## keras.fit () and keras.fit\_generator ()

* **keras.fit () and keras.fit\_generator ()** in Python are two separate deep learning libraries which can be used to train our machine learning and deep learning models. Both these functions can do the same task, but when to use which function is the main question.
* Keras.fit()
* **Syntax:**

fit(object, x = NULL, y = NULL, batch\_size = NULL, epochs = 10,

verbose = getOption("keras.fit\_verbose", default = 1),

callbacks = NULL, view\_metrics = getOption("keras.view\_metrics",

default = "auto"), validation\_split = 0, validation\_data = NULL,

shuffle = TRUE, class\_weight = NULL, sample\_weight = NULL,

initial\_epoch = 0, steps\_per\_epoch = NULL, validation\_steps = NULL,

...)

* **Understanding few important arguments:**

-> **object** : the model to train.

-> **X** : our training data. Can be Vector, array or matrix

-> **Y** : our training labels. Can be Vector, array or matrix

-> **Batch\_size** : it can take any integer value or NULL and by default, it will

be set to 32. It specifies no. of samples per gradient.

-> **Epochs** : an integer and number of epochs we want to train our model for.

-> **Verbose** : specifies verbosity mode(0 = silent, 1= progress bar, 2 = one

line per epoch).

-> **Shuffle** : whether we want to shuffle our training data before each epoch.

-> **steps\_per\_epoch** : it specifies the total number of steps taken before

one epoch has finished and started the next epoch. By default it values is set to NULL.

**How to use Keras fit:**

model.fit(Xtrain, Ytrain, batch\_size = 32, epochs = 100)

Here we are first feeding the training data(Xtrain) and training labels(Ytrain). We then use Keras to allow our model to train for 100 epochs on a batch\_size of 32.

When we call the .fit() function it makes assumptions:

* The entire training set can fit into the Random Access Memory (RAM) of the computer.
* Calling the model. fit method for a second time is not going to reinitialize our already trained weights, which means we can actually make consecutive calls to fit if we want to and then manage it properly.
* There is no need for using the Keras generators(i.e no data argumentation)
* Raw data is itself used for training our network and our raw data will only fit into the memory.

**The Keras.fit\_generator():**

**Syntax:**

fit\_generator(object, generator, steps\_per\_epoch, epochs = 1,

verbose = getOption("keras.fit\_verbose", default = 1),

callbacks = NULL, view\_metrics = getOption("keras.view\_metrics",

default = "auto"), validation\_data = NULL, validation\_steps = NULL,

class\_weight = NULL, max\_queue\_size = 10, workers = 1,

initial\_epoch = 0)

**Understanding few important arguments:**

-> **object** : the Keras Object model.

-> **generator** : a generator whose output must be a list of the form:

- (inputs, targets)

- (input, targets, sample\_weights)

a single output of the generator makes a single batch and hence all arrays in the list

must be having the length equal to the size of the batch. The generator is expected

to loop over its data infinite no. of times, it should never return or exit.

-> **steps\_per\_epoch** : it specifies the total number of steps taken from the generator

as soon as one epoch is finished and next epoch has started. We can calculate the value

of steps\_per\_epoch as the total number of samples in your dataset divided by the batch size.

-> **Epochs** : an integer and number of epochs we want to train our model for.

-> **Verbose** : specifies verbosity mode(0 = silent, 1= progress bar, 2 = one line per epoch).

-> **callbacks** : a list of callback functions applied during the training of our model.

-> **validation\_data** can be either:

- an inputs and targets list

- a generator

- an inputs, targets, and sample\_weights list which can be used to evaluate

the loss and metrics for any model after any epoch has ended.

-> **validation\_steps** :only if the validation\_data is a generator then only this argument

can be used. It specifies the total number of steps taken from the generator before it is

stopped at every epoch and its value is calculated as the total number of validation data points

in your dataset divided by the validation batch size.

**How to use Keras fit\_generator:**

# performing data argumentation by training image generator

dataAugmentaion = ImageDataGenerator(rotation\_range = 30, zoom\_range = 0.20,

fill\_mode = "nearest", shear\_range = 0.20, horizontal\_flip = True,

width\_shift\_range = 0.1, height\_shift\_range = 0.1)

# training the model

model.fit\_generator(dataAugmentaion.flow(trainX, trainY, batch\_size = 32),

validation\_data = (testX, testY), steps\_per\_epoch = len(trainX) // 32,

epochs = 10)

Here we are training our network for 10 epochs along with the default batch size of 32.

For small and less complex datasets it is recommended to use keras.fit function whereas while dealing with real-world datasets it is not that simple because real-world datasets are huge in size and are much harder to fit into the computer memory.  
It is more challenging to deal with those datasets and an important step to deal with those datasets is to perform data augmentation to avoid the overfitting of a model and also to increase the ability of our model to generalize.

Data Augmentation is a method of artificially creating a new dataset for training from the existing training dataset to improve the performance of deep learning neural networks with the amount of data available. It is a form of regularization which makes our model generalize better than before.  
Here we have used a Keras ImageDataGenerator object to apply data augmentation for randomly translating, resizing, rotating, etc the images. Each new batch of our data is randomly adjusting according to the parameters supplied to ImageDataGenerator.

When we call the .fit\_generator() function it makes assumptions:

* Keras is first calling the generator function(dataAugmentaion)
* Generator function(dataAugmentaion) provides a batch\_size of 32 to our .fit\_generator() function.
* our .fit\_generator() function first accepts a batch of the dataset, then performs backpropagation on it, and then updates the weights in our model.
* For the number of epochs specified(10 in our case) the process is repeated.

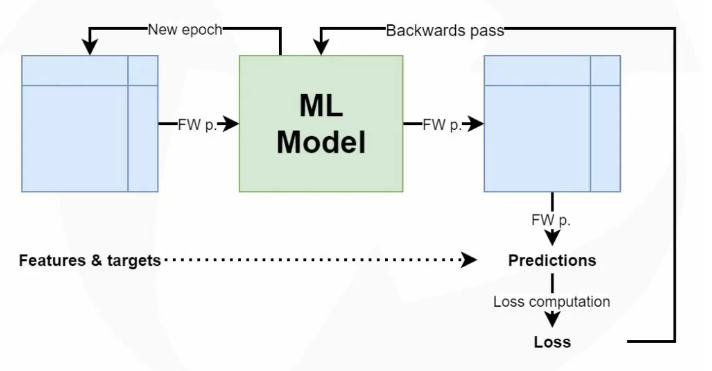
Summary :  
So, we have learned the difference between Keras.fit and Keras.fit\_generator functions used to train a deep learning neural network  
.fit is used when the entire training dataset can fit into the memory and no data augmentation is applied.  
.fit\_generator is used when either we have a huge dataset to fit into our memory or when data augmentation needs to be applied.

Call Back API

* [Callbacks and their role in the training process](https://www.machinecurve.com/index.php/2020/11/10/an-introduction-to-tensorflow-keras-callbacks/#callbacks-and-their-role-in-the-training-process)
* [The Keras Callbacks API](https://www.machinecurve.com/index.php/2020/11/10/an-introduction-to-tensorflow-keras-callbacks/#the-keras-callbacks-api)
  + [How do we add a callback to a Keras model?](https://www.machinecurve.com/index.php/2020/11/10/an-introduction-to-tensorflow-keras-callbacks/#how-do-we-add-a-callback-to-a-keras-model)
  + [ModelCheckpoint callback](https://www.machinecurve.com/index.php/2020/11/10/an-introduction-to-tensorflow-keras-callbacks/#modelcheckpoint-callback)
  + [TensorBoard callback](https://www.machinecurve.com/index.php/2020/11/10/an-introduction-to-tensorflow-keras-callbacks/#tensorboard-callback)
  + [EarlyStopping callback](https://www.machinecurve.com/index.php/2020/11/10/an-introduction-to-tensorflow-keras-callbacks/#earlystopping-callback)
  + [LearningRateScheduler callback](https://www.machinecurve.com/index.php/2020/11/10/an-introduction-to-tensorflow-keras-callbacks/#learningratescheduler-callback)
  + [ReduceLROnPlateau callback](https://www.machinecurve.com/index.php/2020/11/10/an-introduction-to-tensorflow-keras-callbacks/#reducelronplateau-callback)
  + [RemoteMonitor callback](https://www.machinecurve.com/index.php/2020/11/10/an-introduction-to-tensorflow-keras-callbacks/#remotemonitor-callback)
  + [LambdaCallback](https://www.machinecurve.com/index.php/2020/11/10/an-introduction-to-tensorflow-keras-callbacks/#lambdacallback)
  + [TerminateOnNaN callback](https://www.machinecurve.com/index.php/2020/11/10/an-introduction-to-tensorflow-keras-callbacks/#terminateonnan-callback)
  + [CSVLogger callback](https://www.machinecurve.com/index.php/2020/11/10/an-introduction-to-tensorflow-keras-callbacks/#csvlogger-callback)
  + [ProgbarLogger callback](https://www.machinecurve.com/index.php/2020/11/10/an-introduction-to-tensorflow-keras-callbacks/#progbarlogger-callback)
  + [Experimental: BackupAndRestore callback](https://www.machinecurve.com/index.php/2020/11/10/an-introduction-to-tensorflow-keras-callbacks/#experimental-backupandrestore-callback)
  + [Applied by default: History and BaseLogger callbacks](https://www.machinecurve.com/index.php/2020/11/10/an-introduction-to-tensorflow-keras-callbacks/#applied-by-default-history-and-baselogger-callbacks)
* [Creating your own callback with the Base Callback](https://www.machinecurve.com/index.php/2020/11/10/an-introduction-to-tensorflow-keras-callbacks/#creating-your-own-callback-with-the-base-callback)

# Callbacks and their role in the training process

* A machine learning model (today, often a neural network) is [initialized](https://www.machinecurve.com/index.php/2019/08/22/what-is-weight-initialization/).
* Samples from the training set are fed forward, through the model, resulting in a set of predictions.
* The predictions are compared with what is known as the ground truth (i.e. the labels corresponding to the training samples), resulting in one value – a [loss value](https://www.machinecurve.com/index.php/2019/10/04/about-loss-and-loss-functions) – telling us how bad the model performs.
* Based on the [loss](https://www.machinecurve.com/index.php/2019/10/04/about-loss-and-loss-functions/) value and the subsequent backwards computation of the error, the weights are changed a little bit, to make the model a bit better. Then, we’re either moving back to step 2, or we stop the training process.
* As we can see, steps 2-4 are iterative, meaning that the model improves in a cyclical fashion. This is reflected in the figure below.

**

* In Machine Learning terms, each iteration is also called an **epoch**. Hence, training a machine learning model involves the completion of at least one, but often multiple epochs. Note from the article about [gradient descent based optimization](https://www.machinecurve.com/index.php/2019/10/24/gradient-descent-and-its-variants/) that we often don’t feed forward all data at once. Instead, we use what is called a minibatch approach – the entire batch of data is fed forward in smaller batches called minibatches. By consequence, each epoch consists of at least one but often multiple **batches** of data.
* Now, it can be the case that you want to get insights from the training process while it is running. Or you want to provide automated steering in order to avoid wasting resources. In those cases, you might want to add a **callback** to your Keras model.
* A callback is an object that can perform actions at various stages of training (e.g. at the start or end of an epoch, before or after a single batch, etc).

As we shall see later in this article, among others, there are [callbacks for monitoring](https://www.machinecurve.com/index.php/2019/11/13/how-to-use-tensorboard-with-keras/) and for stopping the training process [when it no longer makes the model better](https://www.machinecurve.com/index.php/2019/05/30/avoid-wasting-resources-with-earlystopping-and-modelcheckpoint-in-keras/). This is possible because with callbacks, we can ‘capture’ the training process while it is happening. They essentially ‘hook’ into the training process by allowing the training process to invoke certain callback definitions. In Keras, each callback implements at least one, but possibly multiple of the following definitions (Keras Team, n.d.).

* With the on\_train\_begin and on\_train\_end definitions, we can perform a certain action either when model.fit starts executing or when the training process has just ended.
* With the on\_epoch\_begin and on\_epoch\_end definitions, we can perform a certain action just before the start of an epoch, or directly after it has ended.
* With the on\_test\_begin and on\_test\_end definitions, we can perform a certain action just before or after the model [is evaluated](https://www.machinecurve.com/index.php/2020/11/03/how-to-evaluate-a-keras-model-with-model-evaluate/).
* With the on\_predict\_begin and on\_predict\_end definitions, we can do the same, but then when we generate [new predictions](https://www.machinecurve.com/index.php/2020/02/21/how-to-predict-new-samples-with-your-keras-model/). If we [predict](https://www.machinecurve.com/index.php/2021/02/10/how-to-predict-new-samples-with-your-pytorch-model/) for a batch rather than a single sample, we can use the on\_predict\_batch\_begin and on\_predict\_batch\_end definitions.
* With the on\_train\_batch\_begin, on\_train\_batch\_end, on\_test\_batch\_begin and on\_test\_batch\_end definitions, we can perform a certain action directly before or after we feed a batch to either the training or testing process.

## The Keras Callbacks API

Now that we understand what callbacks are, how they can help us, and what definitions – and hence hooks – are available for ‘breaking into’ your training process in TensorFlow 2.x based Keras. Now, it’s time to take a look at the Keras Callbacks API. Available as tensorflow.keras.callbacks, it’s a set of generally valuable Callbacks that can be used in a variety of cases.

Most specifically, it contains the following callbacks, and we will cover each of them next:

1. [ModelCheckpoint](https://www.machinecurve.com/index.php/2019/05/30/avoid-wasting-resources-with-earlystopping-and-modelcheckpoint-in-keras/) callback: can be used to [automatically save a model](https://www.machinecurve.com/index.php/2019/05/30/avoid-wasting-resources-with-earlystopping-and-modelcheckpoint-in-keras/) after each epoch, or just the best one.
2. [TensorBoard](https://www.machinecurve.com/index.php/2019/11/13/how-to-use-tensorboard-with-keras/) callback: allows us to monitor the training process in realtime with [TensorBoard](https://www.machinecurve.com/index.php/2019/11/13/how-to-use-tensorboard-with-keras/).
3. [EarlyStopping](https://www.machinecurve.com/index.php/2019/05/30/avoid-wasting-resources-with-earlystopping-and-modelcheckpoint-in-keras/) callback: ensures that the training process stops if the loss value [does no longer improve](https://www.machinecurve.com/index.php/2019/05/30/avoid-wasting-resources-with-earlystopping-and-modelcheckpoint-in-keras/).
4. LearningRateScheduler callback: updates the [learning rate](https://www.machinecurve.com/index.php/2019/11/06/what-is-a-learning-rate-in-a-neural-network/) before the start of an epoch, based on a scheduler function.
5. ReduceLROnPlateau callback: reduces learning rate if the loss value does no longer improve.
6. RemoteMonitor callback: sends TensorFlow training events to a remote monitor, such as a logging system.
7. LambdaCallback: allows us to define simple functions that can be executed as a callback.
8. TerminateOnNaN callback: if the loss value is Not a Number (NaN), the training process stops.
9. CSVLogger callback: streams the outcome of an epoch to a CSV file.
10. ProgbarLogger callback: used to determine what is printed to standard output in the Keras progress bar.

* Often, when training a very deep neural network, we want to stop training once the training accuracy reaches a certain desired threshold. Thus, we can achieve what we want (optimal model weights) and avoid wastage of resources (time and computation power). In this brief tutorial, let’s learn how to achieve this in Tensorflow and Keras
* Training Deep Learning models without callbacks is like driving an airplane with no control over speed and altitude — you have little to no control over the whole process that is very likely to result in a disaster. In this article, you will learn how to monitor and improve your Deep Learning models using Keras callbacks like ModelCheckpoint and EarlyStopping
* In Deep Learning models [Keras callbacks functions](https://analyticsindiamag.com/hands-on-guide-to-implementing-alexnet-with-keras-for-multi-class-image-classification/) can play a very significant role. The training of such models can take even days to complete so we should have some function to monitor and control our model. Suppose, if the model is getting overfitted we can stop the training or if we have reached at least loss and for next epoch, it gets increased we can again stop the training.
* Sometimes due to much complexity in deep learning models, they often get crashed and the training gets stopped. Consider you have already trained it for 3 days and all the training gets wasted. To overcome these kinds of situations [Keras](https://analyticsindiamag.com/tutorial-on-keras-tokenizer-for-text-classification-in-nlp/) has several different callbacks functions that can help to get rid of these problems while training the model.
* Sometimes due to much complexity in deep learning models, they often get crashed and the training gets stopped. Consider you have already trained it for 3 days and all the training gets wasted. To overcome these kinds of situations [Keras](https://analyticsindiamag.com/tutorial-on-keras-tokenizer-for-text-classification-in-nlp/) has several different callbacks functions that can help to get rid of these problems while training the model.

### How do we add a callback to a Keras model?

Before we take a look at all the individual callbacks, we must take a look at how we can use the tensorflow.keras.callbacks API in the first place. Doing so is really simple and only changes your code in a minor way:

1. You must add the specific callbacks to the model imports.
2. You must initialize the callbacks you want to use, including their configuration; preferably do so in a list.
3. You must add the callbacks to the model.fit call.

With those three simple steps, you ensure that the callbacks are hooked into the training process!

step (1), we first **add the imports**:

from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint

Then, for step (2), we **initialize the callbacks** in a list:

keras\_callbacks = [

EarlyStopping(monitor='val\_loss', patience=5, mode='min', min\_delta=0.01),

ModelCheckpoint(checkpoint\_path, monitor='val\_loss', save\_best\_only=True, mode='min')

]

# And then, for step (3), we simply add the callbacks to model.fit:

model.fit(train\_generator,

epochs=50,

verbose=1,

callbacks=keras\_callbacks,

validation\_data=val\_generator)

# What is callback in Keras

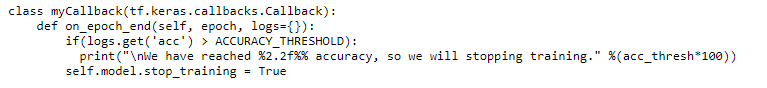
A callback is a set of functions to be applied at given stages of the training procedure. You can use callbacks to get a view on internal states and statistics of the model during training.

First, set the accuracy threshold to which you want to train your model.

acc\_thresh = 0.96

For implementing the callback first you have to create class and function.

Now, let us implement it to stop training when accuracy reaches acc\_thresh.



We are creating a new class by extending tf.keras.callbacks.Callback and implementing the on\_epoch\_end() method which will invoke at the end of an epoch.

Next, we are fetching the value of accuracy at the end of that epoch, and if it is greater than our threshold, we are setting the stop\_training of model to True using the “self” keyword to access the attributes and methods of the class in python.

Now, let us create the instance of an object of myCallback class.

callbacks = myCallback()

Next, build a Neural Network or Conv-Net or any model following the normal steps of TensorFlow or Keras. You can pass callbacks (as the keyword argument callbacks) to any of tf.keras.Model.fit(), tf.keras.Model.evaluate(), and tf.keras.Model.predict() methods. The methods of the callbacks will then be called at different stages of training/evaluating/inference.

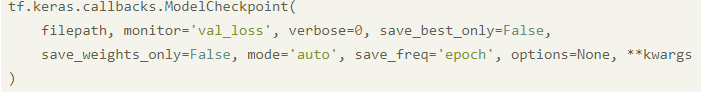
callbacks=[the newly instantiated object of myCallback class]

model.fit(x\_train, y\_train, epochs=20, callbacks=[callbacks])

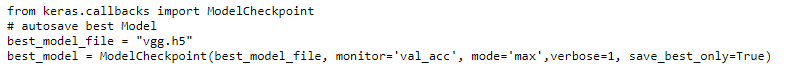
While training, as soon as accuracy reaches the value set in acc\_thresh, training will be stopped. In the above example, you can also use on\_epoch\_begin() which is called at the beginning of an epoch during training.

### ModelCheckpoint callback

* If you want to periodically save your Keras model – or the model weights – to some file, the ModelCheckpoint callback is what you need.
* Callback to save the Keras model or model weights at some frequency.
* This function of keras callbacks is used to save the model after every epoch. We just need to define a few of the parameters like where we want to store, what we want to monitor and etc.

**

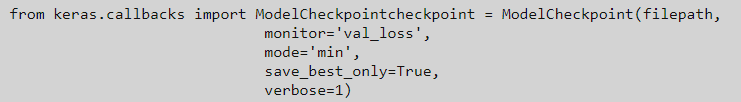
* Let’s say you want to save your model when validation accuracy reaches a minimum in between training before overfitting this can be implemented using the ModelCheckpoint in a callback which can be done as shown below.



Here we are monitoring the validation accuracy say you want training accuracy then just change the monitor value as ‘acc’ as shown below you can also put ‘loss’ for training loss and ‘val\_loss’ for validation accuracy and mode can be min or max depending wheater it is accuracy or loss.

This callback saves the model after every epoch. Here are some relevant metrics:

* filepath: the file path you want to save your model in
* monitor: the value being monitored
* save\_best\_only: set this to True if you do not want to overwrite the latest best model
* mode: auto, min, or max. For example, you set mode=’min’ if the monitored value is val\_loss and you want to minimize it.

**

With the following arguments:With filepath, you can specify where the model must be saved.

* If you want to save only if some quantity has changed, you can set this quantity by means of monitor. It is set to validation loss by default.
* With verbose, you can specify if the callback output should be output in your standard output (often, your terminal).
* If you only want to save the model when the monitored quantity improves, you can set save\_best\_only to True.
* Normally, the entire model is [saved](https://www.machinecurve.com/index.php/2020/02/14/how-to-save-and-load-a-model-with-keras/) – that is, the stack of layers as well as the [model weights](https://www.machinecurve.com/index.php/2019/08/22/what-is-weight-initialization/). If you want to save the weights only (e.g. because you can initialize the model yourself), you can set save\_weights\_only to True.
* With mode, you can determine in what direction the monitor quantity must move to consider it to be an improvement. You can choose any from {auto, min, max}. When it is set to auto, it determines the mode based on the monitor – with loss, for example, it will be min; with accuracy, it will be max.
* The save\_freq allows you to determine when to save the model. By default, it is saved after every epoch (or checks whether it has improved after every epoch). By changing the 'epoch' string into an integer, you can also instruct Keras to save after every n minibatches.
* If you want, you can specify other compatible options as well. Check the ModelCheckpoint docs (see link in references) for more information about these options.

checkpoint\_path=f'{os.path.dirname(os.path.realpath(\_\_file\_\_))}/covid-[convnet](https://www.machinecurve.com/index.php/2019/09/17/how-to-create-a-cnn-classifier-with-keras/).h5'

keras\_callbacks = [

ModelCheckpoint(checkpoint\_path, monitor='val\_loss', save\_best\_only=True, mode='min')

]

model.fit(train\_generator,

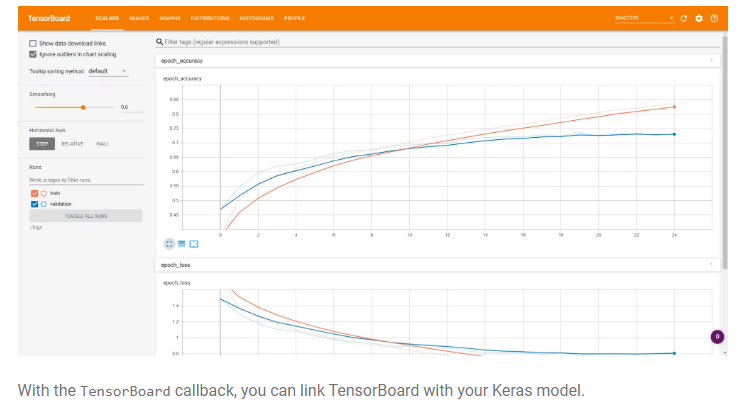
epochs=50,

verbose=1,

callbacks=keras\_callbacks,

validation\_data=val\_generator)

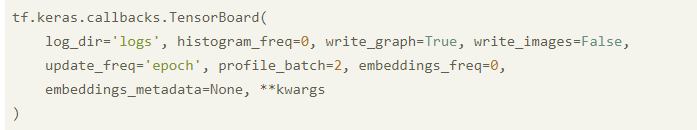
# 2.TensorBoard

**

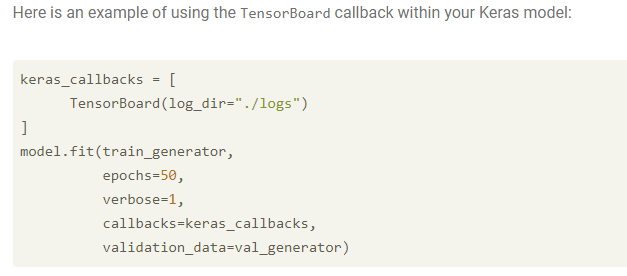
Enable visualizations for TensorBoard.

The callback logs a range of items from the training process into your TensorBoard log location:

* Metrics summary plots
* Training graph visualization
* Activation histograms
* Sampled profiling

**

* With log\_dir, you can specify the file path to your TensorBoard log folder.
* The TensorBoard callback computes activation and weight histograms. With histogram\_freq, you can specify the frequency (in epochs) when this should happen. Histograms will not be computed when histogram\_freq is set to 0.
* Whether to write the TensorFlow graph to the logs can be configured with write\_graph.
* If you want to visualize your model weights as images in TensorBoard, you can set write\_images to True.
* With update\_freq, you can specify when this callback sends data to TensorBoard. If it’s set to epoch, it will send data every epoch. If set to batch, data will be sent on every batch. If set to an integer n instead, data will be sent every n batches.
* With the [TensorFlow Profiler](https://www.tensorflow.org/guide/profiler" \t "_blank), we can calculate the compute performance of TensorFlow – that is, the resources it needs at a point in time. With profile\_batch, you can specify a batch to profile, meaning that Profiling information will be sent to TensorBoard as well.
* If you are using [Embeddings](https://www.machinecurve.com/index.php/2020/03/03/classifying-imdb-sentiment-with-keras-and-embeddings-dropout-conv1d/), it is possible to let TensorFlow visualize them. Specifying the embeddings\_freq allows you to configure when Embeddings need to be visualized; it represents the frequency in epochs. Embeddings will not be visualized when the frequency is set to 0.
* A dictionary with Embeddings metadata can be passed along with embeddings\_metadata.

**

This is the best of all callbacks. By using a TensorBoard callback, logs will be written to a directory that you can then examine with TensorFlow’s excellent TensorBoard visualization tool.

This line creates a Callback Tensorboard object, you should capture that object and give it to the fit function of your model.

tbCallBack = keras.callbacks.TensorBoard(log\_dir=path\_to\_your\_logs, histogram\_freq=0, write\_graph=True, write\_images=False)`  
...  
model.fit(...inputs and parameters..., callbacks=[tbCallBack])

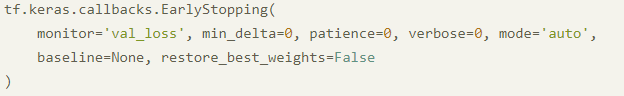
* histogram\_freq: frequency (in epochs) at which to compute activation and weight histograms for the layers of the model. If set to 0, histograms won't be computed. Validation data (or split) must be specified for histogram visualizations.
* write\_graph: whether to visualize the graph in TensorBoard. The log file can become quite large when write\_graph is set to True.
* write\_images: whether to write model weights to visualize as image in TensorBoard.

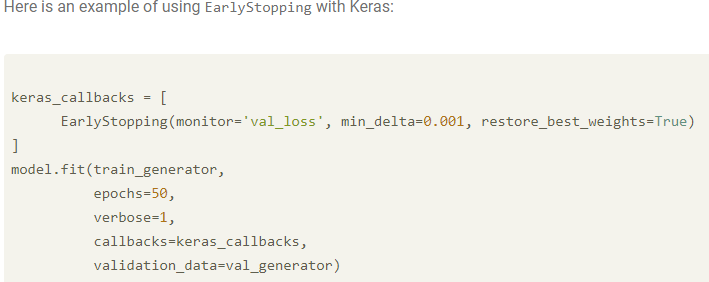
This way you gave your callback object to the function. It will be run during the training and will output files that can be used with tensorboard.

If you want to visualize the files created during training, run in your terminal.

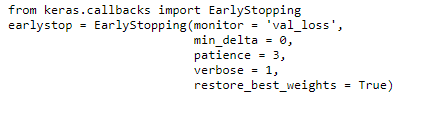
tensorboard --logdir=path\_to\_your\_logs

# 3.Early stopping at minimum loss

* Optimizing your neural network involves applying [gradient descent](https://www.machinecurve.com/index.php/2019/10/24/gradient-descent-and-its-variants/) or [another optimizer](https://www.machinecurve.com/index.php/2019/11/03/extensions-to-gradient-descent-from-momentum-to-adabound/) to a loss value generated by feeding forward batches of training samples, generating predictions that are compared with the corresponding training labels.
* During this process, you want to find a model that performs well in terms of predictions (i.e., it is not [underfit](https://www.machinecurve.com/index.php/2020/12/01/how-to-check-if-your-deep-learning-model-is-underfitting-or-overfitting/)) but that is not too rigid with respect to the dataset it is trained on (i.e., it is neither [overfit](https://www.machinecurve.com/index.php/2020/12/01/how-to-check-if-your-deep-learning-model-is-underfitting-or-overfitting/)). That’s why the EarlyStopping callback can be useful if you are dealing with a situation like this.
* Stop training when a monitored metric has stopped improving.
* Overfitting is a nightmare for Machine Learning practitioners. One way to avoid overfitting is to terminate the process early. The EarlyStoppingfunction has various metrics/arguments that you can modify to set up when the training process should stop.
* **
* The monitor is the quantity to monitor for improvement; it is similar to the quantity monitored for ModelCheckpointing.
* The same goes for the mode.
* With min\_delta, you can configure the minimum change that must happen from the current monitor in order to consider the change an improvement.
* With patience, you can indicate how long in epochs to wait for additional improvements before stopping the training process.
* With verbose, you can specify the verbosity of the callback, i.e. whether the output is written to standard output.
* The baseline value can be configured to specify a minimum monitor that must be achieved at all before any change can be considered an improvement.
* As you would expect, having a patience > 0 will ensure that the model is trained for patience more epochs, possibly making it worse. With restore\_best\_weights, we can restore the weights of the best-performing model instance when the training process stops. This can be useful if you directly perform [model evaluation](https://www.machinecurve.com/index.php/2020/11/03/how-to-evaluate-a-keras-model-with-model-evaluate/) after stopping the training process.

**

The code example below will define an EarlyStopping function that tracks the val\_loss value, stops the training if there are no changes towards val\_loss after 3 epochs, and keeps the best weights once the training stops.

**

monitor is the value being monitored, i.e: val\_loss, min\_delta is the minimum change in the monitored value and patience is the number of epochs with no improvement after which training will be stopped, restore\_best\_weights is the set this metric to True if you want to keep the best weights once stopped.

Here are some relevant metrics:

* monitor: value being monitored, i.e: val\_loss
* min\_delta: minimum change in the monitored value. For example, min\_delta=1 means that the training process will be stopped if the absolute change of the monitored value is less than 1
* patience: number of epochs with no improvement after which training will be stopped
* restore\_best\_weights: set this metric to True if you want to keep the best weights once stopped

The code example below will define an EarlyStopping function that tracks the val\_loss value, stops the training if there are no changes towards val\_loss after 3 epochs, and keeps the best weights once the training stops:

from keras.callbacks import EarlyStoppingearlystop = EarlyStopping(monitor = 'val\_loss',

min\_delta = 0,

patience = 3,

verbose = 1,

restore\_best\_weights = True)

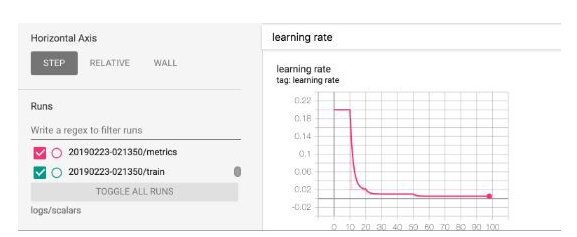
This function of Keras callbacks is used to stop the model training in between. This function is very helpful when your models get overfitted. It is used to stop the model as soon as it gets overfitted. We defined what to monitor while saving the model checkpoints. We also need to define the factor we want to monitor while using the early stopping function. We will monitor validation loss for stopping the model training. Use the below code to use the early stopping function.

from keras.callbacks import EarlyStopping

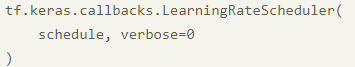
earlystop = EarlyStopping(monitor = 'val\_loss',min\_delta = 0,patience = 3, verbose = 1,restore\_best\_weights = True)

As we can see the model training has stopped after 10 epoch. This is the benefit of using early stopping.

# 4.LearningRateScheduler

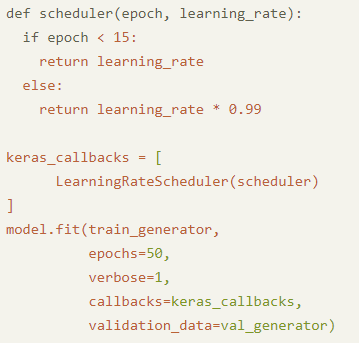
**

* During the optimization process, a so called weight update is computed. However, if we compare the optimization process with rolling a ball down a mountain (reflecting the [loss landscape](https://www.machinecurve.com/index.php/2020/02/26/getting-out-of-loss-plateaus-by-adjusting-learning-rates/)), we want to smooth the ride, ensuring that our ball does not bounce out of control. That is why a [learning rate](https://www.machinecurve.com/index.php/2019/11/06/what-is-a-learning-rate-in-a-neural-network/) is applied: it specifies a fraction of the weight update to be used by the [optimizer](https://www.machinecurve.com/index.php/2019/10/24/gradient-descent-and-its-variants/).
* Preferably being relatively large during the early iterations and lower in the later stages, we must adapt the learning rate during the training process. This is called [learning rate decay](https://www.machinecurve.com/index.php/2019/11/11/problems-with-fixed-and-decaying-learning-rates/) and shows what a learning rate scheduler can be useful for. The LearningRateScheduler callback implements this functionality.
* This one is pretty straightforward: it adjusts the learning rate over time using a schedule that you already write beforehand. This function returns the desired learning rate (output) based on the current epoch (epoch index as input).
* This is a very simple function of callback that can be used to tweak the learning rate over a while. This is scheduled before the training. This gives us the desired output based on the respective epoch.
* At the beginning of every epoch, this callback gets the updated learning rate value from schedule function provided at \_\_init\_\_, with the current epoch and current learning rate, and applies the updated learning rate on the optimizer.

**

* It accepts a schedule function which you can use to decide yourself how the learning rate must be scheduled during every epoch.
* With verbose, you can decide to illustrate the callback output in your standard output.

Here is an example of using the LearningRateScheduler with Keras:

**

from keras.callbacks import LearningRateScheduler

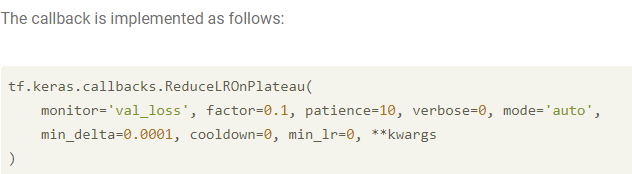
lrs = LearningRateScheduler(schedule, verbose=0) # schedule is a function

Use the below code to use the learning rate scheduler.

from keras.callbacks import LearningRateSchedulerscheduler = LearningRateScheduler(schedule, verbose=0)

### 5. ReduceLROnPlateau callback

* During the optimization process – i.e., rolling the ball downhill – it can be the case that you encounter so-called loss plateaus. In those areas, the gradient of the [loss function](https://www.machinecurve.com/index.php/2019/10/04/about-loss-and-loss-functions/) is close to zero, but not entirely – indicating that you are in the vicinity of a loss minimum. That is, close to where you want to be (unless you are dealing with a local minimum, of course).
* Keeping your learning rate equal when close to a plateau means that your model will likely not improve any further. This happens because your model will optimize, oscillating around the loss minimum, simply because the steps the current [learning rate](https://www.machinecurve.com/index.php/2019/11/06/what-is-a-learning-rate-in-a-neural-network/) it instructs to set are too big.
* With the ReduceLROnPlateau callback, the optimization process can be instructed to reduce the learning rate (and hence the step) when a plateau is encountered.
* Models often benefit from reducing the learning rate by a factor of 2-10 once learning stagnates. This callback monitors a quantity and if no improvement is seen for a ‘patience’ number of epochs, the learning rate is reduced.

**

* The monitor and patience resemble the monitors and patience values that we have already encountered. In other words, it is the quantity to observe that helps us judge whether improvement has happened. Patience tells us how long to wait before we consider improvement impossible. The mode is related to the monitor and instructs what kind of operation to perform while monitoring: min or max (or automatically determined).
* The min\_delta tells us how much the model should improve at minimum before we consider the change an improvement.
* The factor determines how much to decrease the learning rate upon encountering a plateau: new\_lr = lr \* factor.
* The verbose attribute can be configured to display the callback output in your standard output.
* The min\_lr gives us a lower bound on the learning rate.
* The cooldown attribute instructs the model to wait with invoking this specific callback for a number of epochs, allowing us to find some improvement with the reduced learning rate (this could take a few epochs).

keras\_callbacks = [

ReduceLROnPlateau(monitor='val\_loss', factor=0.25, patience=5, cooldown=5, min\_lr=0.000000001)

]

model.fit(train\_generator,

epochs=50,

verbose=1,

callbacks=keras\_callbacks,

validation\_data=val\_generator)

### 6.RemoteMonitor callback

* Above, we saw that training logs can be distributed to [TensorBoard](https://www.machinecurve.com/index.php/2019/11/13/how-to-use-tensorboard-with-keras/) for visualization and logging purposes. However, it can be the case that you have your own logging and visualization system – whether that’s a cloud-based system or a locally installed Grafana or Elastic Stack visualization tooling.
* In those cases, you might wish to send the training logs there instead. The RemoteMonitor callback can help you do this.

It is implemented as follows:

tf.keras.callbacks.RemoteMonitor(

root='http://localhost:9000', path='/publish/epoch/end/', field='data',

headers=None, send\_as\_json=False

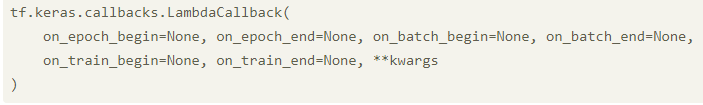
)

* With the root argument, you can specify the root of the endpoint to where data must be sent.
* The path indicates the path relative to root where data must be sent. In other words, root + path describe the full endpoint.
* The JSON field under which data is sent can be configured with field.
* In headers, additional HTTP headers (such as an Authorization header) can be provided.
* With send\_as\_json as True, the content type of the request will be changed to application/json. Otherwise, it will be sent as part of a form.

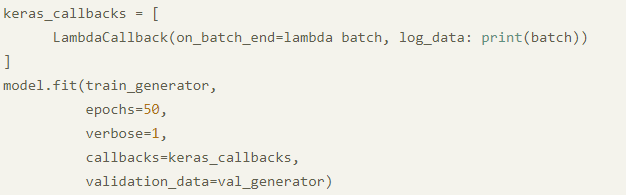
**

# 7.LambdaCallback

* Say that you want a certain function to fire after every batch or every epoch – a simple function, nothing special. However, it’s not provided in the collection of callbacks presented with the tensorflow.keras.callbacks API. In this case, you might want to use the LambdaCallback.
* Callback for creating simple, custom callbacks on-the-fly. This callback is constructed with anonymous functions that will be called at the appropriate time.
* If you want to create your own callback then you can use LambdaCallback. This allows you to trigger events when an epoch, batch, or training process begins or ends.
* It can thus be used to provide anonymous (i.e. lambda functions without a name) functions to the training process. The callback looks as follows:

**

* Here, the on\_epoch\_begin, on\_epoch\_end, on\_batch\_begin, on\_batch\_end, on\_train\_begin and on\_train\_end event based arguments can be filled with Python definitions. They are executed at the right point in time.
* An example of a LambdaCallback added to your Keras model:

**



As you can see above I have created a custom callback using LambdaCallback to show samples at the epoch end. And I have also illustrated the use of multiple callbacks in the above example. You can also see multiple examples of custom callbacks implemented in the main code at the link given below.

It’s a good idea to use more than one of these callbacks. I use **ModelCheckpoint** and **RemoteMonitor and TensorBoard mainly.**Callbacks like **EarlyStopping** and **LearningRateScheduler** are great for optimization. This is not the complete list but I have covered almost most of it.

# 8. TerminateOnNaN callback

In some cases (e.g. when you did not apply min-max normalization to your input data), the loss value can be very strange – outputting values close to Infinity or values that are Not a Number (NaN). In those cases, you don’t want to pursue further training. The TerminateOnNaN callback can help here.

Callback that terminates training when a NaN loss is encountered.

### *C:\Users\dell\Desktop\d.jpg*

### 9. CSVLogger callback

CSV files can be very useful when you need to exchange data. If you want to flush your training logs into a CSV file, the CSVLogger callback can be useful to you.

Callback that streams epoch results to a CSV file.

It is implemented as follows:

tf.keras.callbacks.CSVLogger(

filename, separator=',', append=False

)

* The filename attribute determines where the CSV file is located. If there is none, it will be created.
* The separator attribute determines what character separates the columns in a single row, and is also called delimiter.
* With append, you can indicate whether data should simply be added to the end of the file, or a new file should overwrite the old one every time.

This is an example of using the CSVLogger callback with Keras:

### *C:\Users\dell\Desktop\f.jpg*

### 10.ProgbarLogger callback

When you are training a Keras model with verbosity set to True, you will see a progress bar in your terminal. With the ProgbarLogger callback, you can change what is displayed there.

Callback that prints metrics to stdout.

It is implemented as follows:

tf.keras.callbacks.ProgbarLogger(

count\_mode='samples', stateful\_metrics=None

)

Code language: JavaScript (javascript)

* With count\_mode, you can instruct Keras to display samples or steps (i.e. batches) already fed forward through the model
* The stateful\_metrics attribute can contain metrics that should not be averaged over time.

Here is an example of using the ProgbarLogger callback with Keras.

keras\_callbacks = [

ProgbarLogger(count\_mode='samples')

]

model.fit(train\_generator,

epochs=50,

verbose=1,

callbacks=keras\_callbacks,

validation\_data=val\_generator)

### Experimental: BackupAndRestore callback

* When you are training a neural network, especially in a [distributed setting](https://www.machinecurve.com/index.php/2020/10/16/tensorflow-cloud-easy-cloud-based-training-of-your-keras-model/), it would be problematic if your training process suddenly stops – e.g. due to machine failure. Every iteration passed so far will be gone. With the experimental BackupAndRestore callback, you can instruct Keras to create temporary checkpoint files after each epoch, to which you can restore later.
* BackupAndRestore callback is intended to recover from interruptions that happened in the middle of a model.fit execution by backing up the training states in a temporary checkpoint file (based on TF CheckpointManager) at the end of each epoch.

It is implemented as follows:

tf.keras.callbacks.experimental.BackupAndRestore(

backup\_dir

)

Here, the backup\_dir attribute indicates the folder where checkpoints should be created.

Here is an example of using the BackupAndRestore callback with Keras.

keras\_callbacks = [

BackupAndRestore('./checkpoints')

]

model.fit(train\_generator,

epochs=50,

verbose=1,

callbacks=keras\_callbacks,

validation\_data=val\_generator)

### Applied by default: History and BaseLogger callbacks

There are two callbacks that are part of the tensorflow.keras.callbacks API but which can be covered less extensively – because of the simple reason that they are already applied to each Keras model under the hood.

They are the [History](https://www.machinecurve.com/index.php/2019/10/08/how-to-visualize-the-training-process-in-keras/) and BaseLogger callbacks.

* The History callback generates a History [object](https://www.machinecurve.com/index.php/2019/10/08/how-to-visualize-the-training-process-in-keras/#the-history-object) when calling model.fit.
* The BaseLogger callback accumulates basic metrics to display later.

### Creating your own callback with the Base Callback

* Sometimes, neither the default or the lambda callbacks can provide the functionality you need. In those cases, you can create your own callback, by using the Base callback class tensorflow.keras.callbacks.Callback. Creating one is very simple: you define a class, create the relevant definitions (you can choose from on\_epoch\_begin, on\_epoch\_end, on\_batch\_begin, on\_batch\_end, on\_train\_begin and on\_train\_end etc.), and then add the callback to your callbacks list. There you go!

class OwnCallback(tensorflow.keras.callbacks.Callback):

def on\_train\_begin(self, logs=None):

print('Training is now beginning!')

keras\_callbacks = [

OwnCallback()

]

model.fit(train\_generator,

epochs=50,

verbose=1,

callbacks=keras\_callbacks,

validation\_data=val\_generator)

### Other Callbacks functions

Along with the above functions, there are other callbacks that you might encounter or want to use in your Deep Learning project:

* **History**and**BaseLogger**: callbacks that are applied automatically to your model by default
* **TensorBoard**: This is hands down my favorite Keras callback. This callback writes a log for TensorBoard, which is TensorFlow’s excellent visualization tool. If you have installed TensorFlow with pip, you should be able to launch TensorBoard from the command line: tensorboard — logdir=/full\_path\_to\_your\_logs
* **CSVLogger**: This callback streams epoch results to a csv file
* **LambdaCallback**: This callback allows you to build custom callback