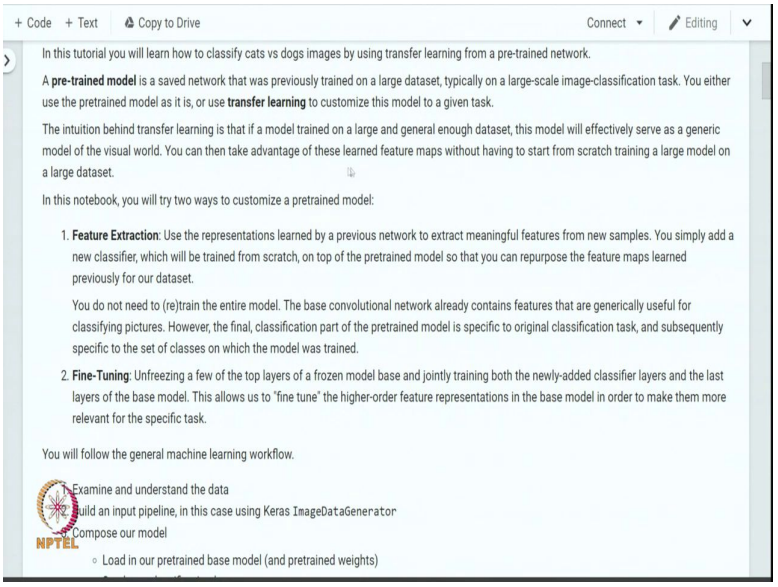
Practical Machine Learning
Dr. Ashish Tendulkar
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture – 24
Transfer Learning with pre-trained CNNs

In the last lecture, we looked at concepts behind convolution neural networks and use them for building an image classifier. We also learned that these models have a large number of parameters and in order to train them without overfitting we require a reasonably large amount of data. How do we use these models when we do not have a large amount of data? For example, say a manufacturing company wants to train a CNN model for recognizing faulty machine parts, and they do not have enough data, what can we do?

We learn techniques behind solving these kind of problems in this session. We will demonstrate these techniques by classifying cats and dogs.

(Refer Slide Time: 01:23)

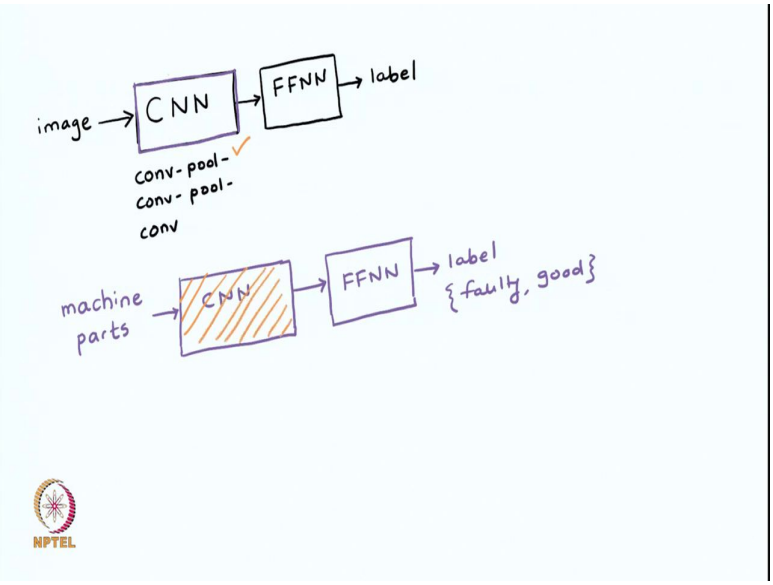


A pre-trained model is at the centre of our strategy. A pre-trained model is a saved network that was previously trained on a large dataset, typically on a large scale image classification task. We either use the pre-trained model as it is or use transfer learning to customize this model for a given task. The intuition behind transfer learning is that if a model is trained on a

large and general enough data set, this model will effectively serve as a general model of the visual world. We can take advantage of these learned feature maps without having to start training a large model from scratch on a large data set.

In this exercise we will try two ways to customize a pre-trained model.

(Refer Slide Time: 02:19)

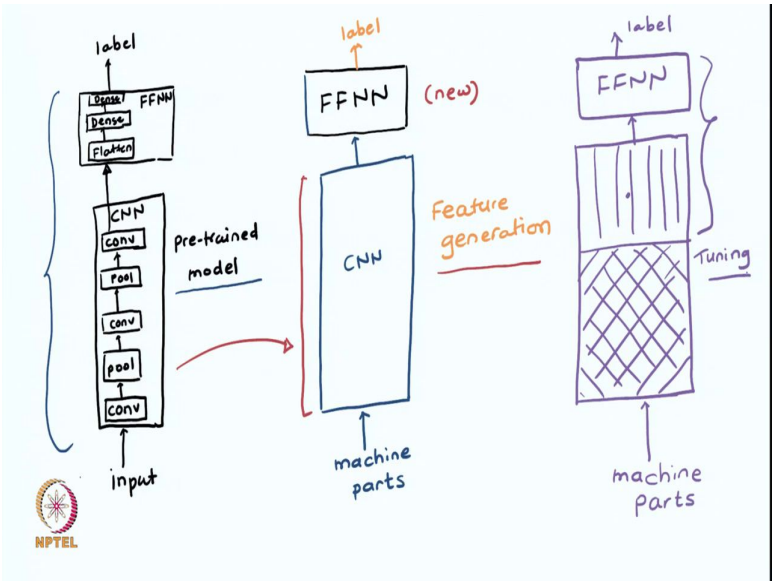


CNNs are made up of one or more convolution and pooling layers, generally followed by dense layers. For example, we might have two convolution pooling layers followed by a convolution layer whose output is then fed into a dense layer to generate a label.

Now, the idea here is to use this CNN model which was trained on a large data set and use it for performing some other task. For example, we have a data set of machine parts and we want to recognize faulty machine parts. So, we want to build a CNN followed by feed forward neural network to get the label, which in this case is faulty or good. We know that a CNN model has a large number of parameters and if you do not have enough data points about machine parts and a label you are likely to overfit the CNN model.

Here, what we want to do is we want to take advantage of a pre-trained model which is trained on a different dataset from the dataset pertaining to our problem, and use it for solving the problem at hand. There are really two ways in which we can achieve this.

(Refer Slide Time: 05:29)



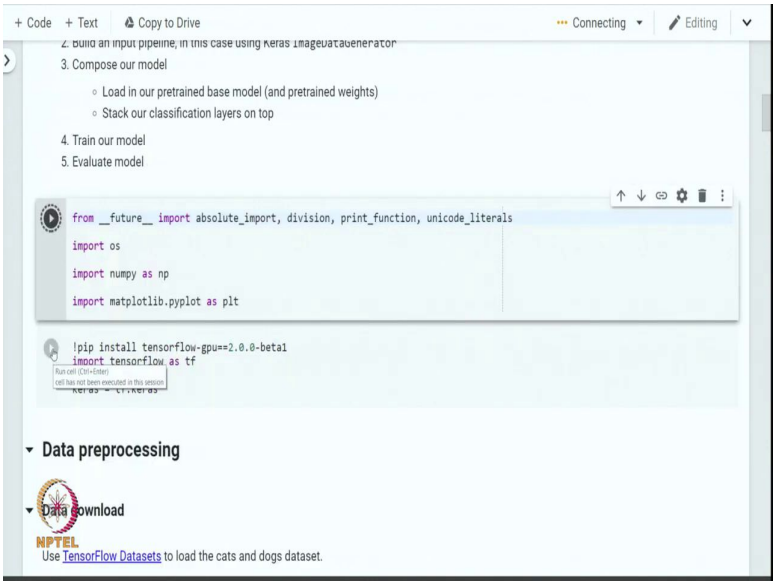
Let us quickly introduce some generic structure to our CNNs. We have the convolutional neural network (CNN) part, which consists of convolutional layers and pooling layers. The other part, referred to as feed forward neural network (FFNN) consists of dense layers.

In the first of the two ways we can use our pretrained model, we will only use the convolutional part of the model. We use the convolutional part of the model for feature generation and pass the output of the CNN to a new network of dense layers. With the new data, we only train these new layers that were added and not the CNN. The weights of the CNN remain the same as the CNN part of the pretrained model.

In the second approach what we do is, we use a certain section of the CNN as it is in the pretrained model and retrain the rest of the network. The certain section is typically the first few layers of the network (refer to the image above). The training is done by freezing the weights in the earlier layers and tuning the rest of the weights using the training data at hand.

Now that you have understood how to use pre-trained model for building a custom model, let us look at the machine learning workflow involved in this particular process.

(Refer Slide Time: 12:05)

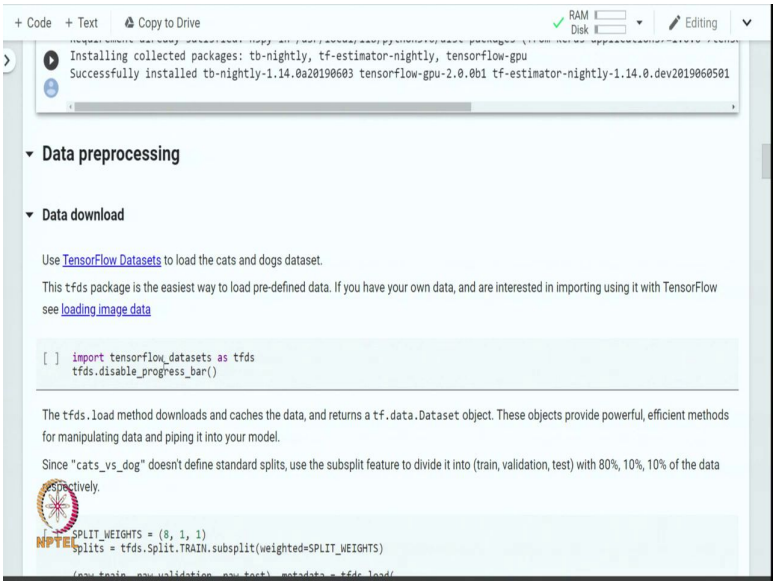


So, we will first examine and understand the data which is exactly the same as traditional machine learning algorithms, then we build an input pipeline, then we build and compile our model. Only the model composition differs from the traditional machine learning algorithms.

In traditional machine learning algorithms, we define the model completely. In this case, we will have to load the pre-trained model and then add the classifier layer (as a dense layer) on top of it. Then, the remaining two steps (training and evaluation) are again very similar to our traditional machine learning algorithms.

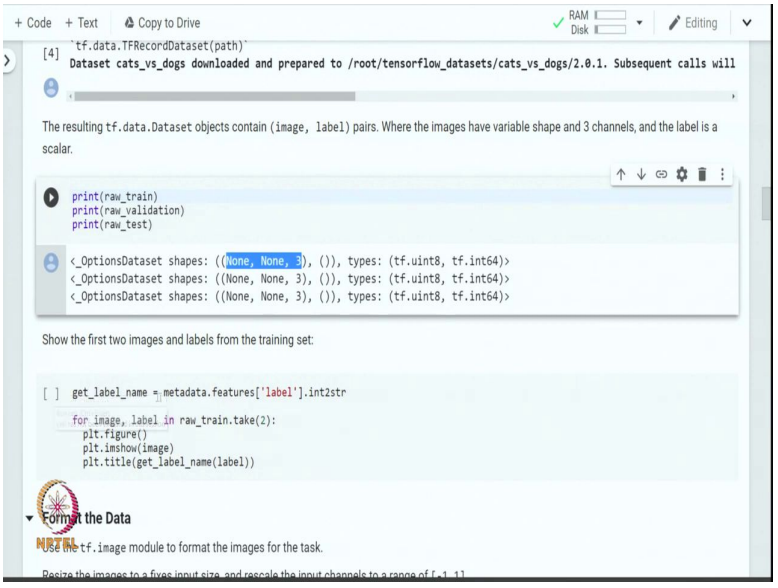
So, let us start by importing all the necessary libraries. Let us also install and import the tensorflow package.

(Refer Slide Time: 13:29)



Let us load the data set using tfds package.

(Refer Slide Time: 13:34)

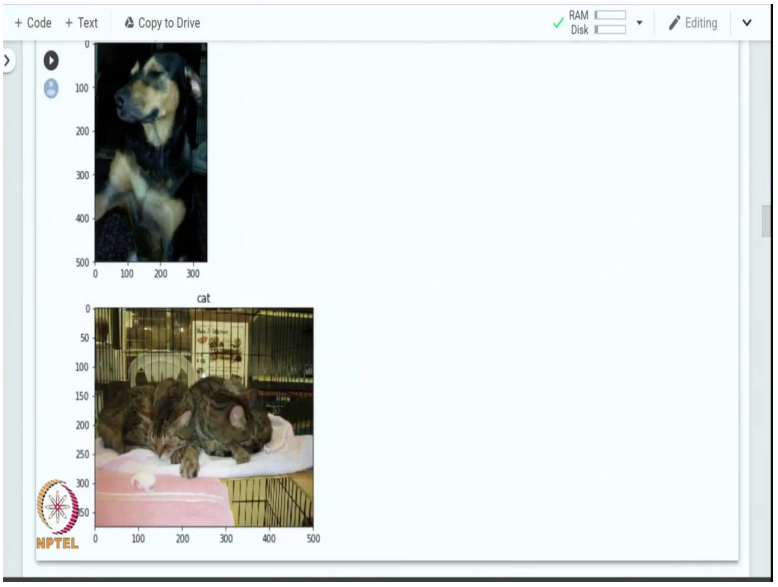


The tfds.load method downloads and caches data and returns a tf.data.Dataset object. These objects provide powerful and efficient methods for manipulating data and piping it into our model. Since cats and dogs data set does not define standard split we use the subsplit function to divide it into train, validation and test with the split specified by the weighted parameter.

Here we use 80 percent data for training, 10 percent data for test and the remaining 10 percent data for validation.

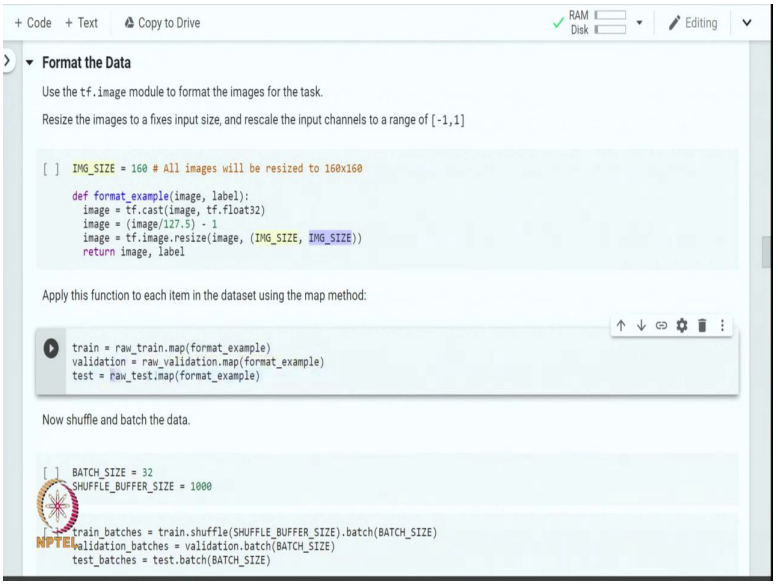
The resulting dataset object contains images and label pairs. The images have variable shape and 3 channels, and label is a scalar quantity. Let us look at the first two images and add labels from the training set. So, iterate on the training set using take method on the tensor and we generate the string of the label using into string property.

(Refer Slide Time: 14:59)



You can see that this is a picture of a dog with label displayed at the top and there is a picture of the cat. You can see that the dog picture has height of 500 and width of over 350; whereas, the cat picture has height close to 400 and width of 500. So, you can see that all the pictures are not of the same size.

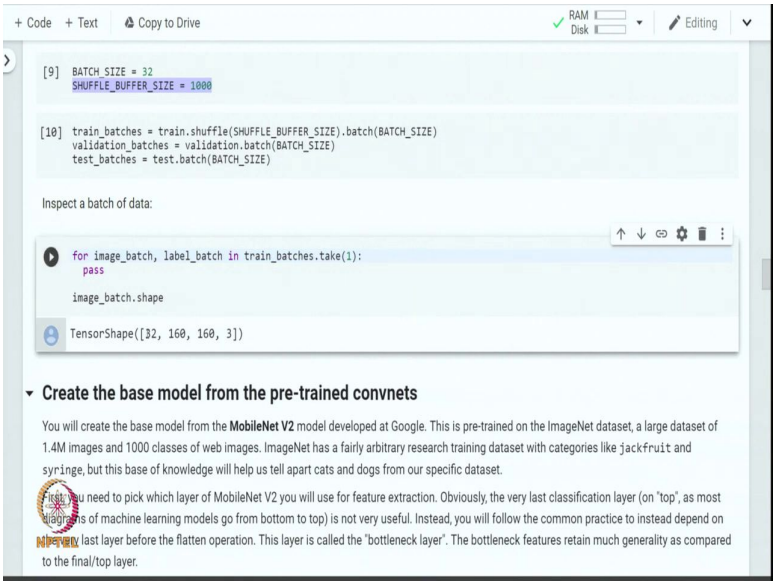
(Refer Slide Time: 15:28)



So, the first thing is to resize the image, so that we have the same input size for all the images. We will be using tf.image module to format the images for this task. We will also rescale the input channel to a range of minus 1 to plus 1. Here the desired size of image is 160 by 160. We will apply the function on each item in the dataset using the map method.

So, you can see that the format_example is applied on the training set, validation set and test set. We get three tensors - train, validation and test - that contains images of the same size and their labels.

(Refer Slide Time: 16:39)



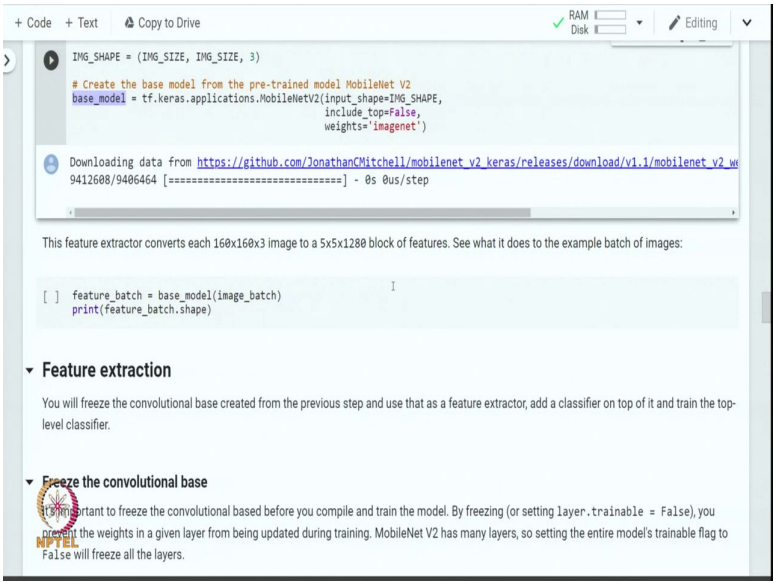
Let us shuffle and batch the data. We use batch size of 32, and for shuffling we define a buffer size of 1000. We shuffle only the training data and we batch all the three data sets by the batch size of 32. Let us inspect a batch of data from training batches. So, we can see that in the first training batch we have 32 images, each with height of 160 and width of 160 on the 3 channels. So, you can see that the image batch here is a 4D tensor.

Now that we have got in the data in the desired shape let us create the model. In model creation, there are two steps; first is to load the base convolution model and second, o add a classification layer on top of it.

Here we will create the base model from MobileNet version 2 developed at Google. This model was pre-trained on image data set which is a large data set of 1.4 million images from thousand classes of web images. ImageNet has categories like jack fruit and syringe, but we will use the image net classified here to classify cats versus dogs.

First you need to pick which layer of MobileNet you will use for feature extraction; obviously, very last classification layer is not going to be very useful for this task. Instead we will follow a common practice of extracting features at a layer just before the flatten operation. This layer is referred to as the bottleneck layer. A bottleneck feature retains generality as compared to final or top layer.

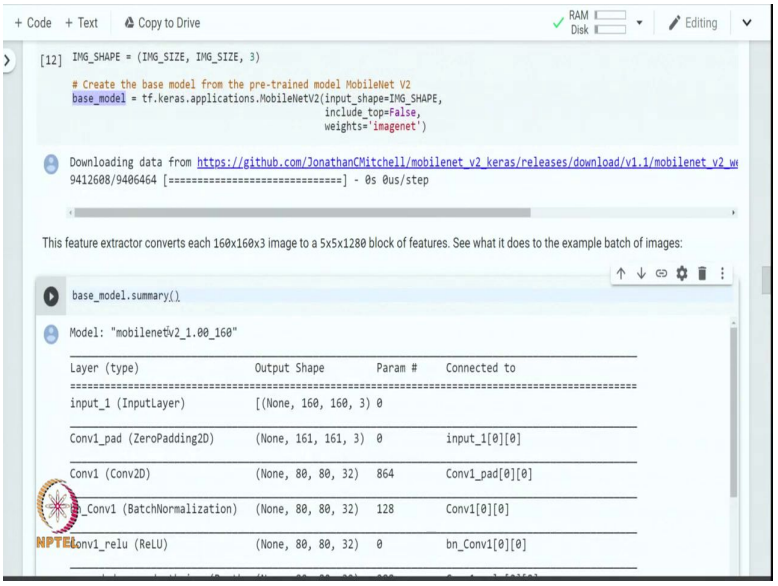
(Refer Slide Time: 18:53)



So, let us first instantiate MobileNet with a pre-loaded weights trained on ImageNet. We can do that using tf.keras.applications.MobileNetV2 function. We specify the input shape. We tell the model that we do not want to include the top layer or the classifier layer by specifying include_top argument to false and we specify that we want to use weights of MobileNetV2 when trained on the ImageNet dataset.

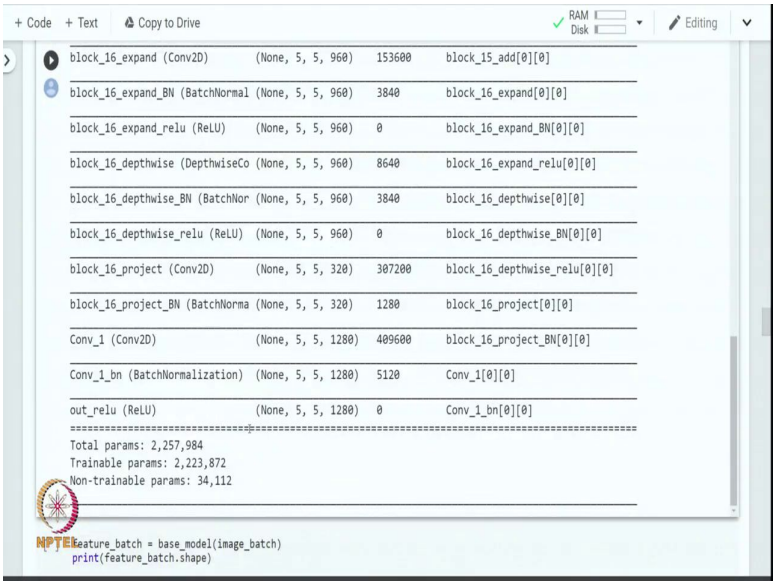
Since we specify include_top as false the network does not include the classification layer at the top which is ideal for feature extraction. The feature extractor converts each of 160x160x3 image into a 5x5x1280 block of features. Let us look at how this model looks like.

(Refer Slide Time: 19:58)



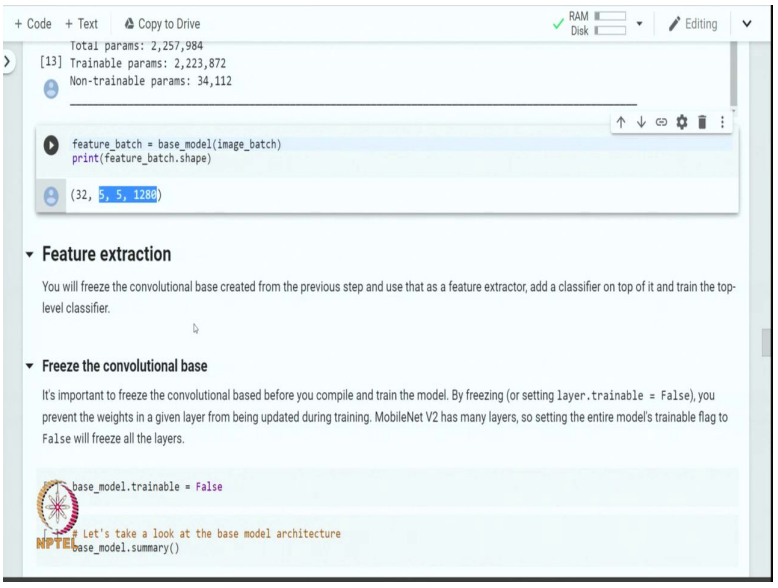
So, when we call summary on the model you get to see the complete architecture of the MobileNet V2. We can see that it takes a 4D tensor, where there are images of size 160x160 across 3 channels; that means, it takes coloured images of size 160x160.

(Refer Slide Time: 20:28)



And, its final layer produces a 4D tensor, where we get 5x5 patches across 1280 channels. We also see the total number of parameters for this model. So, this model has got 2.2 million parameters.

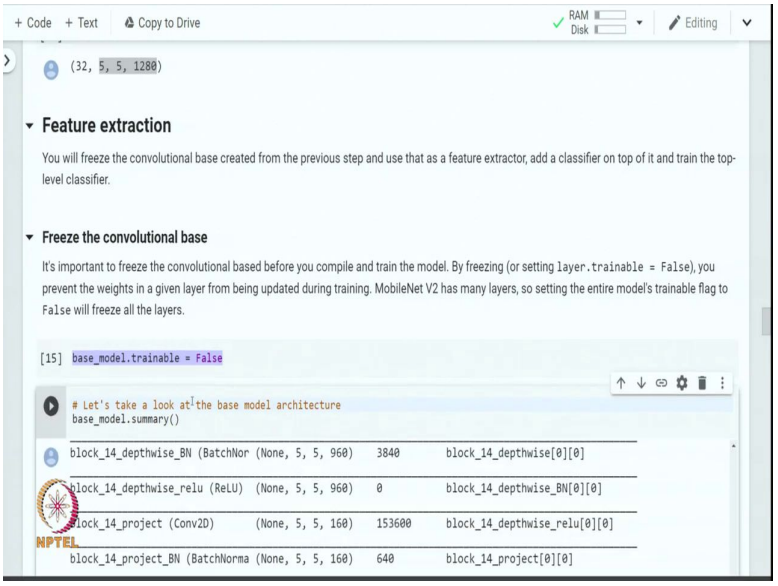
(Refer Slide Time: 20:59)



Let us use the base model to generate the features.

So, you can see that on the image batch that we selected we computed the features for the image batch. You can see that for each of the 32 examples in the batch, we got a 3D tensor of size of shape (5, 5, 1280). We will freeze the convolution base created from the previous step and use that as a feature extractor. We add a classifier on top of it and train the top level classifier.

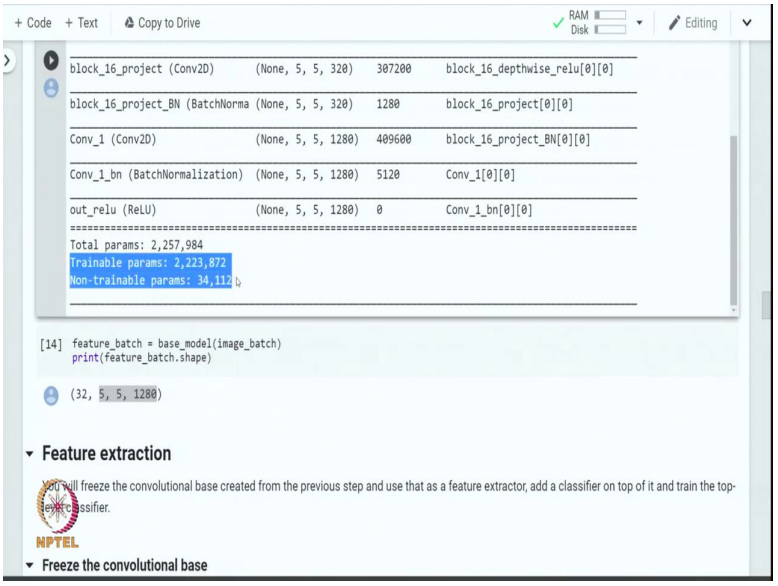
(Refer Slide Time: 21:48)



Let us see how to do that in the code. We use base_model.trainable attribute or property and set it to false. This makes sure that you freeze the convolution base before we compile and train the model. By freezing you prevent the weights in a given layer from being updated during the training. MobileNet has many layers, so, setting the entire models trainable flat to false will freeze all the layers.

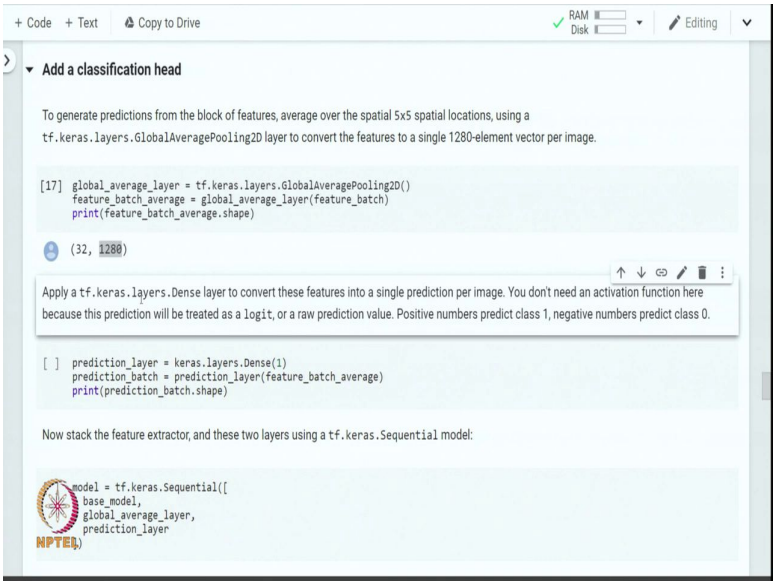
Let us look at how the base model looks like.

(Refer Slide Time: 22:24)



I would like you to compare the trainable parameters after freezing. You can see that after freezing the trainable parameters become 0; that means we do not have to train any of the parameter of this network and all the parameter all the 2.2 million parameters become non-trainable.

(Refer Slide Time: 22:52)

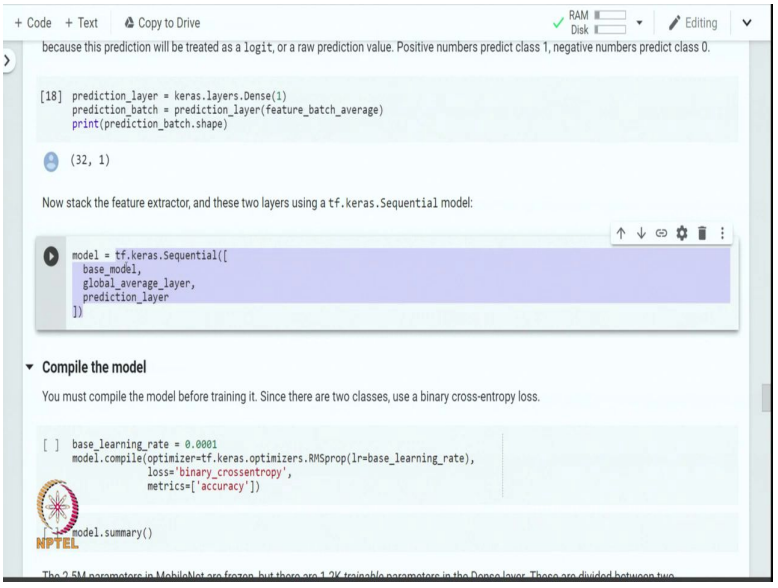


In order to generate predictions from the block of the feature we average the special 5x5 block using a GlobalAveragePooling layer to convert the feature to a single vector of 1280

elements per image. So, we add the GlobalAveragePooling layer to the model. Let us look at the shape of the feature batch. You can see that each of the image each of the 32 images in the batch got converted into a 1D tensor containing 1280 numbers.

Finally we apply tf.keras.layers.dense layer to convert these features into a single prediction per image. We do not need an activation function here because this prediction will be treated as a raw prediction value. A positive number predicts class 1 and negative number predicts class 0.

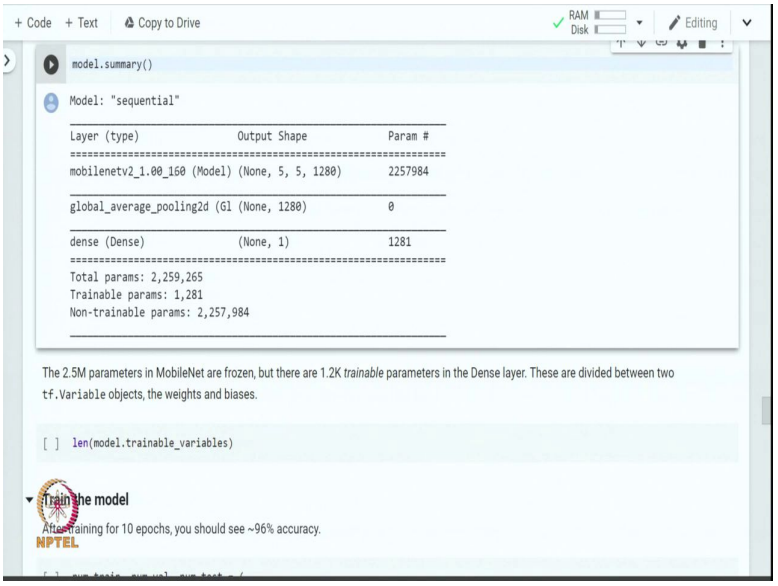
(Refer Slide Time: 24:03)



So, let us stack the feature extractor and two layers using a tf.keras.Sequential model. So, we define the model to be tf.keras.Sequential model which has got a base model, which itself is a sequence of convolution and pooling layers according to the MobileNet architecture. You have a global average layer followed by a prediction layer. Prediction layer is a dense layer having a single unit. Let us compile the model before training it.

Since there are two classes we are using binary cross entropy loss and we are using RMSprop optimizer with a small learning rate.

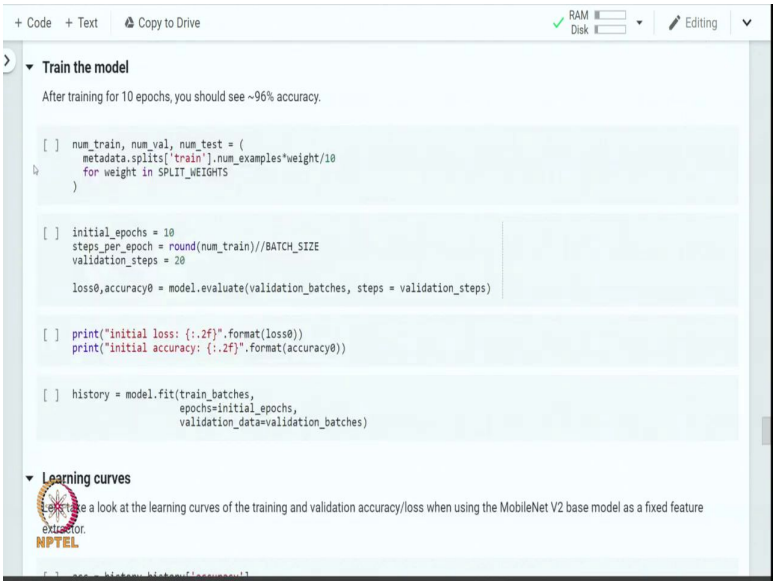
(Refer Slide Time: 24:52)



Let us look at the summary of the model. You can see that the MobileNetV2 returns the 4D tensor of shape (None, 5, 5, 1280) and it has got 2.2 million parameters. The global average pooling returns 1280 numbers per image and finally, we have a dense layer with a single unit.

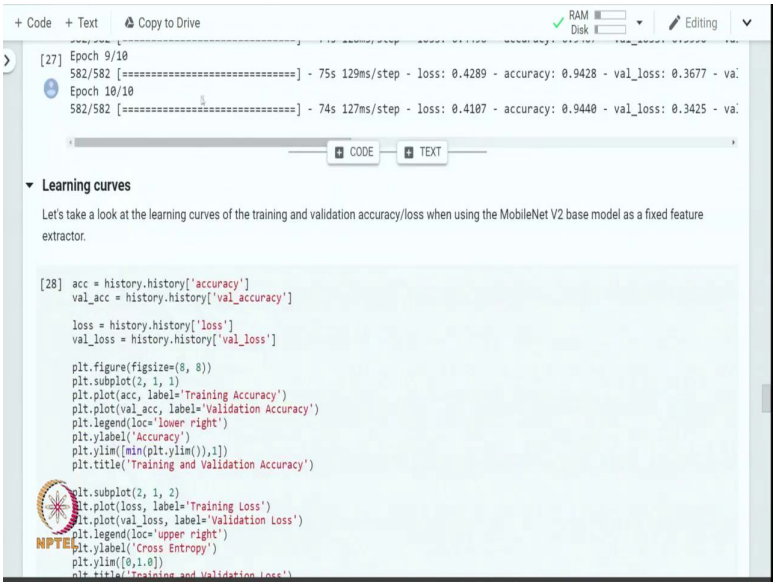
We can see that this dense layer receive 1280 inputs plus 1 bias unit makes 1281 parameters in the dense layer. So, you can see that the total number of parameters are more than 2.2 million, out of which most of the parameters are non-trainable and we have to only train 1281 parameters corresponding to the output layer.

(Refer Slide Time: 25:44)



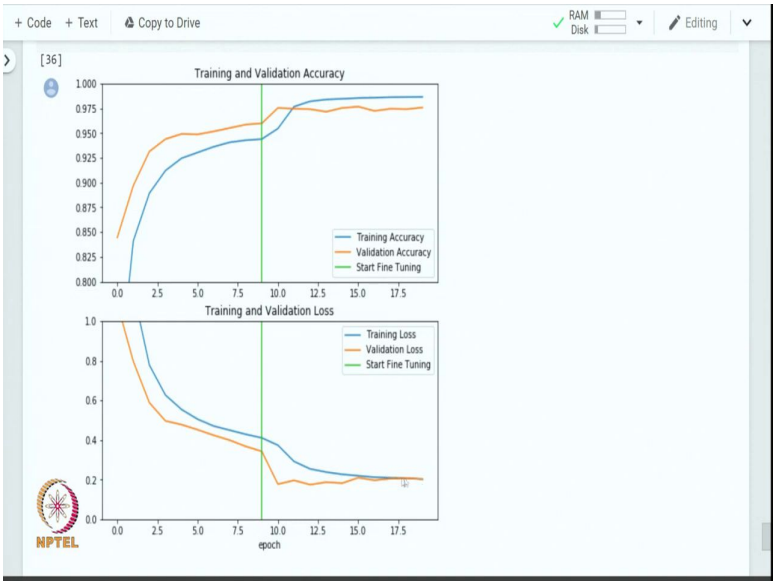
Let us train the model for 10 epochs.

(Refer Slide Time: 25:47)



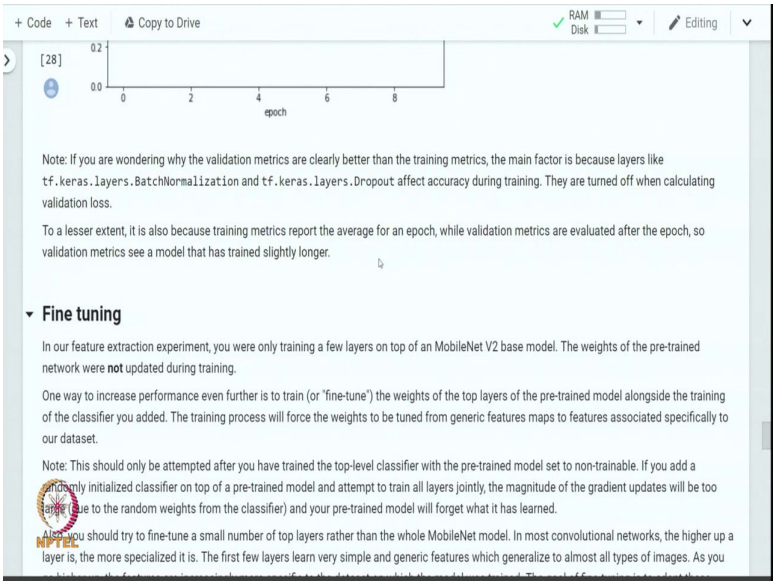
You can see that after 10th epoch we cross the accuracy of 94 percent. Let us take a look at the learning curves of the training and validation accuracy or loss when using MobileNet base model as the fixed feature extractor.

(Refer Slide Time: 26:10)



You can see that training and validation loss and accuracies are quite close by after the 10th epoch. You must be wondering why validation matrices are better than the training matrices. The main factor is because layers like tf.keras.layers.BatchNormalization and dropout affect accuracy during the training. They are turned off when calculating the validation loss. To a lesser extent it is also because training matrices report average for an epoch while validation matrix are evaluated after every epoch. So, validation matrix see a model that has trained slightly longer.

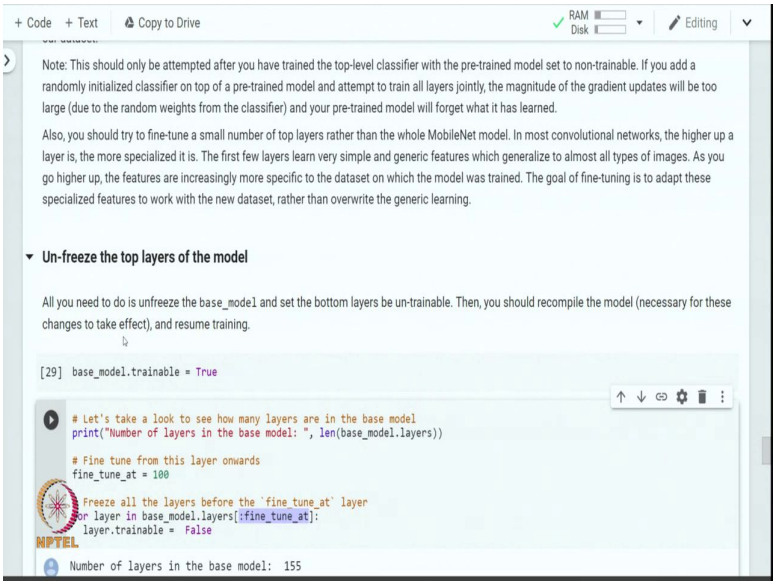
(Refer Slide Time: 26:50)



In our feature extraction experiment, we were only training a few layers on top of MobileNet base model. The weights of the pre-trained network were not updated during the training. The weights were frozen for the base model. One idea to improve the performance can be to fine tune the weights of the top layer of the pre-trained model alongside the classifier layer.

So, now, what we are going to do here is we are going to we are going to unfreeze these layers of the base network and train them along with the classifier layer. Let us see how to specify unfreezing of these layers in the code.

(Refer Slide Time: 28:00)

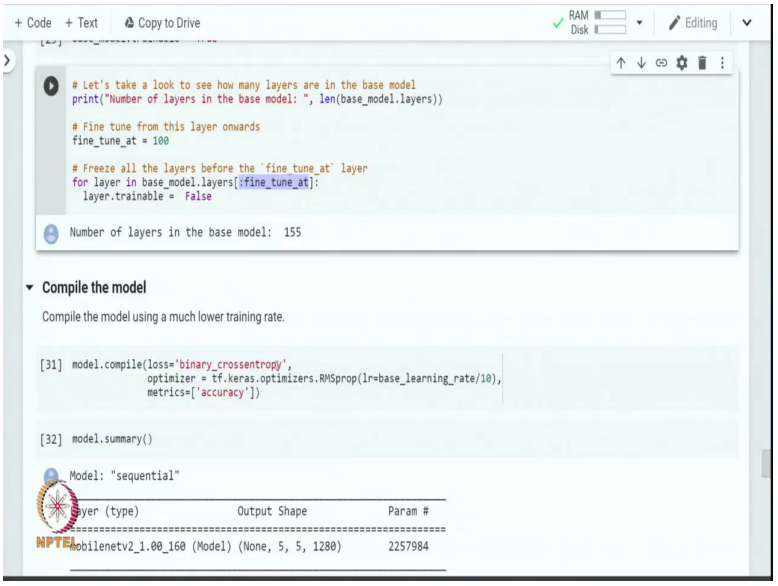


So, we set the trainable property of the base model to true and what we do is we specify the layers after which you want to unfreeze the network. So, here you want to find tune from 100th layer onwards. So, we specify the fine_tune_at variable and anything after that will be trained and anything before this fine_tune_at will be fixed. So, we set the trainable value of the layers before 100th layer as false. So, there are 155 layers in the base model out of which we retrain the top 55 layers.

It is important to note that as we go deeper in the network, network become more specialized to patterns learned from the training data. The initial convolution layer learns general patterns or simpler patterns like edges or corners, but as we go deeper and deeper the models become specialized to recognize patterns from the training data.

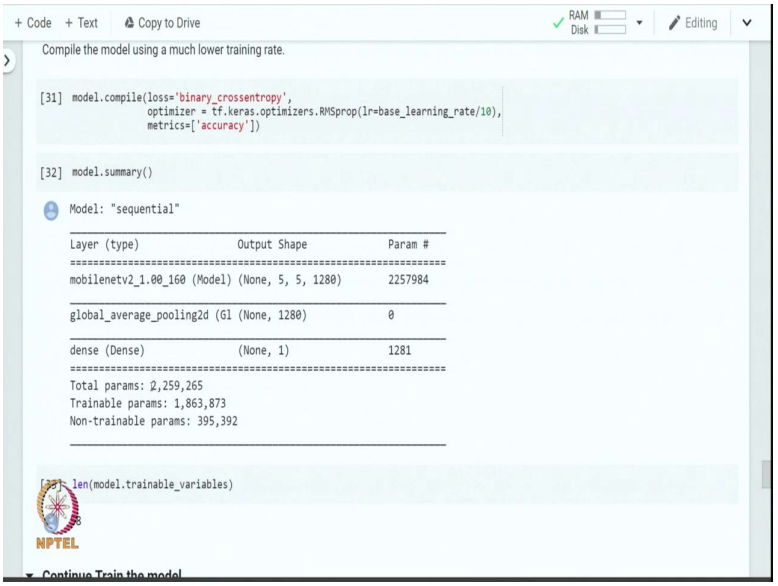
So, if we are using CNNs as the pre-trained model for some other task it is important to cut it somewhere in the middle if the task is very very different from the training data. For example, if the training data contains pictures of cats and dogs whereas, that the training data for the new task is about machine parts we might have to start unfreezing quite early in the network, because the original network which was trained on cats and dogs will start recognizing patterns related to cats and dog as it goes deeper.

(Refer Slide Time: 30:20)



So, let us use the same setting we use binary towards entropy loss and we use RMSprop as an optimizer. Let us look at the model summary after unfreezing some layers of the network.

(Refer Slide Time: 30:35)

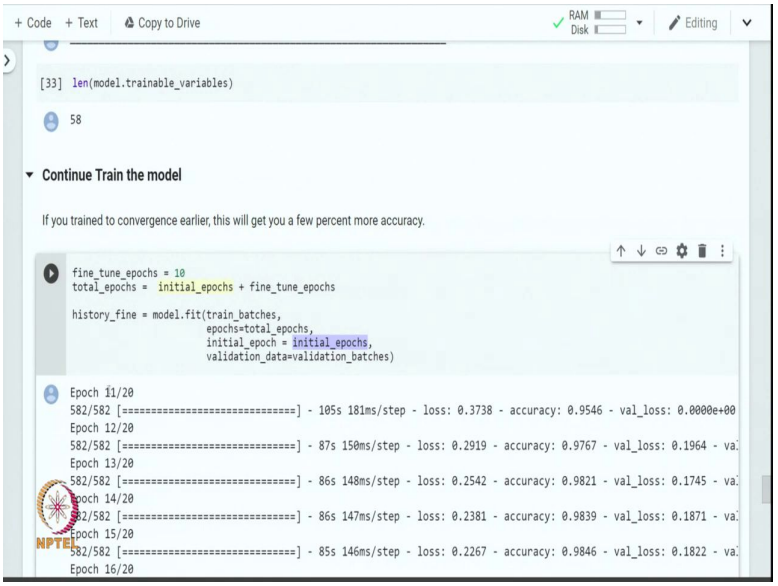


Now, you can see that out of 2.2 million total parameters, the non-trainable parameters are reduced to 395000 and the rest of the parameters are trainable parameters. Let us compare this model summary with the model summary from the previous exercise.

So, you can see that in the previous exercise we only had 1281 trainable parameters and all of the 2.2 million parameters of the base convolution neural network were non-trainable, but since we have unfrozen some of the layers, the number of parameters have gone up.

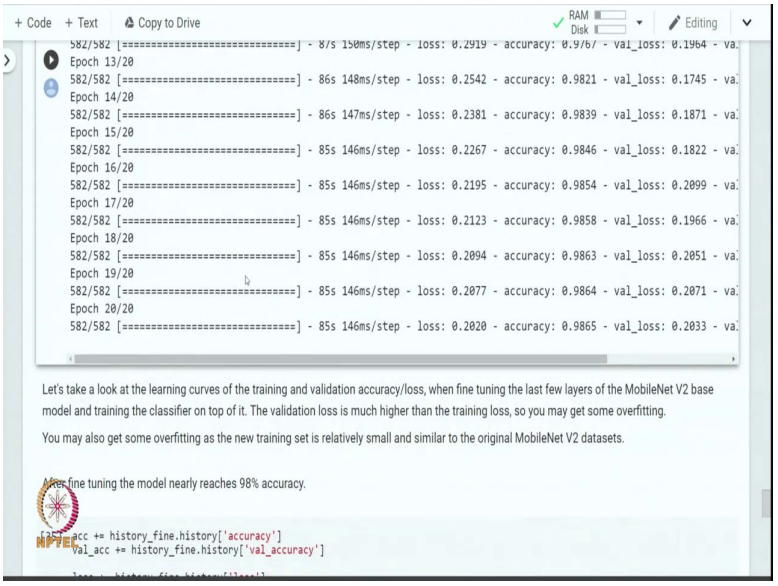
We should note that we should only attempt fine tuning after training the top level classifier with the pre-trained model set to non-trainable. If you add a randomly initialized classifier on the top of the pre-trained model and attempt to train all the layers jointly, the magnitude of the gradient updates will be too large and our pre-trained model will forget what it has learned.

(Refer Slide Time: 31:55)



Let us train the model for 10 epochs. What we do is we start the training of the model where we had stopped in the previous run. So, we had stopped on 10th epoch. So, we want to initialize the model with the 10th epoch and resume the training from that. So, you can see that when we run this particular fit function or the model training we started training from 11th epoch.

(Refer Slide Time: 32:26)



Now, the accuracy has gone up to 99 percent; it has crossed 98 percent which is more than 4 percentage points from the previous exercise where we used MobileNet merely as a feature extraction mechanism.

So, in this module we learned how to use a pre-trained machine learning model as a feature extractor and we also learned how to fine tune a pre-trained model. Hope you had fun learning these concepts. In the next session, we will use TensorFlow hub for loading some of the pre-trained models and using them for feature extraction as well as for fine tuning.