

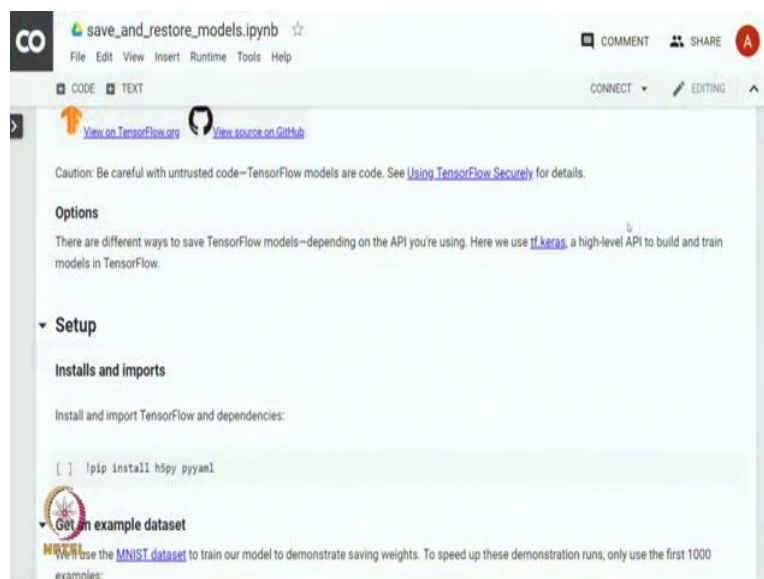
**Practical Machine Learning with TensorFlow**  
**Dr. Ashish Tendulkar**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Madras**

**Lecture - 19**  
**Save and Restore Models**

In the past few modules, we have been building Machine Learning models using TensorFlow APIs. There are situations where the model trains for a long time and we would like to store the intermediate steps of the model so that we can assess how it is performing on the test data or safeguard against unforeseen situations due to which the training loop may not complete. The model can resume training where it left off and avoid long training times after restoring the weights.

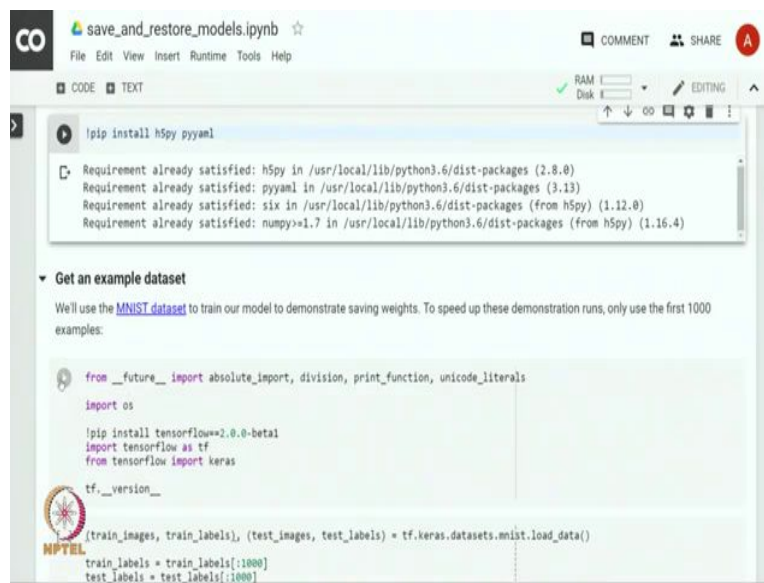
Saving the model also helps us share our work with others so that they can recreate it. When publishing research models and technique most machine learning practitioners share code to create a model and trained weights or parameters of the model. Sharing this data helps others understand how the model works and try themselves with the new data. In this module, we will learn how to store the model during or after the training.

(Refer Slide Time: 01:34)



I would like to caution you against using any untrusted code because TensorFlow models are code at the end of the day. Hence you should be careful and ascertain the origin of the code before using any untrusted code. There are different ways to save TensorFlow models depending on the API that you are using. Here we use tf.keras which is a high level API for building and training the model. Let us begin by importing TensorFlow and other dependencies.

(Refer Slide Time: 02:08)



```
!pip install h5py pyyaml

Requirement already satisfied: h5py in /usr/local/lib/python3.6/dist-packages (2.8.0)
Requirement already satisfied: pyyaml in /usr/local/lib/python3.6/dist-packages (3.13)
Requirement already satisfied: six in /usr/local/lib/python3.6/dist-packages (from h5py) (1.12.0)
Requirement already satisfied: numpy>=1.7 in /usr/local/lib/python3.6/dist-packages (from h5py) (1.16.4)

▼ Get an example dataset
We'll use the MNIST dataset to train our model to demonstrate saving weights. To speed up these demonstration runs, only use the first 1000 examples:

from __future__ import absolute_import, division, print_function, unicode_literals
import os

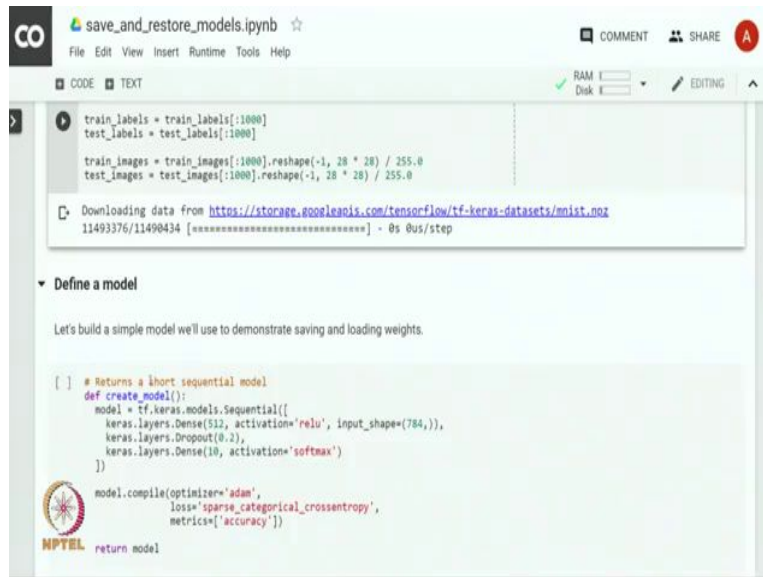
!pip install tensorflow==2.0.0-beta1
import tensorflow as tf
from tensorflow import keras

tf.__version__

(train_images, train_labels), (test_images, test_labels) = tf.keras.datasets.mnist.load_data()
train_labels = train_labels[:1000]
test_labels = test_labels[:1000]
```

Let us install TensorFlow 2.0 and make sure that the right version is installed. We also import OS package because we want to write and read files to the disk. Let us load a MNIST dataset and take 1000 examples each from training and test tensors so that are modelled can run faster and will be able to demonstrate the same and restore functionality.

(Refer Slide Time: 02:41)



```
train_labels = train_labels[:1000]
test_labels = test_labels[:1000]

train_images = train_images[:1000].reshape(-1, 28 * 28) / 255.0
test_images = test_images[:1000].reshape(-1, 28 * 28) / 255.0
```

Downloading data from <https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz>  
11493376/11490434 [=====] - 0s 0us/step

**Define a model**

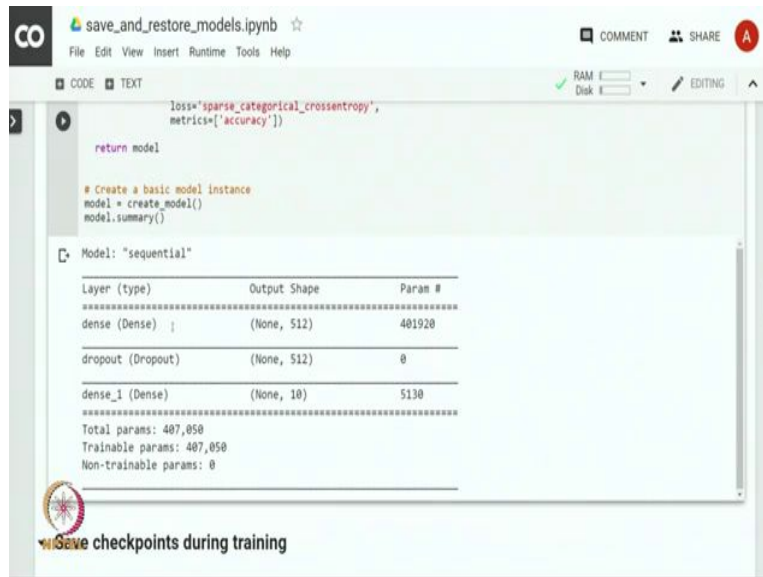
Let's build a simple model we'll use to demonstrate saving and loading weights.

```
[ ] # Returns a short sequential model
def create_model():
    model = tf.keras.models.Sequential([
        keras.layers.Dense(512, activation='relu', input_shape=(784,)),
        keras.layers.Dropout(0.2),
        keras.layers.Dense(10, activation='softmax')
    ])
    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    return model
```

Let us define the model in a python function so that we can call this function for creating the model before and after saving and for restoration purposes. So, we define a simple neural network model which has got a single hidden layer with 512 units, we use relu as an activation function and input to this particular hidden layer are 784 values. So, these 784 values come from 28 cross 28 image of digit that is stored in a MNIST dataset.

In addition to that we use a dropout regularization with a dropout rate of 0.2. And finally, we have a dense layer with 10 units as an output layer that uses Softmax as an activation. We want to output one of the 10 digits as a desired output, we use Adam as an optimizer and sparse categorical cross entropy loss as we are interested in getting integers as an output and we will track accuracy as a metric. Let us create a model and examine the model through `model.summary()` method.

(Refer Slide Time: 04:04)



The screenshot shows a Jupyter Notebook interface with a file named 'save\_and\_restore\_models.ipynb'. The code cell contains the following Python code:

```
loss='sparse_categorical_crossentropy',  
metrics=['accuracy'])  
  
return model  
  
# Create a basic model instance  
model = create_model()  
model.summary()
```

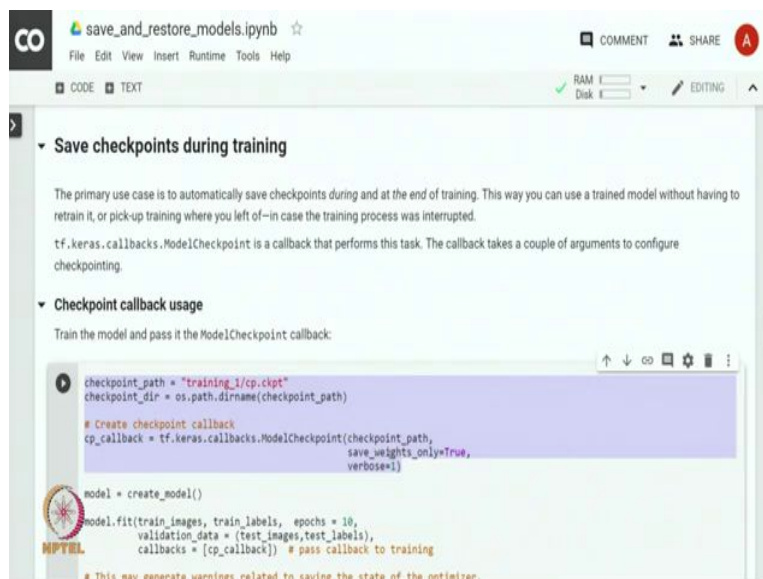
The output of the `model.summary()` call is displayed below the code cell:

```
Model: "sequential"  
Layer (type) Output Shape Param #  
-----  
dense (Dense) (None, 512) 401920  
dropout (Dropout) (None, 512) 0  
dense_1 (Dense) (None, 10) 5130  
-----  
Total params: 407,050  
Trainable params: 407,050  
Non-trainable params: 0
```

At the bottom of the notebook, there is a section titled "Save checkpoints during training" with a small icon of a person and a gear.

So, you can see that the model has got exactly two layers; one hidden layer with 512 units. Then there is a dropout layer, dropout is actually applied on the first layer and then we have an output layer with 10 units. So, there are 407050 parameters in the model.

(Refer Slide Time: 04:28)



The screenshot shows a Jupyter Notebook interface with a file named 'save\_and\_restore\_models.ipynb'. The code cell contains the following Python code:

```
checkpoint_path = "training_1/cp.ckpt"  
checkpoint_dir = os.path.dirname(checkpoint_path)  
  
# Create checkpoint callback  
cp_callback = tf.keras.callbacks.ModelCheckpoint(checkpoint_path,  
                                                save_weights_only=True,  
                                                verbose=1)  
  
model = create_model()  
model.fit(train_images, train_labels, epochs = 10,  
        validation_data = (test_images, test_labels),  
        callbacks = [cp_callback]) # pass callback to training
```

Below the code cell, there is a section titled "Save checkpoints during training" with a small icon of a person and a gear. The text in this section reads:

The primary use case is to automatically save checkpoints during and at the end of training. This way you can use a trained model without having to retrain it, or pick-up training where you left off—in case the training process was interrupted.

`tf.keras.callbacks.ModelCheckpoint` is a callback that performs this task. The callback takes a couple of arguments to configure checkpointing.

**Checkpoint callback usage**

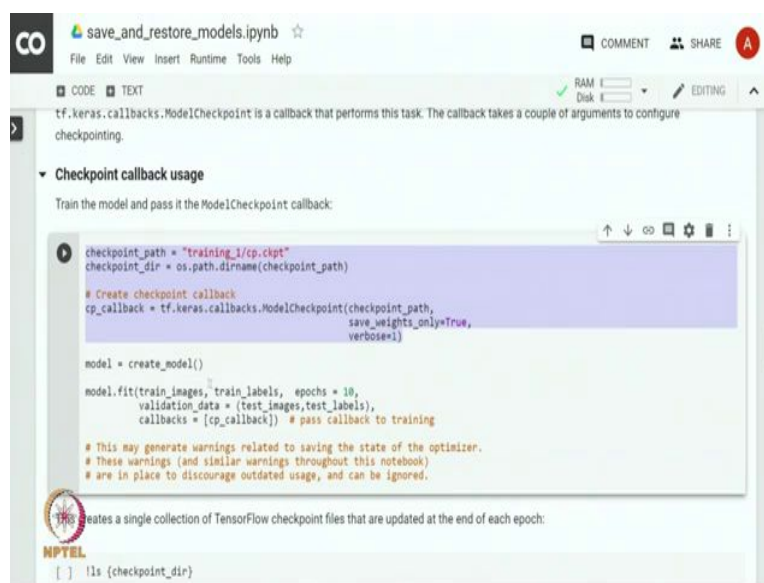
Train the model and pass it the `ModelCheckpoint` callback:

We would like to automatically save checkpoints during training; this way we can use a trained model without having to retrain it or we can pick up the training where we stopped it last time; in case the training process was interrupted or stopped for some reason. We use a

call back model checkpoint for performing this task; this call back takes a few arguments for configuring the checkpointing. Let us look at the usage of checkpoint call back.

So, first we will have to define and configure the checkpoint call backs with call back which is done through this particular code that is highlighted below. We define the checkpoint path as the directory path where we want to store the checkpoint. And then we also configure the checkpoint by specifying what part of model we want to save. Here we are trying to only save the weights and not the architecture or the optimizer configuration.

(Refer Slide Time: 05:51)



```
tf.keras.callbacks.ModelCheckpoint is a callback that performs this task. The callback takes a couple of arguments to configure checkpointing.

▼ Checkpoint callback usage
Train the model and pass it the ModelCheckpoint callback:

checkpoint_path = "training_1/cp.ckpt"
checkpoint_dir = os.path.dirname(checkpoint_path)

# Create checkpoint callback
cp_callback = tf.keras.callbacks.ModelCheckpoint(checkpoint_path,
                                                save_weights_only=True,
                                                verbose=1)

model = create_model()

model.fit(train_images, train_labels, epochs = 10,
        validation_data = (test_images, test_labels),
        callbacks = [cp_callback]) # pass callback to training

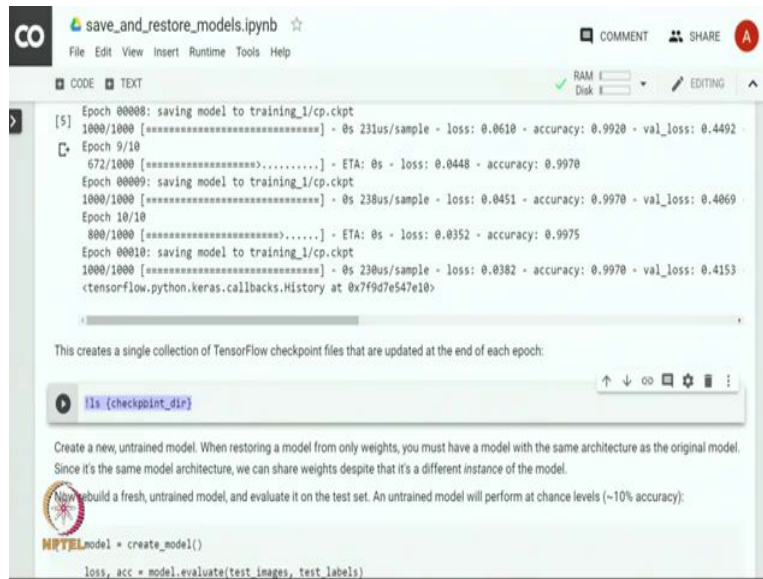
# This may generate warnings related to saving the state of the optimizer.
# These warnings (and similar warnings throughout this notebook)
# are in place to discourage outdated usage, and can be ignored.

# This creates a single collection of TensorFlow checkpoint files that are updated at the end of each epoch:
[ ] !ls {checkpoint_dir}
```

So, this is the simplest configuration for checkpointing. We will see even more advanced usage of checkpointing later in this particular exercise. We create a model with create model command; remind you that create model command actually creates a TensorFlow model that has got one hidden layer of 512 units and an output layer with 10 units.

And you will fit the model by running the training loop for 10 epochs and notice that we are using a call back in the training process. This call back creates a single collection of TensorFlow checkpoint files that are updated at the end of each epoch. So, this particular configuration checkpoints the model at the end of each epoch.

(Refer Slide Time: 06:56)



The screenshot shows a Jupyter Notebook interface with a terminal output area. The output displays the progress of training a model over 10 epochs. At the end of each epoch, the model is saved to a checkpoint directory. Below the output, there is a code cell that runs the command `ls {checkpoint_dir}` to list the contents of the checkpoint directory. The output of this command shows a list of files created during the training process.

```
[5] Epoch 00008: saving model to training_1/cp.ckpt
1000/1000 [=====] - 0s 231us/sample - loss: 0.0610 - accuracy: 0.9920 - val_loss: 0.4492
Epoch 9/10
672/1000 [=====] - ETA: 0s - loss: 0.0448 - accuracy: 0.9970
Epoch 00009: saving model to training_1/cp.ckpt
1000/1000 [=====] - 0s 238us/sample - loss: 0.0451 - accuracy: 0.9970 - val_loss: 0.4069
Epoch 10/10
800/1000 [=====] - ETA: 0s - loss: 0.0352 - accuracy: 0.9975
Epoch 00010: saving model to training_1/cp.ckpt
1000/1000 [=====] - 0s 230us/sample - loss: 0.0382 - accuracy: 0.9970 - val_loss: 0.4153
<tensorflow.python.keras.callbacks.History at 0x779d7e547e10>
```

This creates a single collection of TensorFlow checkpoint files that are updated at the end of each epoch:

```
ls {checkpoint_dir}
```

Create a new, untrained model. When restoring a model from only weights, you must have a model with the same architecture as the original model. Since it's the same model architecture, we can share weights despite that it's a different instance of the model.

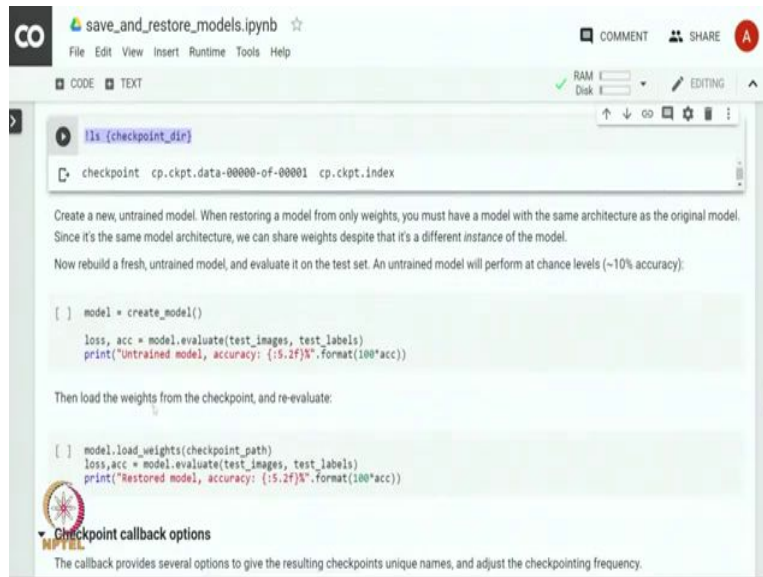
Now rebuild a fresh, untrained model, and evaluate it on the test set. An untrained model will perform at chance levels (~10% accuracy):

```
model = create_model()
loss, acc = model.evaluate(test_images, test_labels)
```

Let us train the model quickly and look at the checkpoint directory. So, an exclamation mark followed by any command that we write is interpreted as a unix command and is run as if we are running it on the command line. So, this particular code snippet will print the directory listing for the checkpoint directory.

So, you can see that there is a checkpoint; there are there are few files that are that are created in the checkpoint directory. So, here we will have to first create a model with the same architecture as the original model and then restore the weights and apply those weights in the new model. It is perfectly fine to share the weights from the previous run; even though this is a different instance of a model.

(Refer Slide Time: 08:10)



The screenshot shows a Jupyter Notebook interface with the title 'save\_and\_restore\_models.ipynb'. The code is written in Python and demonstrates how to create a new, untrained model and then load weights from a checkpoint to restore the model. The code includes comments explaining the process and the expected accuracy of an untrained model (~10%).

```
!ls {checkpoint_dir}

checkpoint cp.ckpt.data-00000-of-00001 cp.ckpt.index

Create a new, untrained model. When restoring a model from only weights, you must have a model with the same architecture as the original model.
Since it's the same model architecture, we can share weights despite that it's a different instance of the model.

Now rebuild a fresh, untrained model, and evaluate it on the test set. An untrained model will perform at chance levels (~10% accuracy):

[ ] model = create_model()
    loss, acc = model.evaluate(test_images, test_labels)
    print("Untrained model, accuracy: {:.2f}%".format(100*acc))

Then load the weights from the checkpoint, and re-evaluate:

[ ] model.load_weights(checkpoint_path)
    loss, acc = model.evaluate(test_images, test_labels)
    print("Restored model, accuracy: {:.2f}%".format(100*acc))
```

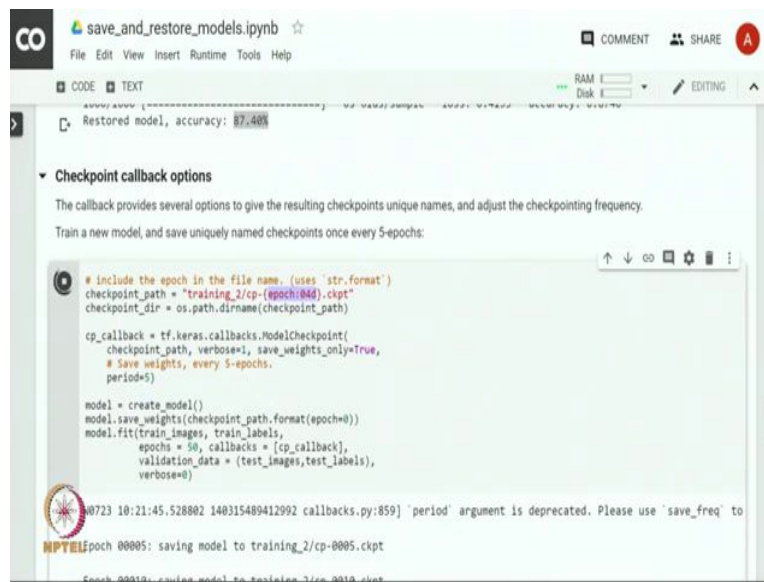
**Checkpoint callback options**  
The callback provides several options to give the resulting checkpoints unique names, and adjust the checkpointing frequency.

Before applying the weights, we will create a model and evaluate the model performance in the test even before restoring the parameters. So, in this case some random values will be used for parameters and we will see the accuracy that we get is just by chance. So, here we get only 10 percent accuracy as against the 99 percent accuracy that we got or 87 percent validation accuracy, that we got during the training.

Now, let us load the weights from the checkpoint path and again evaluate the model and check the accuracy. We can see that we are able to get the accuracy of 87 percent as we got earlier during the training of the model.



(Refer Slide Time: 09:01)



```
Restored model, accuracy: 87.48%
```

**Checkpoint callback options**

The callback provides several options to give the resulting checkpoints unique names, and adjust the checkpointing frequency.

Train a new model, and save uniquely named checkpoints once every 5-epochs:

```
# include the epoch in the file name. (uses 'str.format')
checkpoint_path = "training_2/cp-(epoch:040).ckpt"
checkpoint_dir = os.path.dirname(checkpoint_path)

cp_callback = tf.keras.callbacks.ModelCheckpoint(
    checkpoint_path, verbose=1, save_weights_only=True,
    # Save weights, every 5-epochs.
    period=5)

model = create_model()
model.save_weights(checkpoint_path.format(epoch=0))
model.fit(train_images, train_labels,
          epochs=50, callbacks=[cp_callback],
          validation_data=(test_images, test_labels),
          verbose=0)
```

0723 10:21:45.528802 140315489412992 callbacks.py:859] 'period' argument is deprecated. Please use 'save\_freq' to

Epoch 00005: saving model to training\_2/cp-0005.ckpt

So, we can see that just by using the same model, but just by building the architecture as the original model and restoring weights helped us to get the same performance as the original model. So, you can see that this is very powerful.

Let us look at various options that we have for creating a checkpoint call back. We can specify a period instead of saving the checkpoint after every epoch; we can specify a period after which the model should be saved. So, in this case we can; we can do that with a period argument and here we are setting period to 5. So, we are going to save weights every 5 epochs rather than doing it after every epoch and here we are only going to save the weights.

We also give the checkpoint path and configure it to store the ID of the epoch. So, this is a unique ID that is created for a checkpoint which consist of the ID of the epoch so that it is easy to identify what epoch is a checkpoint from. Then we create the model, we save the weights to the checkpoint path and then fit the model. And note that in the fit function we give the call back as one of the arguments and you can see that the model is getting saved after every 5 epochs.

As we are training for 50 epochs, we should see that there are 10 checkpoints. So, you can see the checkpoint at 5th epoch, 10th epoch and so on up to 50th epoch.



(Refer Slide Time: 11:13)

save\_and\_restore\_models.ipynb

File Edit View Insert Runtime Tools Help

CODE TEXT

RAM 100% Disk 100% EDITING

```
[9] Epoch 00050: saving model to training_2/cp-0050.cpkt
<tensorflow.python.keras.callbacks.History at 0x7f9d7da52a20>
```

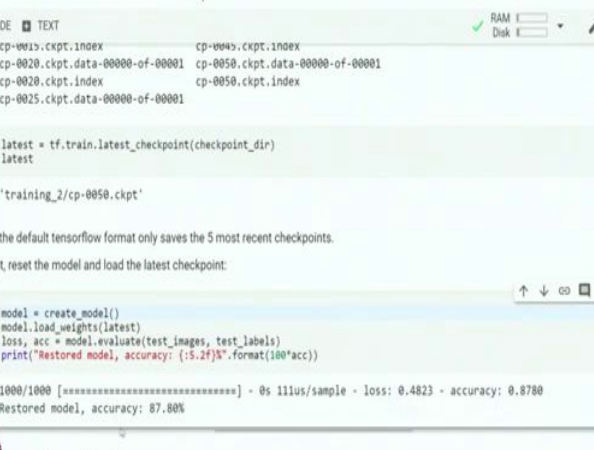
Now, look at the resulting checkpoints and choose the latest one:

```
ls (checkpoint_dir)
checkpoint
cp-0000.cpkt.data-00000-of-00001
cp-0000.cpkt.index
cp-0005.cpkt.data-00000-of-00001
cp-0005.cpkt.index
cp-0010.cpkt.data-00000-of-00001
cp-0010.cpkt.index
cp-0015.cpkt.data-00000-of-00001
cp-0015.cpkt.index
cp-0020.cpkt.data-00000-of-00001
cp-0020.cpkt.index
cp-0025.cpkt.data-00000-of-00001
cp-0025.cpkt.index
cp-0030.cpkt.data-00000-of-00001
cp-0030.cpkt.index
cp-0035.cpkt.data-00000-of-00001
cp-0035.cpkt.index
cp-0040.cpkt.data-00000-of-00001
cp-0040.cpkt.index
cp-0045.cpkt.data-00000-of-00001
cp-0045.cpkt.index
cp-0050.cpkt.data-00000-of-00001
cp-0050.cpkt.index
```

```
latest = tf.train.latest_checkpoint(checkpoint_dir)
latest
```

Let us look at the content of the checkpoint directory and we can see that there are 10 different checkpoints that are stored in the directory.

(Refer Slide Time: 11:29)



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

```

save_and_restore_models.ipynb
File Edit View Insert Runtime Tools Help

CODE TEXT
[11] cp-0020.ckpt.data-00000-of-00001 cp-0050.ckpt.data-00000-of-00001
     cp-0020.ckpt.index cp-0050.ckpt.index
     cp-0025.ckpt.data-00000-of-00001

[12] latest = tf.train.latest_checkpoint(checkpoint_dir)
     latest

     'training_2/cp-0050.ckpt'

Note: the default tensorflow format only saves the 5 most recent checkpoints.

To test, reset the model and load the latest checkpoint:

model = create_model()
model.load_weights(latest)
loss, acc = model.evaluate(test_images, test_labels)
print('Restored model, accuracy: {:.2f}%'.format(100*acc))

1000/1000 [=====] - 0s 111us/sample - loss: 0.4823 - accuracy: 0.8780
Restored model, accuracy: 87.80%
  
```

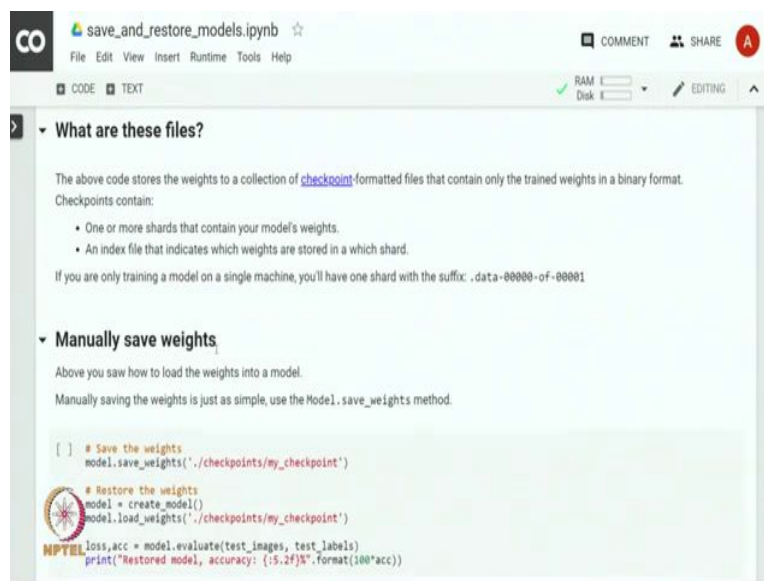
What are these files?

If we use `latest_checkpoint` as a function and give `checkpoint` underscore directory or the checkpoint directory as an argument, we get the latest checkpoint. By default TensorFlow

format only saves the 5 most recent checkpoints. So, let us retrieve the latest checkpoint and create the model with the weights from the latest checkpoint.

What are these different files that are there in the checkpoint directory? Let us take a look at them. Since we train our model on a single machine each checkpoint will have all the weight stored in a single shard. If you are doing it on multiple machines there could be there could have been multiple shards.

(Refer Slide Time: 12:19)



The screenshot shows a Jupyter Notebook interface with the title 'save\_and\_restore\_models.ipynb'. The notebook is in 'TEXT' view. It contains two sections: 'What are these files?' and 'Manually save weights'.

**What are these files?**

The above code stores the weights to a collection of [checkpoint](#)-formatted files that contain only the trained weights in a binary format. Checkpoints contain:

- One or more shards that contain your model's weights.
- An index file that indicates which weights are stored in a which shard.

If you are only training a model on a single machine, you'll have one shard with the suffix: `.data-00000-of-00001`

**Manually save weights**

Above you saw how to load the weights into a model.

Manually saving the weights is just as simple, use the `Model.save_weights` method.

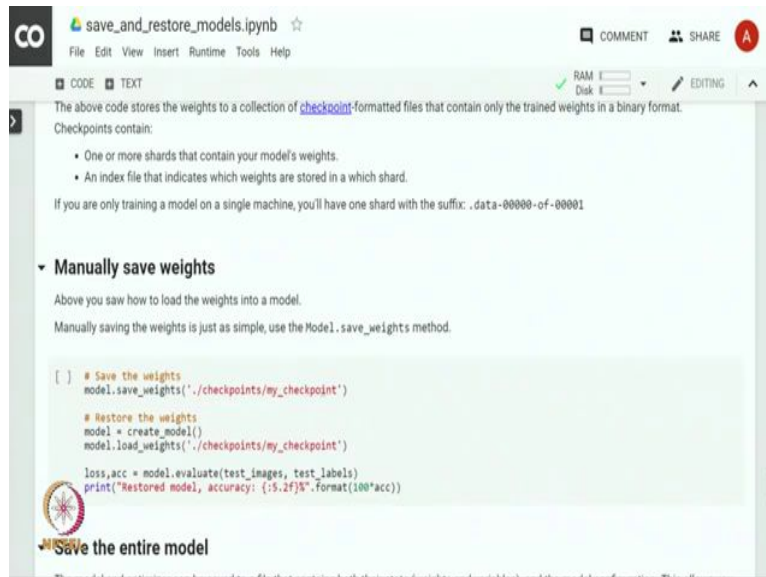
```
[ ] # Save the weights
model.save_weights('./checkpoints/my_checkpoint')

# Restore the weights
model = create_model()
model.load_weights('./checkpoints/my_checkpoint')

loss, acc = model.evaluate(test_images, test_labels)
print("Restored model, accuracy: {:.2f}%".format(100*acc))
```

Apart from the call back we can also manually save the weight that is the other way of saving the weight.

(Refer Slide Time: 12:27)



The screenshot shows a Jupyter Notebook interface with the title 'save\_and\_restore\_models.ipynb'. The notebook is in 'TEXT' mode. The content includes a description of checkpoints, a list of their components, and a code cell for manually saving and restoring weights. The code cell contains the following Python code:

```
[ ] # Save the weights
model.save_weights('./checkpoints/my_checkpoint')

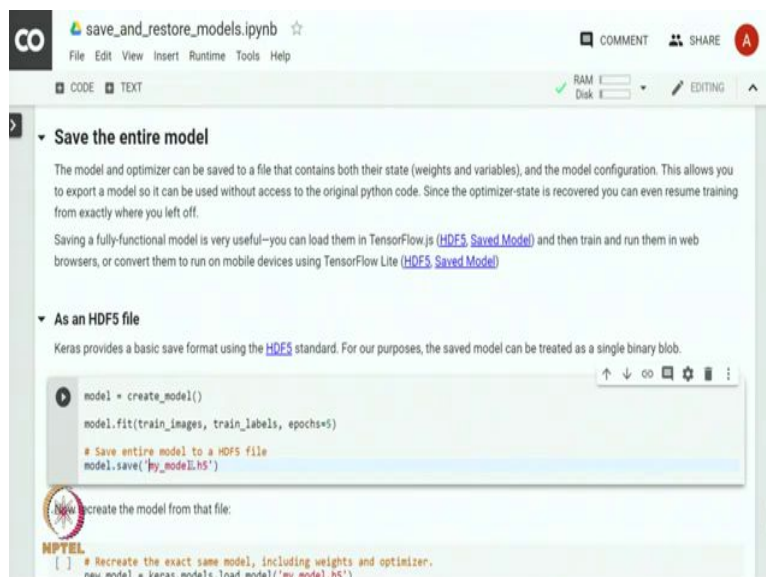
# Restore the weights
model = create_model()
model.load_weights('./checkpoints/my_checkpoint')

loss, acc = model.evaluate(test_images, test_labels)
print("Restored model, accuracy: {:.2f}%".format(100*acc))
```

Below the code cell, there is a section titled 'Save the entire model' with a sub-section 'Manually save weights'.

And we can simply use `model.save_weights` function and we can provide the directory or the path and we have to provide the file name where we want to store the weights. Let us run it to check it. So, we are essentially saving the weights to `my_checkpoint` file and we are loading the weight from that particular file.

(Refer Slide Time: 12:51)



The screenshot shows a Jupyter Notebook interface with the title 'save\_and\_restore\_models.ipynb'. The notebook is in 'TEXT' mode. The content includes a description of saving the entire model and a code cell for saving and restoring the entire model. The code cell contains the following Python code:

```
[ ] # Save the entire model
model = create_model()
model.fit(train_images, train_labels, epochs=5)

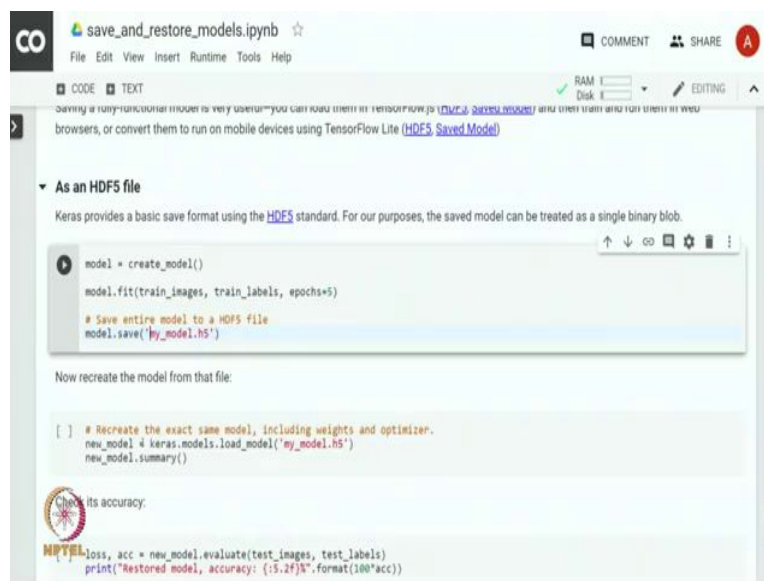
# Save entire model to a HDF5 file
model.save('my_model.h5')
```

Below the code cell, there is a section titled 'Save the entire model' with a sub-section 'As an HDF5 file'.

So, you can see that we are getting again 87 percent accuracy after saving the weight and restoring it in a new model. Instead of only saving the weights we can also save the architecture of the model or the optimizer configuration.

So, the entire model can be saved using hierarchical data format or HDF5; we can specify the HDF5 with h5 as an extension. Here we create the model, we train the model and we will save the model into HDF5 file with the file name my\_model.h5.

(Refer Slide Time: 13:42)



```
model = create_model()
model.fit(train_images, train_labels, epochs=5)

# Save entire model to a HDF5 file
model.save('my_model.h5')

Now recreate the model from that file:

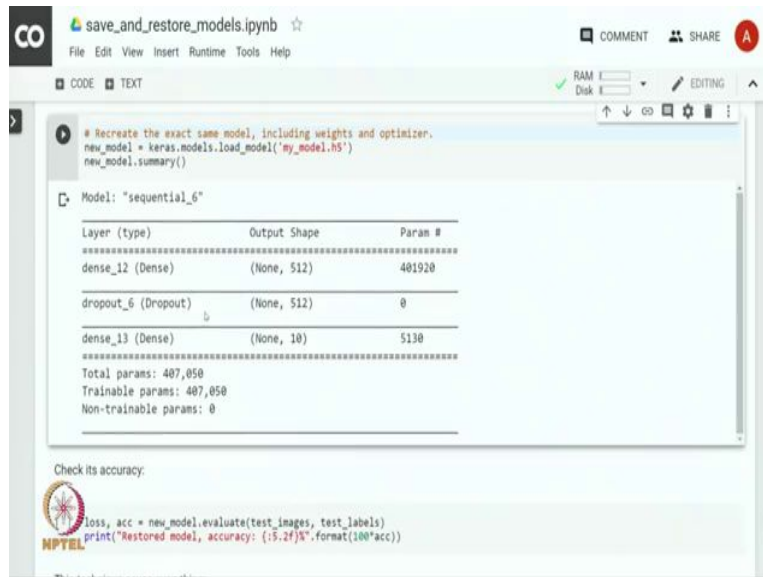
[ ] # Recreate the exact same model, including weights and optimizer.
new_model = keras.models.load_model('my_model.h5')
new_model.summary()

# Evaluate its accuracy.
loss, acc = new_model.evaluate(test_images, test_labels)
print('Restored model, accuracy: {:.2f}%'.format(100*acc))
```

Later we can load the model from this particular file the HDF5 file and use it for prediction. I would like to point out the difference between the earlier checkpointing method where we were only storing weight as against this particular method where we are storing the entire model.

In the checkpointing, we had to first create the model and then load the weights into the model and then use it for the prediction task. In this case, we do not have to create the model as the model itself has been saved in HDF format, we simply load the model that; that essentially creates the model, puts the weight and the model is used and the model is ready for the prediction task; it is important to note this particular difference; let us load the model.

(Refer Slide Time: 14:46)



The screenshot shows a Jupyter Notebook titled 'save\_and\_restore\_models.ipynb'. The code cell contains the following Python code:

```
# Recreate the exact same model, including weights and optimizer.
new_model = keras.models.load_model('my_model.h5')
new_model.summary()
```

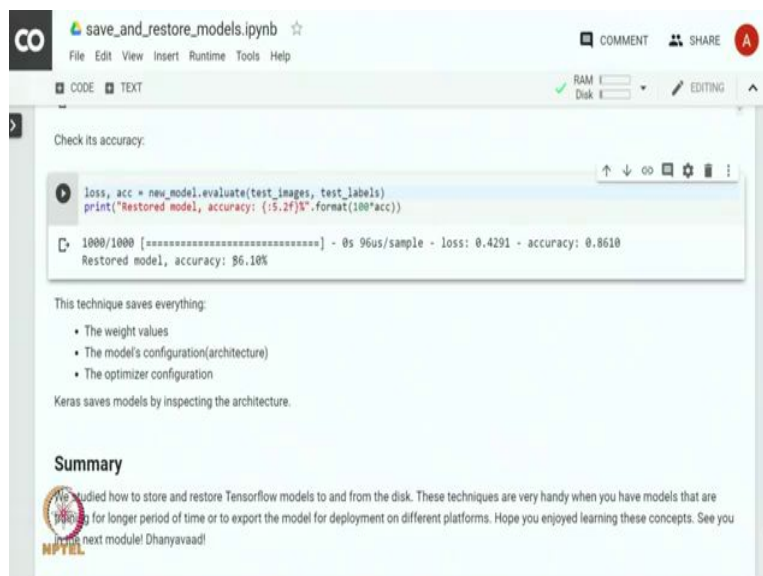
The output of the `summary()` method is displayed as a table:

Layer (type)	Output Shape	Param #
=====		
dense_12 (Dense)	(None, 512)	401920
dropout_6 (Dropout)	(None, 512)	0
dense_13 (Dense)	(None, 10)	5130
=====		
Total params:	407,050	
Trainable params:	407,050	
Non-trainable params:	0	

Below the table, the text 'Check its accuracy:' is followed by a code cell:

```
loss, acc = new_model.evaluate(test_images, test_labels)
print("Restored model, accuracy: {:.2f}%".format(100*acc))
```

(Refer Slide Time: 14:54)



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

```
loss, acc = new_model.evaluate(test_images, test_labels)
print("Restored model, accuracy: {:.2f}%".format(100*acc))
```

The output of the `evaluate()` method is displayed as a text block:

```
1000/1000 [=====] - 0s 96us/sample - loss: 0.4291 - accuracy: 0.8610
Restored model, accuracy: 86.10%
```

Below the output, the text 'This technique saves everything:' is followed by a bulleted list:

- The weight values
- The model's configuration (architecture)
- The optimizer configuration

Below the list, the text 'Keras saves models by inspecting the architecture.' is displayed.

**Summary**

We studied how to store and restore Tensorflow models to and from the disk. These techniques are very handy when you have models that are training for longer period of time or to export the model for deployment on different platforms. Hope you enjoyed learning these concepts. See you in the next module! Dhanyavaad!

And we can see that this model has got exactly the same summary as before and then check the accuracy of the model; it is almost the same accuracy of around 87 percent. So, this technique saves everything essentially weights, model configuration and optimizer configuration and keras saves the model by inspecting its architecture.

In this module, we studied how to store and restore TensorFlow models to and from the disk. These techniques are very handy when you have models that are training for a long period of

time or to export model for deployment on different platforms. Hope you enjoyed learning these concepts.