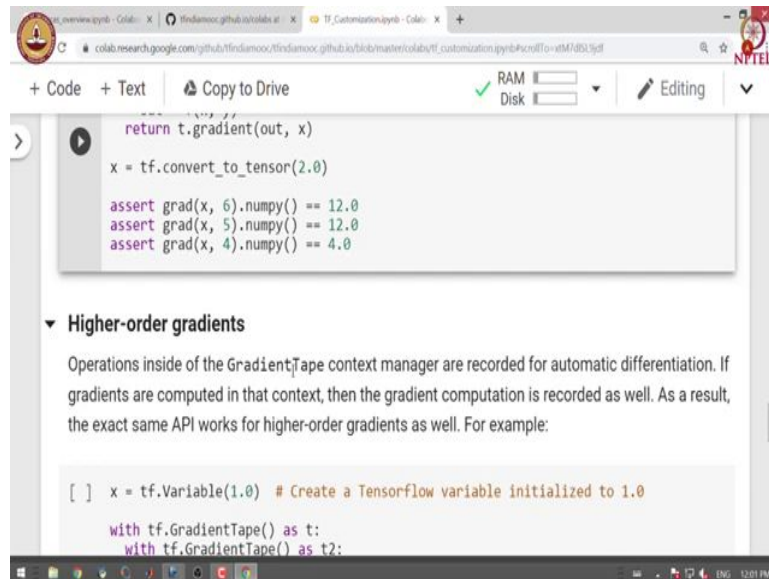
x = tf.convert_to_tensor(2.0)

assert grad(x, 6).numpy() == 12.0
assert grad(x, 5).numpy() == 12.0
assert grad(x, 4).numpy() == 4.0
```

We can also record forward operations even in the presence of the python control statements or loops. So, this is an example where we first define a tensor containing value 2. And then

we define a function called grad; the grad function essentially defines a couple of forward computations in the context of gradient tape and the computation that we carry out is defined in function f. f runs the computation in a loop that repeats y times and if the value of i is greater than 1 and less than 5. It performs the multiplication operation. It multiplies the output with x.

(Refer Slide Time: 25:12)



```
return t.gradient(out, x)

x = tf.convert_to_tensor(2.0)

assert grad(x, 6).numpy() == 12.0
assert grad(x, 5).numpy() == 12.0
assert grad(x, 4).numpy() == 4.0
```

Higher-order gradients

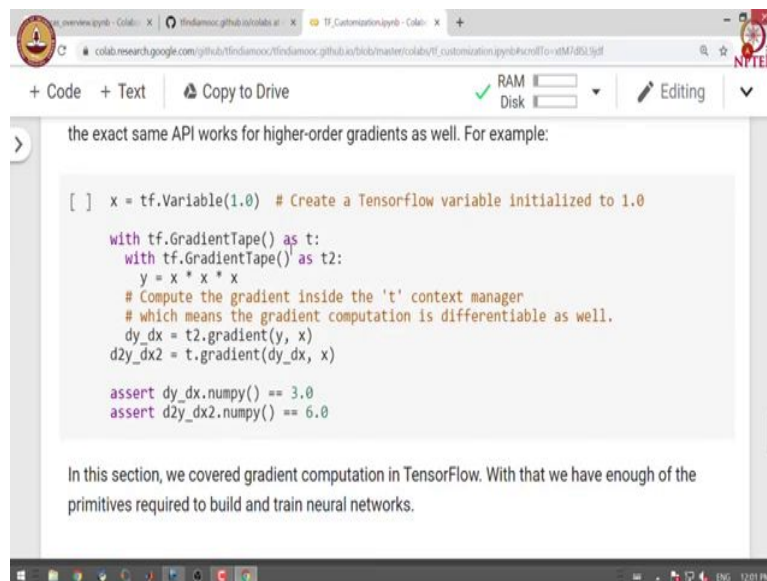
Operations inside of the GradientTape context manager are recorded for automatic differentiation. If gradients are computed in that context, then the gradient computation is recorded as well. As a result, the exact same API works for higher-order gradients as well. For example:

```
[ ] x = tf.Variable(1.0) # Create a Tensorflow variable initialized to 1.0

with tf.GradientTape() as t:
    with tf.GradientTape() as t2:
```

The gradient tape can also be used in the presence of loops or control statements. We can also use operations inside gradient tape to calculate higher order gradients.

(Refer Slide Time: 25:24)



The screenshot shows a Google Colab notebook interface. The top bar includes tabs for 'Overview.ipynb', 'tf.nn.conv2d.ipynb', and 'TF Customization.ipynb'. The main code cell contains the following Python code:

```
[ ] x = tf.Variable(1.0) # Create a Tensorflow variable initialized to 1.0

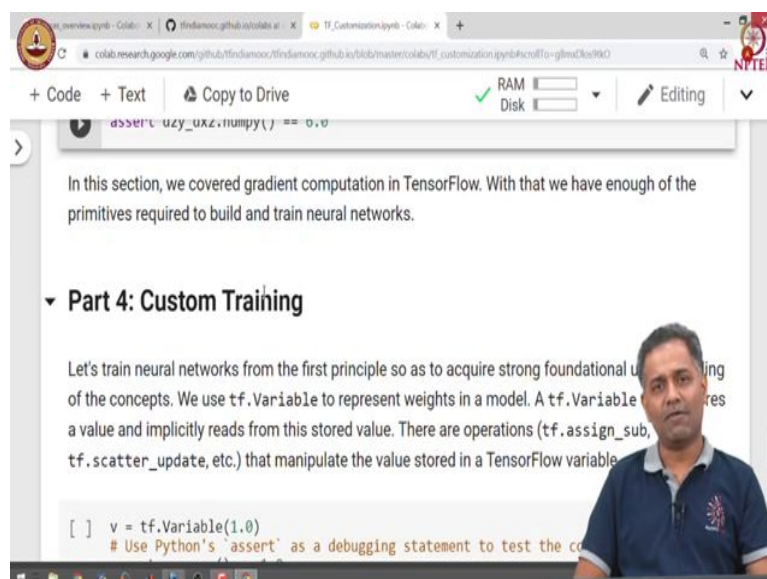
with tf.GradientTape() as t:
    with tf.GradientTape() as t2:
        y = x * x * x
        # Compute the gradient inside the 't' context manager
        # which means the gradient computation is differentiable as well.
        dy_dx = t2.gradient(y, x)
        d2y_dx2 = t.gradient(dy_dx, x)

    assert dy_dx.numpy() == 3.0
    assert d2y_dx2.numpy() == 6.0
```

Below the code, a text block states: "In this section, we covered gradient computation in TensorFlow. With that we have enough of the primitives required to build and train neural networks."

In this case, we are defining a nested gradient tape. So, there is an outer gradient tape in the context of which there is another gradient tape and in the context of both these gradient tapes; we are performing x^3 operation to obtain the value of y . In the context of inner gradient tape, we can get derivative of y with respect to x . And in the context of outer gradient tape, we can calculate the second derivative of y with respect to x .

(Refer Slide Time: 26:08)



The screenshot shows a Google Colab notebook interface. The top bar includes tabs for 'Overview.ipynb', 'tf.nn.conv2d.ipynb', and 'TF Customization.ipynb'. The main code cell contains the following Python code:

```
[ ] v = tf.Variable(1.0)
# Use Python's 'assert' as a debugging statement to test the co
```

Below the code, a text block states: "In this section, we covered gradient computation in TensorFlow. With that we have enough of the primitives required to build and train neural networks."

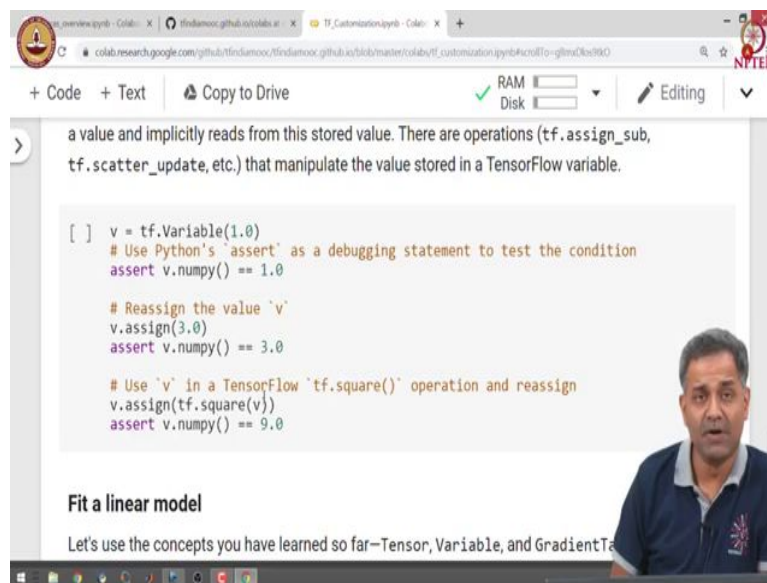
Part 4: Custom Training

Let's train neural networks from the first principle so as to acquire strong foundational understanding of the concepts. We use `tf.Variable` to represent weights in a model. A `tf.Variable` stores a value and implicitly reads from this stored value. There are operations (`tf.assign_sub`, `tf.scatter_update`, etc.) that manipulate the value stored in a TensorFlow variable.

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

So far, we studied how to use specific devices to carry out tensorflow operations. Then we studied how to write custom layers and we also studied how to perform automatic differentiation to obtain gradients of loss functions with respect to input variables through automatic differentiation. With these three concepts, we have some tools available to us for writing custom machine learning algorithms. Let us use the concepts that we learnt in automatic differentiation to write our custom training loop.

(Refer Slide Time: 27:06)



```
[ ] v = tf.Variable(1.0)
# Use Python's 'assert' as a debugging statement to test the condition
assert v.numpy() == 1.0

# Reassign the value 'v'
v.assign(3.0)
assert v.numpy() == 3.0

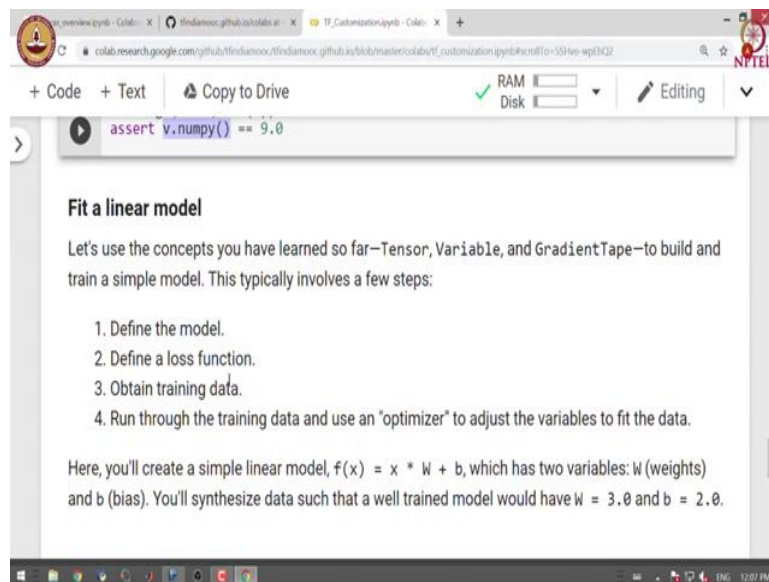
# Use 'v' in a TensorFlow 'tf.square()' operation and reassign
v.assign(tf.square(v))
assert v.numpy() == 9.0
```

Fit a linear model

Let's use the concepts you have learned so far—Tensor, Variable, and GradientTape

Here, we will train tensorflow model from first principles. So, we use variables for storing weights of tensorflow model and then we use functions like assign to assign values to the variable. In this case, we are assigning the value of 3 to variable v and here we are assigning the value of square of v to v itself. So, let us run this particular code, we can see that the value of v was initialized to 1 here, then we assigned a value of 3 and here we assigned a value of 9 to v. And note that we are using v.numpy function to obtain the value present in the tensor.

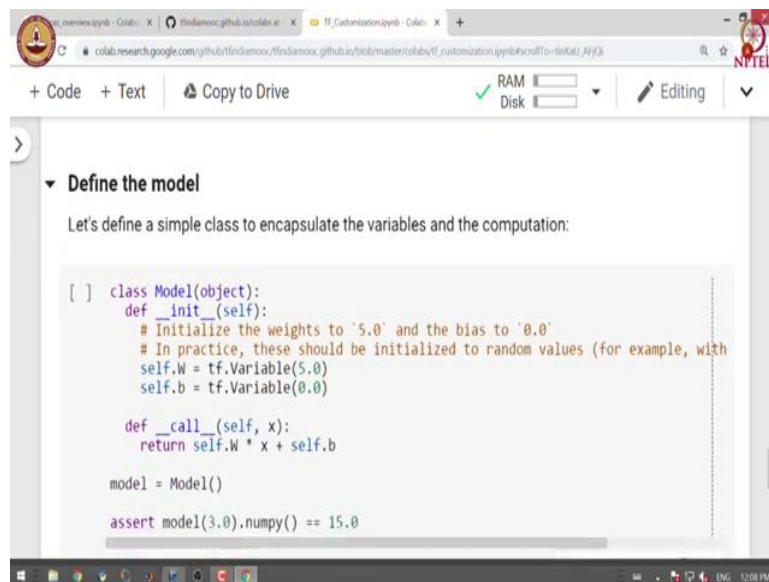
(Refer Slide Time: 28:06)



What we will do is, we will define a linear model using the concepts that we have learnt so far.

So, there are four different steps; we have to define the model, then loss function, then obtain the training data and train the model. And how do you train the model? We use a specific optimization algorithm for training the model. So, here in linear regression, we have a model which is a linear combination of weights and the input along with a bias term to it. So, if x into W plus b it has got two variables - W and bias.

(Refer Slide Time: 29:19)



The screenshot shows a Google Colab notebook interface. At the top, there are tabs for 'TF_Customization.ipynb' and 'TF_Customization.ipynb'. The main content area is titled 'Define the model' and contains the following text: 'Let's define a simple class to encapsulate the variables and the computation:'. Below this, there is a code block with the following Python code:

```
[ ] class Model(object):
    def __init__(self):
        # Initialize the weights to '5.0' and the bias to '0.0'
        # In practice, these should be initialized to random values (for example, with
        self.W = tf.Variable(5.0)
        self.b = tf.Variable(0.0)

    def __call__(self, x):
        return self.W * x + self.b

model = Model()

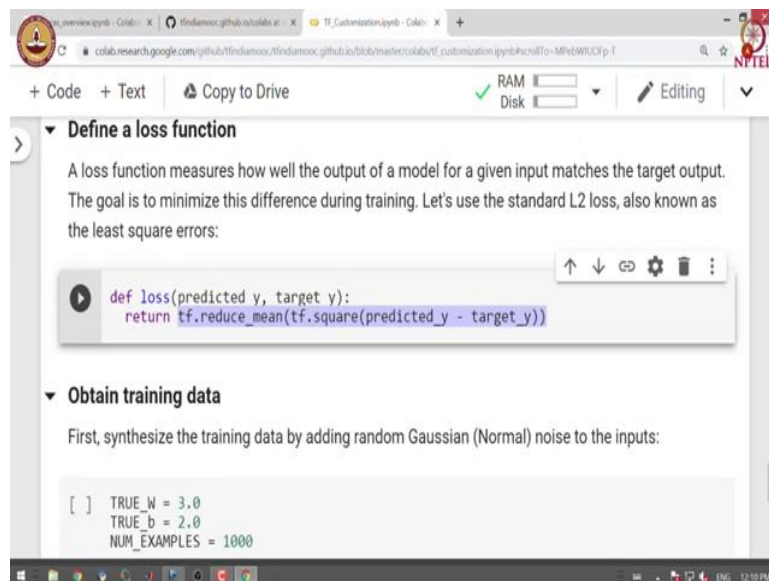
assert model(3.0).numpy() == 15.0
```

So, what we will do is; we will first obtain synthetic data with $W=2$ and $b=2$. We will define a simple class to encapsulate the variables in the computations.

So, in the constructor, we define variables W and b and we have set them to 5 and 0, in real life or in practice, we randomly initialize these values. But for the sake of simplicity in this example, we have set this variables to some fixed numbers. And in the call method we are performing the forward computation where we are multiplying the input by the weights and adding the bias term to it.

So, we have model over here. So, for value 3 we get value for input 3, we get output of 15. We can see it above, the value of W is 5, x is 3. So, 5 into 3 is 15 and bias is 0. So, 15 plus 0 is 15, that is how we get the value of 15 when we pass 3 as the input to the model.

(Refer Slide Time: 31:01)

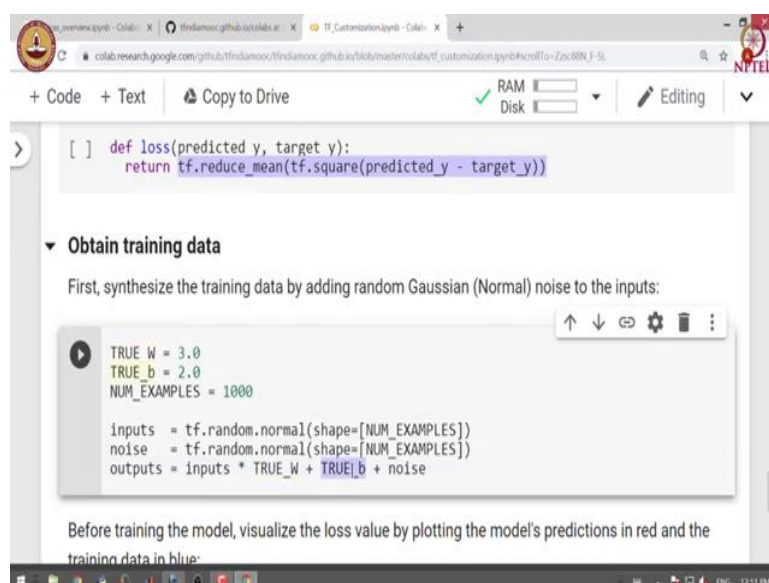


The screenshot shows a Jupyter Notebook interface with a browser window at the top. The notebook has two sections: 'Define a loss function' and 'Obtain training data'. The 'Define a loss function' section contains a text block explaining that a loss function measures how well the output of a model matches the target output, and the goal is to minimize this difference. Below the text is a code cell with the following Python code:

```
def loss(predicted y, target y):  
    return tf.reduce_mean(tf.square(predicted_y - target_y))
```

The next task is to define a loss function. Here we use standard L2 loss or a least square error. And the way we define the least square error is we calculate the square of the difference between the predicted value and the actual value. And we sum up this difference across all the points in the training data. And this is how we define our loss function.

(Refer Slide Time: 31:35)



The screenshot shows a Jupyter Notebook interface with a browser window at the top. The notebook has two sections: 'Define a loss function' and 'Obtain training data'. The 'Obtain training data' section contains a text block explaining that the training data is synthesized by adding random Gaussian (Normal) noise to the inputs. Below the text is a code cell with the following Python code:

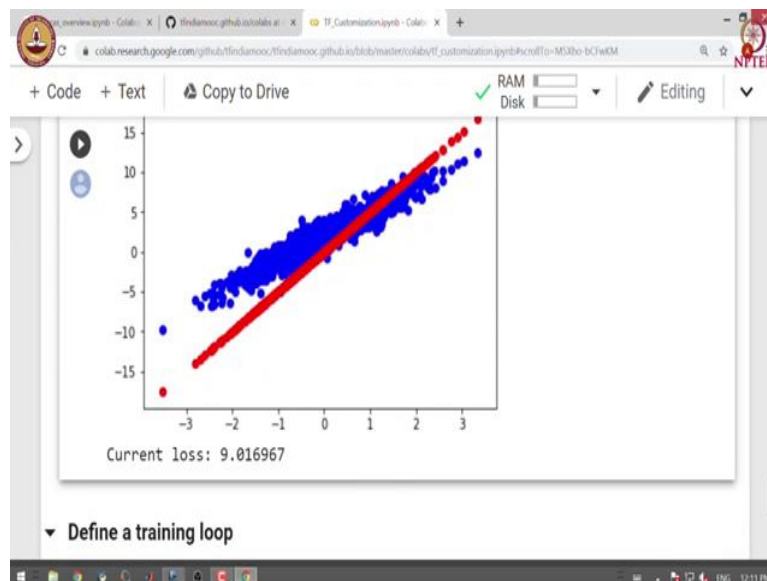
```
[ ] def loss(predicted y, target y):  
    return tf.reduce_mean(tf.square(predicted_y - target_y))  
  
▼ Obtain training data  
First, synthesize the training data by adding random Gaussian (Normal) noise to the inputs:  
  
TRUE_W = 3.0  
TRUE_b = 2.0  
NUM_EXAMPLES = 1000  
  
inputs = tf.random.normal(shape=[NUM_EXAMPLES])  
noise = tf.random.normal(shape=[NUM_EXAMPLES])  
outputs = inputs * TRUE_W + TRUE_b + noise
```

The next step is to obtain the training data; here we synthetically generate our training data by setting W to 3 and b to 2 and we generate 1000 examples by adding some noise to the

regression calculation. So, here we first define input, which is drawn from a normal distribution, and then we have a noise which is again drawn randomly from a normal distribution. And we obtain output by multiplying input by the true weight and add true bias to it, that gives us output.

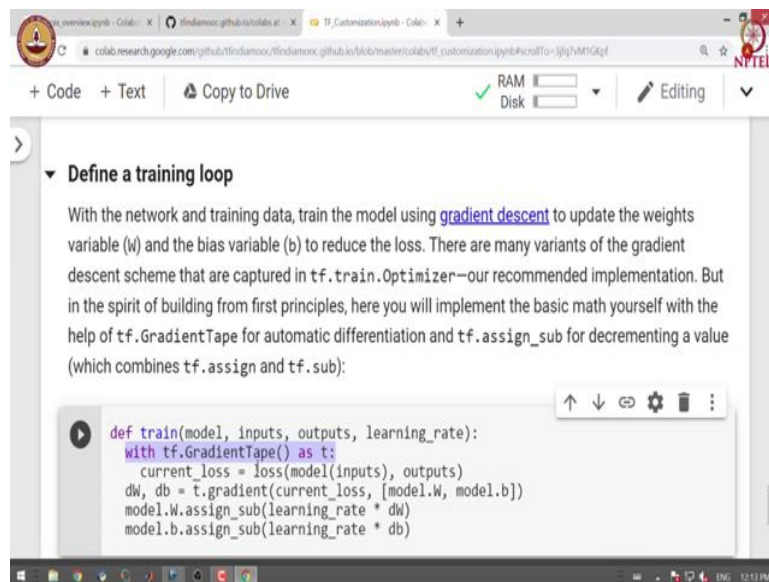
So, let us run these steps, let us define the model; let us define the loss, let us generate the training data and let us visualize the training data before building the model.

(Refer Slide Time: 32:54)



So, you can see that for the chosen parameter values, we have loss of 9.01 and you can see that, the points in the training that we generated are in blue whereas the red line or the points represented with red lines are the predicted points. Having obtained the training data, the next step is to train the model itself.

(Refer Slide Time: 33:38)



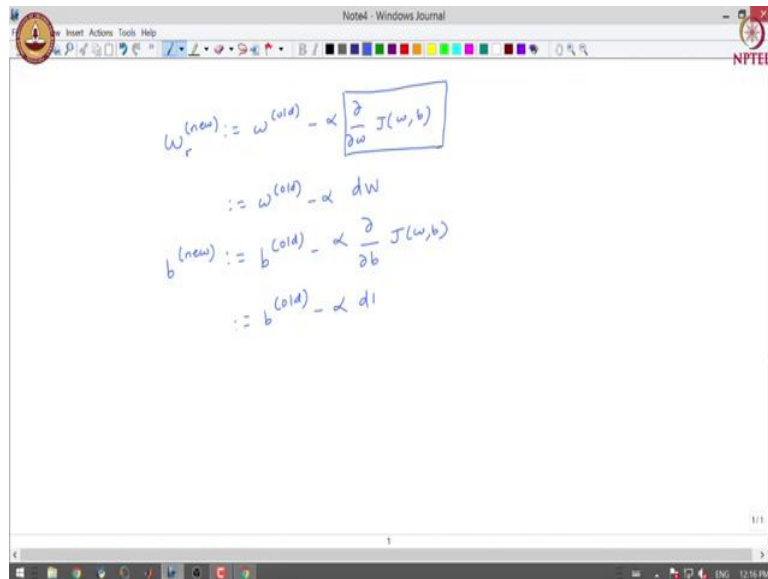
The screenshot shows a Google Colab notebook interface. At the top, there are tabs for 'TF Customization.ipynb' and 'TF Customization.ipynb'. The main content area is titled 'Define a training loop'. Below the title, there is a paragraph explaining the goal: to train a model using gradient descent to update weights (W) and bias (b) to reduce loss. It mentions using `tf.GradientTape` for automatic differentiation and `tf.assign_sub` for decrementing values. Below the text, there is a code block with the following Python code:

```
def train(model, inputs, outputs, learning_rate):  
    with tf.GradientTape() as t:  
        current_loss = loss(model(inputs), outputs)  
        dW, db = t.gradient(current_loss, [model.W, model.b])  
        model.W.assign_sub(learning_rate * dW)  
        model.b.assign_sub(learning_rate * db)
```

Let us define our custom training rule. We are going to use gradient descent to update the weights. Normally in real life applications or examples that we have seen so far, we use one of the optimizers from `tf.train.optimizer`. Gradient descent itself is available in the standard keras package, but here we want to train the model from first principles.

So, we will be using gradient tape to calculate the derivative of the loss function with respect to the input variables, let us see how do we do that. So, we calculate the loss in the context of a gradient tape. So, since we calculate loss in the context of the gradient tape, it records all the operation in the forward computation and when we call a gradient method on the tape, it gives us the gradients. For example, here we have gradients of loss with respect to W and b. Having calculated gradient you must remember to update value of W.

(Refer Slide Time: 35:02)



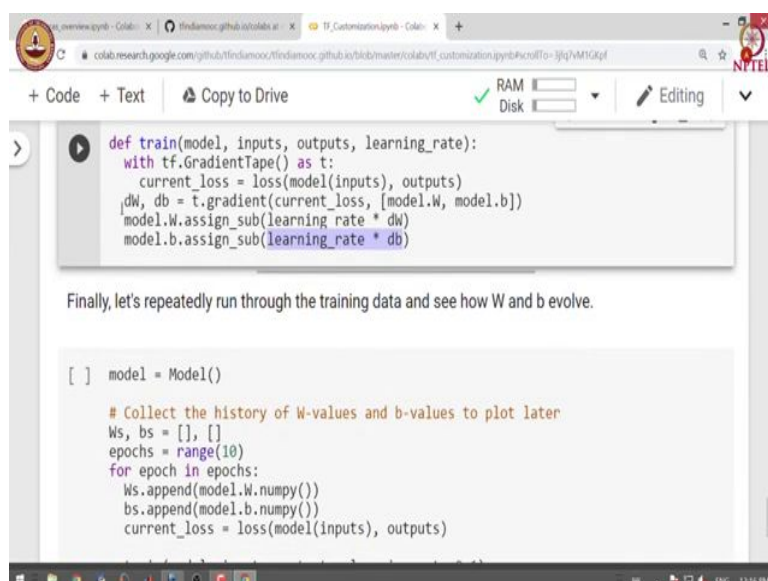
The image shows a handwritten note in a digital notebook. The equations are:

$$w_r^{(new)} := w_r^{(old)} - \alpha \frac{\partial J(w, b)}{\partial w}$$
$$:= w_r^{(old)} - \alpha dw$$
$$b^{(new)} := b^{(old)} - \alpha \frac{\partial J(w, b)}{\partial b}$$
$$:= b^{(old)} - \alpha db$$

$W^{(new)}$ is said to $W^{(old)}$ minus learning rate times the gradient of the loss function with respect to the variables. So, the gradient part, you get from the gradient tape. We call the gradient method on the tape to obtain it. We do the same for b .

So, we assign $W^{(new)}$ by subtracting from W , the learning rate into the gradient, and we obtain $b^{(new)}$ by subtracting from b , the learning rate into the gradient with respect to b .

(Refer Slide Time: 37:03)



The image shows a Python code snippet in a Jupyter notebook. The code defines a `train` function and a training loop.

```
def train(model, inputs, outputs, learning_rate):  
    with tf.GradientTape() as t:  
        current_loss = loss(model(inputs), outputs)  
        dw, db = t.gradient(current_loss, [model.W, model.b])  
        model.W.assign_sub(learning_rate * dw)  
        model.b.assign_sub(learning_rate * db)
```

Finally, let's repeatedly run through the training data and see how W and b evolve.

```
[ ] model = Model()  
  
# Collect the history of W-values and b-values to plot later  
ws, bs = [], []  
epochs = range(10)  
for epoch in epochs:  
    ws.append(model.W.numpy())  
    bs.append(model.b.numpy())  
    current_loss = loss(model(inputs), outputs)
```

This is how we define a training loop, this is how we define the basic training operation; we have to run this function repeatedly.

(Refer Slide Time: 37:15)

```
model = Model()

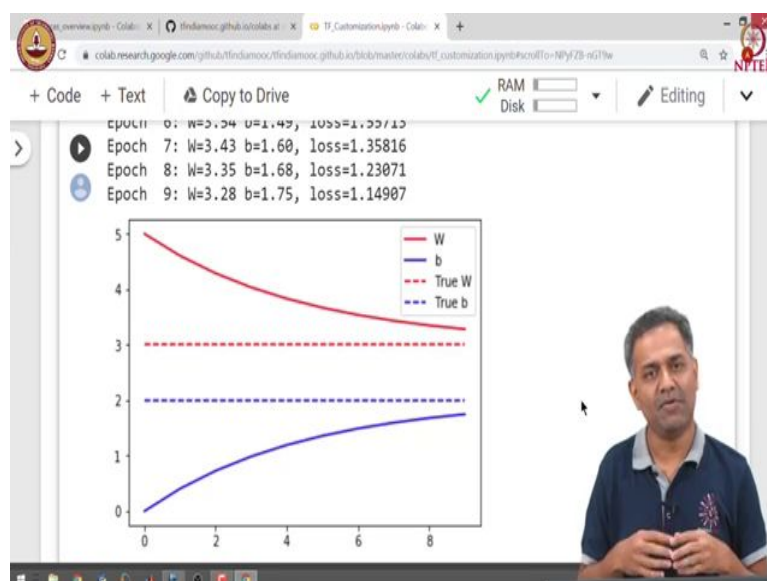
# Collect the history of W-values and b-values to plot later
Ws, bs = [], []
epochs = range(10)
for epoch in epochs:
    Ws.append(model.W.numpy())
    bs.append(model.b.numpy())
    current_loss = loss(model(inputs), outputs)

    train(model, inputs, outputs, learning_rate=0.1)
    print('Epoch %2d: W=%1.2f b=%1.2f, loss=%2.5f' %
          (epoch, Ws[-1], bs[-1], current_loss))

# Let's plot it all
plt.plot(epochs, Ws, 'r',
         epochs, bs, 'b')
plt.plot([True_W] * len(epochs), 'r--',
         [True_b] * len(epochs), 'b--')
plt.legend(['W', 'b', 'True W', 'True b'])
plt.show()
```

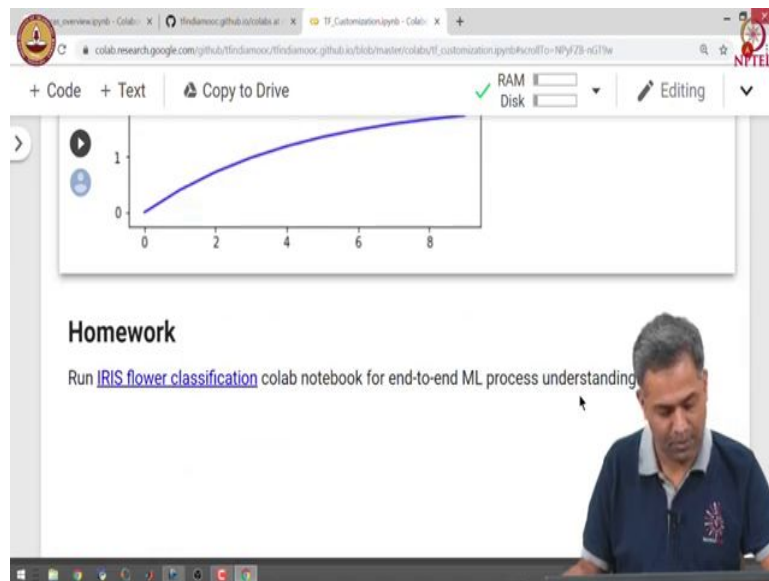
We have output the value of W, b and value of current loss. Finally, we also plot how W and b changed and also plot the true value of W and b. Let us run this.

(Refer Slide Time: 37:45)



We started W at 5 and b at 0 and as we get to 10 epochs, both W and b are getting closer to the actual value of W and b . Here we define our own model, we also defined our own training loop and then we also implemented the gradient calculation using gradient tape.

(Refer Slide Time: 38:23)



As a next step I would strongly encourage you to go through the iris flower classification notebook for end to end ML process. So, in this session, we learn a number of concepts that will help us in customizing the tensorflow functionality. We started with how to force operations on accelerated devices, how to write custom layers, how to perform automatic differentiation using gradient tapes, and we use the concepts learnt in automatic differentiation to define our own custom training.